Master the Techniques You Need to Create Your Own Games



Secrets and techniques from expert game programmers!

Covers QuickDraw™ 3D, porting, networking, Power Macs and more!

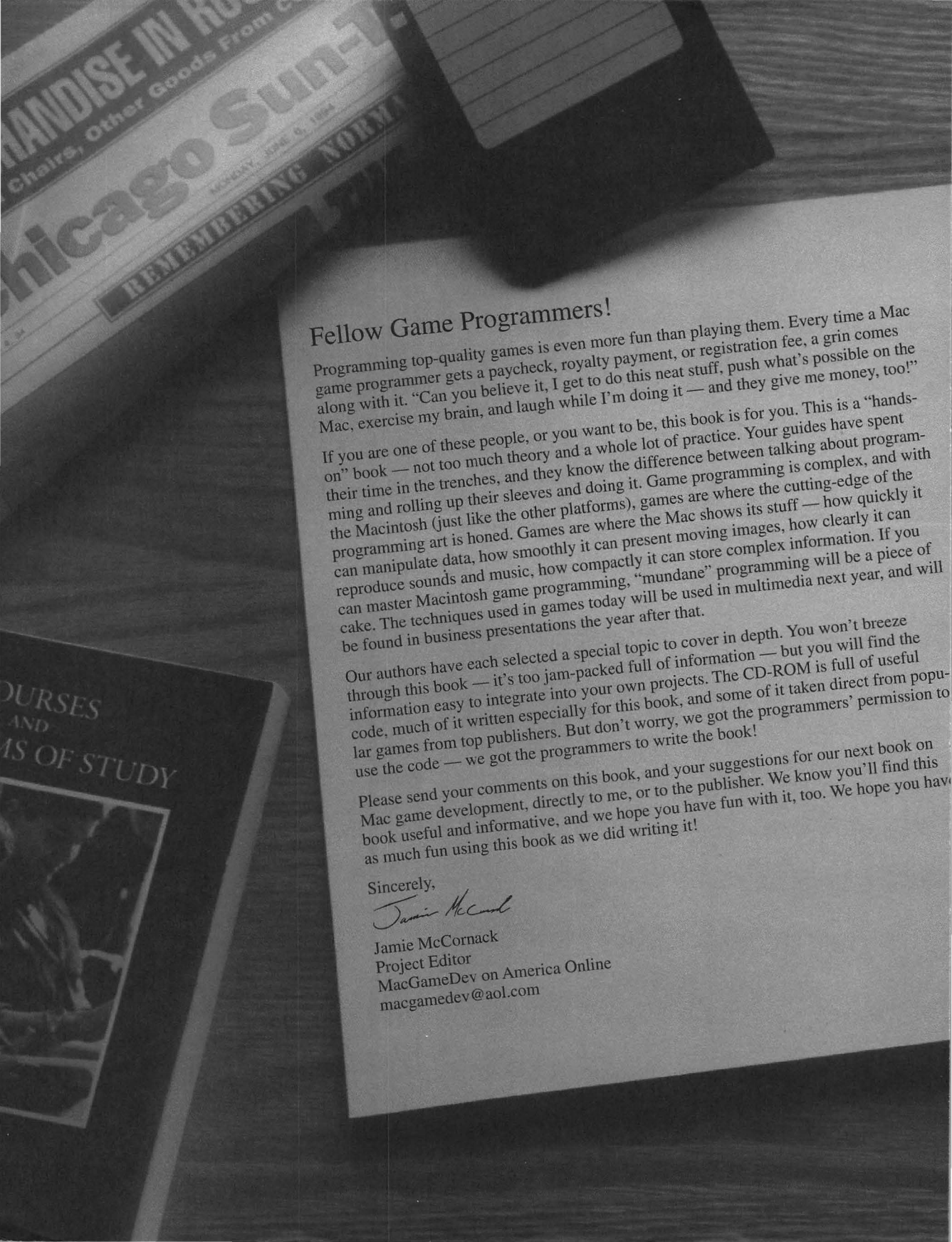
CD-ROM includes programming tools and utilities, sample code, games, demos, and more!



TRICKS OF THE MAC GAME PROGRAMMING GURUS

Jamie McCornack, Ingemar Ragnemalm, Paul Celestin, and many more game programming gurus

The Don Crabb Macintosh Library



Fellow Game Programmers!

Programming top-quality games is even more fun than playing them. Every time a Mac game programmer gets a paycheck, royalty payment, or registration fee, a grin comes along with it. "Can you believe it, I get to do this neat stuff, push what's possible on the Mac, exercise my brain, and laugh while I'm doing it — and they give me money, too!"

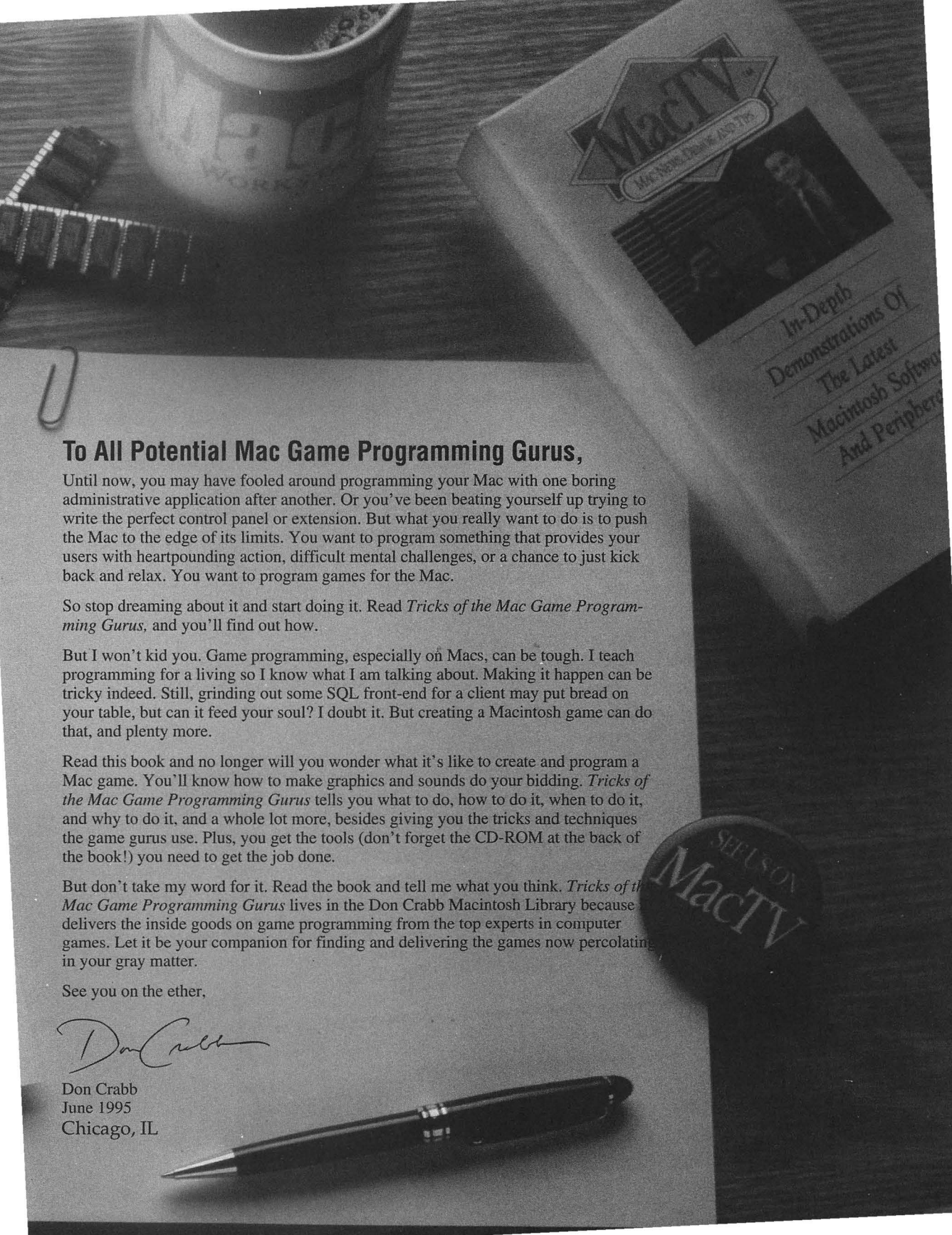
If you are one of these people, or you want to be, this book is for you. This is a "hands-on" book — not too much theory and a whole lot of practice. Your guides have spent their time in the trenches, and they know the difference between talking about programming and rolling up their sleeves and doing it. Game programming is complex, and with the Macintosh (just like the other platforms), games are where the cutting-edge of the programming art is honed. Games are where the Mac shows its stuff — how quickly it can manipulate data, how smoothly it can present moving images, how clearly it can reproduce sounds and music, how compactly it can store complex information. If you can master Macintosh game programming, "mundane" programming will be a piece of cake. The techniques used in games today will be used in multimedia next year, and will be found in business presentations the year after that.

Our authors have each selected a special topic to cover in depth. You won't breeze through this book — it's too jam-packed full of information — but you will find the information easy to integrate into your own projects. The CD-ROM is full of useful code, much of it written especially for this book, and some of it taken direct from popular games from top publishers. But don't worry, we got the programmers' permission to use the code — we got the programmers to write the book!

Please send your comments on this book, and your suggestions for our next book on Mac game development, directly to me, or to the publisher. We know you'll find this book useful and informative, and we hope you have fun with it, too. We hope you have as much fun using this book as we did writing it!

Sincerely,

Jamie McCornack
Project Editor
MacGameDev on America Online
macgamedev@aol.com



To All Potential Mac Game Programming Gurus,

Until now, you may have fooled around programming your Mac with one boring administrative application after another. Or you've been beating yourself up trying to write the perfect control panel or extension. But what you really want to do is to push the Mac to the edge of its limits. You want to program something that provides your users with heartpounding action, difficult mental challenges, or a chance to just kick back and relax. You want to program games for the Mac.

So stop dreaming about it and start doing it. Read *Tricks of the Mac Game Programming Gurus,* and you'll find out how.

But I won't kid you. Game programming, especially on Macs, can be tough. I teach programming for a living so I know what I am talking about. Making it happen can be tricky indeed. Still, grinding out some SQL front-end for a client may put bread on your table, but can it feed your soul? I doubt it. But creating a Macintosh game can do that, and plenty more.

Read this book and no longer will you wonder what it's like to create and program a Mac game. You'll know how to make graphics and sounds do your bidding. *Tricks of the Mac Game Programming Gurus* tells you what to do, how to do it, when to do it, and why to do it, and a whole lot more, besides giving you the tricks and techniques the game gurus use. Plus, you get the tools (don't forget the CD-ROM at the back of the book!) you need to get the job done.

But don't take my word for it. Read the book and tell me what you think. *Tricks of th Mac Game Programming Gurus* lives in the Don Crabb Macintosh Library because delivers the inside goods on game programming from the top experts in computer games. Let it be your companion for finding and delivering the games now percolatin in your gray matter.

See you on the ether,

Don Crabb
June 1995
Chicago, IL

Tricks of the Mac®
Game Programming
Gurus

by McCornack, Ragnemalm, Celestin, et al.



Tricks of the Mac® Game Programming Gurus

©1995 Hayden Books, a division of Macmillan Computer Publishing

All rights reserved. Printed in the United States of America. No part of this book may be used or reproduced in any form or by any means, or stored in a database or retrieval system, without prior written permission of the publisher except in the case of brief quotations embodied in critical articles and reviews. Making copies of any part of this book for any purpose other than your own personal use is a violation of United States copyright laws. For information, address Hayden Books, 201 W. 103rd Street, Indianapolis, Indiana 46290.

Library of Congress Catalog Number: 95-60416
ISBN: 1-56830-183-9

This book is sold as is, without warranty of any kind, either express or implied. While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information or instructions contained herein. It is further stated that the publisher and author are not responsible for any damage or loss to your data or your equipment that results directly or indirectly from your use of this book.

97 96 95 4 3 2

Interpretation of the printing code: The rightmost double-digit number is the year of the book's printing; the rightmost single-digit number is the number of the book's printing. For example, a printing code of 95-1 shows that the first printing of the book occurred in 1995.

Trademark Acknowledgments: All products mentioned in this book are either trademarks of the companies referenced in this book, registered trademarks of the companies referenced in this book, or neither. We strongly advise that you investigate a particular product's name thoroughly before you use the name as your own.

Apple, Mac, and Macintosh are registered trademarks of Apple Computer, Inc.

The Hayden Books Team

Publisher

David Rogelberg

Product Development Manager

Patrick Gibbons

Acquisitions Editor

Stacy Kaplan

Development Editor

Brad Miser

Copy/Production Editor

Steve Mulder

Technical Reviewers

Mark Adams
Stuart Cheshire
Dennis Cohen
Brian Greenstone
Jake Hoelter

Interviewer

Tom Thompson

Publishing Coordinator

Rosemary Lewis

Interior Designer

Fred Bower

Cover Designer

Jay Corpus

Production

Angela Calvert, Dan Caparo, Kim Cofer, Kevin Foltz, David Garratt, Shawn MacDonald, Joe Millay, Erika Millen, Beth Rago, Regina Rexrode, Erich J. Richter, Christine Tyner, Karen Walsh, Robert Wolf

Indexer

Bront Davis

To Our Readers

Dear Friend,

Thank you on behalf of everyone at Hayden Books for choosing *Tricks of the Mac Game Programming Gurus* to enable you to explore game programming on the Macintosh. For many reasons, the Mac is the best platform for games, and fortunately games on the Mac are exploding in both quality and popularity. This book will help you make your own contribution to this growth.

We'd love to see any games that this book helps you with. Please write and let us know. What you think of this book is important to our ability to better serve you in the future. If you have any comments, no matter how great or small, we'd appreciate you taking the time to send us email or a note by snail mail. Of course, we'd love to hear your book ideas.

Sincerely yours,



You can reach Hayden Books at the following:

Hayden Books
201 West 103rd Street
Indianapolis, IN 46290
(800) 428-5331 voice
(800) 448-3804 fax

Email addresses:

America Online: Hayden Bks
AppleLink: hayden.books
CompuServe: 76350,3014
Internet: hayden@hayden.com

Contents at a Glance

0 A Game Programming Primer 1
1 Your First Game 41
2 Basic Game Graphics 87
3 Advanced Graphics 121
4 File Management 209
5 User Interaction 265
6 Audio 327
7 Game Mechanics: Environments and Entities 389
8 Game Mechanics: Behavior 433
9 QuickDraw 3D 491
10 Networking 547
11 Porting 627
12 Game Aesthetics 709
13 Optimizing Game Code for PowerPC 731
14 Glypha III: A Case Study 749
A Top Speed—or Not? 797
B Macintosh Environments 811
Index 843

Table of Contents

0 A Game Programming Primer 1

Animation Basics 2

What's Animation? 2

How Quick Is Quick Enough? 2

How Smooth Is Smooth Enough? 4

How Fast Is Too Fast? 5

The HelloWorld Code 6

HelloWorld1: Bare Bones 6

HelloWorld2: Color and Complexity 16

HelloWorld3: Clams in Formation 23

HelloWorld4: Clams in Collision 25

HelloWorld5: Clams in Multibit 26

HelloWorld6: 8 Bits Preferred 28

HelloWorld6.1: No Brakes 30

Some Sound Advice 30

Sound Markers 31

Synchronous vs. Asynchronous Sound 32

MGWSound.c 32

What Do I Do Next? 39

1 Your First Game 41

Dungeon 1 41

Graphics 42

Sounds 43

Moving Around and Fighting 44

Random Numbers 45

Let's See Some Source Code! 45

Building Dungeon 46

Understanding Dungeon 46

Declarations, Constants, and Variables 46

Subroutines 49

The Main Program 58

But How about My Own First Game? 59

Problem 1: A Good Idea 60

Can You Do It? 61

Can the Computer Do It? 61

Can Your Players Do It? 64

All Those Games 65
2-D Space Games 65
Platform Games 66
Side-Scrollers 67
Board Games and Puzzle Games 69
Card Games 70
First-Person 3-D Action Games 71
Adventure Games 72
Dungeon-Digging Games 74
Abstract Action Games 75
Simulation Games 77
Strategy Games 77
Programmer's Games 79
Dungeon 1 Plus 80
Map Generation 80
Source Code Additions 80
Leaving the Game 85
2 Basic Game Graphics 87
Color Theory 87
How to Dress Better 88
Comparing RGB and RYB 89
The Color Wheel 90
Using Color Wisely 91
HSV and the Color Picker 92
Indexed and Direct Graphics Hardware 96
Direct Devices 96
Fixed Devices 96
Indexed Devices and Color Tables 96
Palettes and the Palette Manager 99
Using Palettes 102
Drawing with Palettes 105
Palette Animation 106
Of GWorlds and GDevices 108
A Graphic Buffer, Pixmap, and `CopyBits()` Primer 108
Why Use an Off-Screen? 109
GWorlds 110
GDevices 113
Game Graphics Programming 116
Multiple Monitor Tactics 116
Fading Strategies 117
Hiding the Menu Bar Suavely 118
Your Friend, `CopyBits()` 118

3 Advanced Graphics 121
The Theory 121
The Practice 122
Example 1: A Very Simple Scene 122
Example 2: `CopyMask()` 144
Example 3: A Clear Color Sprite Draw 150
Example 4: Encoded Sprites 159
Example 5: Dirty Rectangle 179
Example 6: An Update Rectangle List 188
Example 7: The Background 195
4 File Management 209
The File Manager and the Resource Manager 209
The Two Forks 210
A Quick Look into the File Manager 211
Resources and Handles 211
Loading and Creating Resources 213
Using Resources and Handles 214
Storing Pregenerated Data 215
Saving Settings 220
Switching between Several Resource Files 223
Saving High Score Lists 226
Saving the Player's "Save" Files 232
Save Files 232
Internally Recorded Game State 233
Passwords 233
Dungeon 3 233
Dungeon 3's Save File Management Routines 235
Apple Events in Dungeon 3 238
Other Changes in the Dungeon Source 241
Using Data Formats 243
Putting Together Your Resource File with ResEdit 243
Navigating in ResEdit 244
'STR#' Resources 246
Adding String Lists 246
'STR ' Resources 247
'TEXT'/'styl' Resources 247
Balloon Help Resources 248
Resources for the OS and the Finder 250
Menu Resources 253

Dialogs and Alerts 253
Sound Resources 254
Graphics Formats 255
Adding Graphics Resources 257
Choosing a Graphics Format 259
Adding and Editing Other Resources 263
5 User Interaction 265
Mouse and Keyboard Input 265
Keyboard Control 266
Reading the Keyboard 267
Mouse Control 269
Reading the Mouse 269
User Interface 277
Events 278
Designing the User Interface 290
Can I Take Over the Machine Completely to Win Performance? 293
Menus? We Don't Need No Stinkin' Menus! 294
Centering the Window 297
Hiding the Desktop 298
Hiding the Menu Bar 299
Dialogs 301
More Reading 318
6 Audio 327
The Sound Manager 328
Features of Sound Manager 3.0 328
Sound Manager 101 329
Data Types: 8-Bit, 16-Bit, Mono, and Stereo 330
Multichannel Sound 331
Channel Setup and Disposal 331
Making Noise and Silence 335
Advanced Sound Manager 342
Sample Rates: 11,025/22,050 vs. 11,127/22,254 342
Compression and Expansion with MACE 343
Callbacks 344
Volume Control 346
Panning 347
Rate Playback Control 348
Setting Up a Callback and a VBL Task 349
File Formats 354
The Future of the Sound Manager 358

Hollywood Sound API 358
Setup and Cleanup 358
Memory Management 359
Playing Sound 360
Hollywood Callbacks 362
Determining Features 363
Getting Information and Changing Sounds That Are Playing 363
Other Mundane but Important Functions 365
VBL Tasks 365
MACE Compression 365
Creating and Obtaining Information about 'snd ' Resources 366
QuickTime 367
QuickTime Movies 368
QuickTime Music Architecture 368
Creating Content 369
Importing Content into a QuickTime Movie 370
Embedding and Playing Music Tracks 372
Requesting Instruments and Playing Notes 377
QuickTime Software Synthesizer Instruments 383
Example Code Using the Sound Manager, QuickTime, and Hollywood 384
Debugging Audio 385
Where to Go from Here 386
7 Game Mechanics: Environments and Entities 389
A Game's "Virtual Reality" 389
World Representation 390
Unstructured World Representations 390
Structured World Representations 391
Distance in a Game World 398
Game Entities 410
Data Structures for Game Entities 411
Game Entities and Sprites 416
SpriteEngine 417
Position, Speed, and Acceleration 419
8 Game Mechanics: Behavior 433
Game Entity Behavior 433
Action Game Entities 434
Board Game Entities 445

Collision Detection 466
Collision Detection between Sprites 466
Collision Detection in a Grid 469
Collision Detection in 3-D 470
Image Space Collision Detection 470
Collision Handling 470
Separating Overlapping Objects 471
Realistic Bounces 474
9 QuickDraw 3D 491
Getting Started 493
Objects 494
Attributes 494
Initializing QuickDraw 3D 495
Creating a Draw Environment 495
Displaying a 3-D Object 512
Making a Box 512
Assigning attributes 514
Defining a Box Object 514
Referencing and Dereferencing 515
Drawing Objects 516
The Draw Loop 516
Tweaking Parameters 518
Stereoscopic 3-D 519
All about Meshes 522
Building the Mesh 522
Applying Transforms to an Object 526
Rotating the Mesh Object 527
Transformation Scope 530
Other Transform Objects 530
Texture Maps 531
(u,v) Coordinates 532
Texture Shader Objects 533
Mapping a Polygon 537
10 Networking 547
AppleTalk 548
AppleTalk Networking Concepts 549
AppleTalk Name Binding Protocol Services 555
Sending and Receiving Network Messages 565
Serial Communication 585

Outland 600
Open Transport 608
Why Apple Made Open Transport 608
What Is XTI? 609
The Features of Open Transport 610
The Difficulties of Open Transport 612
IP Multicast 616
Other Sources of Information 618
11 Porting 627
Approaching the Conversion 627
Game Play 628
Graphics 628
Sound 629
Interface 629
Technical Prowess 632
Conversions vs. Ports 632
Porting a Title 633
Converting a Title 637
Converting a Game Idea 638
Technical Considerations 639
Tools of the Trade 639
Knowledge Is Where You Find It 641
Converting the Code 643
Converting the Artwork 655
Converting the Data 656
Converting the Music and Sound 657
QuickTime and Video 659
Memory Management 661
File Management 662
Graphic Glitches 663
CD-ROM Considerations 664
Disc Layout 664
Compressing Data 665
Shipping the Title 666
Fixing the Bugs 666
Polishing the Application 668
Selected Problems and Solutions 669
CPU Issues 670
Graphics Issues 677
Sound Issues 698
Control Issues 700

12 Game Aesthetics 709
Game Metaphor 709
Milestones in Game Metaphor 709
Discovering a Metaphor 711
Artwork Aesthetics 712
Interface Aesthetics 714
Cleaning Up Your Menus 715
Mac-Like or Not Mac-Like 717
Hiding the Menu Bar 718
The File and Edit Menus 722
Dialog Boxes 722
Sound Aesthetics 725
Packaging 727
Sexism, Racism, and Violence 728
13 Optimizing Game Code for PowerPC 731
PowerPC 601 and 603 Features 731
PowerPC 604 Features 732
Optimizing the Blitter 732
The Old Way and the New Way 732
Let's Make It Faster 733
Using Indexes 737
Alignment Issues 741
Optimizing with Floating Point Math 742
Optimization Potpourri 744
`static` Declarations 744
Avoid Local Structs and Arrays 744
Limit Function Parameters 745
Consistent Data Sizes 746
14 Glypha III: A Case Study 749
Top Down 749
Initialization 750
Idle Mode 751
Play Mode 751
Scaled Coordinates 752
Player Modes 753
Rendering 754
Speed Governing 756
Sample Glypha Code 757

A Top Speed—or Not? 797
Getting Top Speed 797
Programming Language 797
Inner Loops 798
Fixed Point Arithmetics 800
Tables, Tables, Tables! 800
Avoiding Function Call and Toolbox Trap Overhead 802
Calling Nonnative Calls on Power Macs 806
Moving Data in Big Chunks 806
Putting in a Speed Limit 807
Setting a Speed Limit with `TickCount()` 808
Scheduling Events after the Current Time 808
B Macintosh Environments 811
The M68000 Microprocessors 811
MC68000 811
MC68020 812
MC68030 813
MC68040 813
MC68LC040 814
MC68881 and MC68882 Floating Point Math Coprocessors 815
PowerPC Microprocessors 815
PowerPC 601 816
PowerPC 603 816
PowerPC 603e 817
PowerPC 604 817
PowerPC 620 817
Macintosh Computers 818
Compact Macintosh Models 818
Macintosh II Series 822
Macintosh Quadra and Centris Computers 825
Macintosh LC Computers 829
Macintosh PowerBook (and Portable) Series 831
Macintosh Performa Series 839
Power Macintosh Computers 839
System 6 Compatibility 841
Making Your Program Not Crash under System 6 841
Index 843

Introduction

"If we'd had this book,
we would have shipped this game a month ago."

This book brings you the essential skills you need to program high-quality games on the Macintosh. The techniques we are presenting have been tried and tested in the commercial game market, and many have been kept as trade secrets until now. The authors include a dozen of the top experts in the Mac entertainment world (and Yours Truly, an enthusiastic intermediate) and we all have one thing in common: We sure wish this book had been available a long time ago.

We will lead you step-by-step through the intricacies of Macintosh game programming; we will provide you with tools and samples, source code and examples. The CD-ROM includes all the code in the text, and a lot more—we don't think you bought this book because you want typing practice, we think you want easy access to practical programs, and we fully expect you to cut and paste our routines into your own products.

The CD-ROM is filled with games and game demos, so you can see how your own work compares to the state-of-the-art.

The CD-ROM also includes a limited version of Metrowerks CodeWarrior C/C++, which is fully functional in every respect but these: You cannot create new projects or add new files. You can use it to compile and run any of our demos, and Metrowerks hopes you will be so impressed that you'll order the full version of CodeWarrior. When you buy the full version, you also get free tech support, which is a service Metrowerks can't provide for the "CodeWarrior Lite" version we've included here.

What You'll Need

- A working knowledge of your particular Mac.
- A basic grasp of C programming.
- A Mac II or later, running System 7.x, with a color monitor and, yes, a CD-ROM drive.

What You'll Read

Chapter 0, "A Game Programming Primer," introduces the basic concepts of game programming: animation, sound, collision detection, and error handling. If you're a pretty old hand at this, you'll whiz right through chapter 0, saying, "I hope the whole book isn't like this!" Don't worry, it isn't. On the other hand, if you're saying, "Chapter 0? That doesn't make any sense, lists are supposed to start with 1," then friend, this chapter is for you.

Chapter 1, "Your First Game," introduces a simple grid-based dungeon-digging game, with player rewards, player mortality, trolls in pursuit, and random map generation. The chapter includes an overview of the common game types, including comments on how to create them, and leads you through the challenge of choosing which Mac models to support.

In **chapter 2, "Basic Game Graphics,"** you'll find color theory, color look-up tables (CLUTs and palettes), graphic buffers, GWorlds, and the benefits of `CopyBits()`. Along the way, you'll find out how to do full-screen fades and palette animation, how to hide the menu bar, and how to deal with multiple monitors—that's a lot of information from a chapter with "Basic" in its title.

Chapter 3, "Advanced Graphics," takes us from `EraseRect()` and `CopyMask()` through eight steps of speed augmentation, including "dirty rectangles," drawing with clear colors, encoded sprites, and rectangle update lists. Meanwhile, the author shows how to use the Metrowerks profiler to show you where your program is spending (and sometimes wasting) its time. By the time you finish this chapter, you'll have a whole lot happening on your screen.

Chapter 4, "File Management," covers storage of game data, fixed data such as sound and graphics, and modifiable data such as high score lists and saved games. This is not a topic that gives people the warm fuzzies, and nobody ever finished a game and said, "Wow! Great file management!"—but they'll sure notice if you get it wrong. This chapter shows how to get it right and how to use ResEdit to assemble your resource files.

Games are inherently interactive. **Chapter 5, "User Interaction,"** shows how to get input from the player via mouse and keyboard, with all the features a modern Mac game player expects, such as user-configurable keys and mouse motions interpreted as force rather than position. The chapter also covers user interface design, event processing, menus, and dialogs.

Chapter 6, "Audio," says it all—and your game can too! The authors have provided Hollywood API, which is a mini-SoundMusicSys for readers of this book. Have you ever used SoundMusicSys? If you've played commercial games recently, then you have—SoundMusicSys is the most popular sound system for the glossy box

brigade. Hollywood lets your games play AIFF files, 'snd ' resources, mono, stereo, MACE-compressed, streaming sound, 8-bit sound or 16-bit sound, and it will do them all at once. The demo, called Hollywood API Tester, jumps through all those hoops while letting you play notes with QuickTime Music Architecture or a sound-only QuickTime movie in the background, with user-selected MIDI instruments (while I was editing this chapter, I played "Rhapsody in Blue" in the background, with glockenspiel and bird tweets as my instruments).

Chapter 7, "Game Mechanics: Environments and Entities," describes the creation of a realistic (or surrealistic) environment for your game—the laws of physics in the universe of your game, such as gravity, distance, and acceleration. Do your game's battle tanks travel faster on roads than they do in swamps? They should. Do your game's spaceships accelerate at different rates depending whether they move diagonally or horizontally? They shouldn't. Here's how to make your game entities follow the rules.

Chapter 8, "Game Mechanics: Behavior," describes the independent activities of the entities themselves. Do your game's billiard balls bounce off each other in a realistic manner? They'd better, or you'd better not call them billiard balls. Do your trolls have the smarts that God gave a turnip? Do they react with indignation to the swipes of the game player's sword, or do they stand there like salad fixings? Your game can simulate artificial intelligence with pursuit and evasion routines, random acts, patterns, and preferences. Your mobile game entities can interact and collide, bond or bounce, and your computer opponent can be a worthy challenger in card, board, and strategy games. Here are the techniques.

Chapter 9, "QuickDraw 3D," is the first look at Apple's remarkable new 3-D technology. For Power Mac only, QD 3D allows you to create three-dimensional worlds without writing your own 3-D graphics engine. It is powerful, precise, and surprisingly comprehensible. We are fortunate to have found a guru at such an early date, what with the ink not yet dry on the QD 3D boxes and the solder still warm on the accelerator cards. Expect this technology, and the Power Mac, to revolutionize the game industry.

In **chapter 10, "Networking,"** you'll learn how to get multiple game players into the action. One of the many advantages we Mac users have over our "Intel Inside" colleagues is built-in networking capability. This chapter presents the techniques you need to add networking capabilities to your game or devise a strictly network multiplayer game, using everything from Apple Talk to Open Transport. As befits a chapter about networking, two experts wrote this chapter together.

Chapter 11, "Porting," has two authors too, and is the longest chapter of the book. Why do we give porting so much attention? Two reasons: because crossplatform development is the wave of the future (indeed, it's the wave of the present—many publishers whose primary commitment is to the IBM-compatible market have

wall-to-wall Macs in the art and authoring departments, and you didn't think anyone actually programmed Nintendo games on Nintendo machines, did you?) and because we have plenty we can learn from the mass market people. The chapter is broken into General Guidelines and Specific Solutions, so it won't be too much to absorb at once.

Chapter 12, "Game Aesthetics," is an overview of many aspects of game development that are not strictly programming, but that a programmer needs to understand. The experience your game offers to the player is defined more by the concept and the art than by the actual programming, and if you want to be more than the programming cog in a game development machine, you need an understanding of these issues. What sets the mood of your game? Are you losing market share by turning off members of demographic groups other than your own—such as a different gender, race, or age? Can you communicate clearly with your art department? If you are your own art department, can you clearly envision the metaphor that draws players into your game?

Chapter 13, "Optimizing Game Code for PowerPC," shows how to optimize your code to run native on Power Macs, with particular attention to optimizing the blitter—the code that draws sprites and dumps video buffers to the screen. You may have cried, "Hot puppies!" when you found you could double your 68K program's execution speed by porting it to the Power Mac. Chapter 13 shows you how to double its speed again, through parallel execution, floating point math, `doubles` in place of longs, avoidance of local structs and arrays, and catering to other quirks peculiar to the PPC 60x processors. Imagine what you'll say then?

Chapter 14, "Glypha III: A Case Study," supplements the extensively commented Glypha III source code provided on the CD-ROM. This is the code to the complete game, and every feature is laid out for your inspection. Glypha III is a Joust-type game by an award-winning game developer, provided as freeware as his gift to the Mac gaming community, with comments and explanatory text special to this book and CD-ROM. The text offers valuable insights to the development process of an actual game, and the code covers every programming process from initialization to shutdown. With this information and your own imagination, you can create your own commercial-quality arcade game.

The **appendixes** cover speed enhancement techniques and an in-depth look at the various Mac models and their specs.

One more question might be passing through your mind....

Who Are These Guys?

Where did Hayden find a dozen gurus and one determined amateur? What makes these authors qualified to write this book?

Let's do the introductions in alphabetical order.

Jon Benton ["Basic Game Graphics"] works for Presage, writing in-house tools for lip-synching and compositing. His most recent game project was Mario's Game Gallery, published by MacPlay. He got tired of slogging COBOL about a decade ago, and has been slammin' on the Mac ever since.

john calhoun ["Game Aesthetics" and "Glypha III: A Case Study"] is something of a legend in the Mac game community. He does complete game development by himself: the concept, the code, the art, even the music. His popular shareware games, including early versions of Glider and Glypha, attracted the attention of Casady & Greene, which publishes his recent works. Glider 4.0 put john in the *MacWorld* Game Hall of Fame with the Best Arcade Game of 1991.

Stuart Cheshire ["Networking" coauthor] is the author of Bolo, a highly popular network action/strategy game. He earned his Computer Science M.A. at Cambridge, and is currently a Computer Science Ph.D. student at Stanford. You can reach him at `cheshire@cs.stanford.edu` (Web Page: `http://ResComp.Stanford.EDU/~cheshire/bio.html`).

Bill Dugan ["Macintosh Environments" appendix] is the "Macintosh High Priest" at MacPlay. Bill joined Interplay Productions in 1986 as a scripter and map designer, and was promoted to producer in 1991. MacPlay, the Macintosh division of Interplay Productions, was introduced in January 1993. Since then, Bill has overseen the development of MacPlay's entertainment titles—over 20 so far.

Mick Foley ["Advanced Graphics"] has been with Presage for a year. Before that, he worked at Maxis on the SimCity series, and was lead programmer on the SimCity Graphics Sets and Widget Workshop. He got into the Mac game programming profession out of a simple desire to work on something fun. You can contact Mick at `mick@emf.net` (Web Page: `http://www.emf.net/~mick`).

Brian Greenstone ["QuickDraw 3D" and "Optimizing Game Code for PowerPC"] has done Bloodsuckers, Firefall Arcade, Power Pete, and Gerbils for the Mac, Harley's Humongous Adventure, Claymates, and Lester the Unlikely for Super Nintendo, and is currently working on Steel Harbinger for Sony Playstation, a game using QuickDraw 3D. His employment history includes Pangea Software, Visual Concepts, and Mindscape. You'll find him at `BrianG19` on AOL, `Pangea` on eWorld, `PANGEASOFTWR` on AppleLink, and `bgreenstone@mindscape.com` from everywhere else.

Douglas Grounds ["Porting" coauthor] is president of Lion Entertainment, which converts and produces Mac games for a variety of publishers. He is Producer/Programmer for the Macintosh versions of Super Wing Commander and Doom II, and did the Mac conversions of Might and Magic I & II and Mario Teaches Typing CD-ROM. He has been a Mac programmer since 1987, and is currently working on conversion of Wing Commander III for the Macintosh and Power Macintosh.

Steve Hales ["Audio" coauthor] has been a contractor for most of the major Mac game publishers, and licenses SoundMusicSys, a sound driver written by Steve and Jim Nitchals. At last count, there were over 30 games using SoundMusicSys—literally too many games to mention, but they include Prince of Persia, Flying Nightmares, Out of this World, Falcon MC, Lemmings, Spectre VR, the SimCity series, and the Carmen Sandiego series. There are more, but you get the point. The best ways to find him are `hales@netcom.com` and `SteveHales@aol.com`.

Eric Johnston ["Porting" coauthor] works for LucasArts and has sole programming responsibility for the *MacWorld* Game Hall of Fame-winning versions of Pipe Dream, Indiana Jones and the Last Crusade, Loom, and The Secret of Monkey Island, among other Mac conversion projects. He is currently technical lead on an entirely new real-time 3-D console project for LucasArts. He chose his career primarily because, in games, "looks cool" is justification enough to proceed with a project. In his copious free time, Eric is mastering the art of the flying trapeze.

Jamie McCornack ["A Game Programming Primer"] is the Intermediates' Advocate for this book. He was chosen as Project Editor for his lack of guru status—to keep the authors from losing us mortals by assuming "everybody knows" what took them years to learn—and if Jamie can understand this, so can you. He is coauthor of Macintosh GameWriter, a beginners' game development suite. In *his* copious free time, Jamie represents the United States in international ultralight aircraft competitions, and writes about himself in the third person. You'll find him (me, actually) at `MacGameDev` on AOL.

Jim Nitchals ["Audio" coauthor] shares credits with Steve Hales for SoundMusicSys. Jim owned Cavalier Computer, has worked for Electronic Arts, and is now with Apple Computer, working on improvements to QuickTime's software music synthesizer. He has a good reputation for optimizing 68K code (he optimized the QuickTime VR display software, for example) and his audio programming skills are generally accepted as the reason QuickTime can run MIDI. For personal queries, email him at `jimn8@netcom.com` (for technical assistance, Jim suggests you use a `comp.sys.mac.programmer` newsgroup or Apple Developer Technical Support).

Ingemar Ragnemalm ["Your First Game," "File Management," "User Interaction," "Game Mechanics"...] has developed and released ten shareware/freeware games, plus Sprite Animation Toolkit. SAT has been used for a number of freeware/shareware games, including Asterax and Fracas, which you'll find on the CD-ROM. He is also one of the more frequent suppliers of game-related sample code on the net, especially through the `alt.sources.mac` group. He earned his Ph.D. at Linkoping University (the best university in Sweden on computer-related science and technology) in 1993, on optimizing image-processing algorithms. You'll find his email address on his various demos and games on the CD-ROM.

Roy Wood ["Networking" coauthor] wrote NetRisk, an AppleTalk-based conquer-the-world game that was very loosely based on the board game Risk, and Conquest, another conquer-the-world game that runs under the Outland networking system. In the daylight, he's a high school mathematics and computer science teacher. Roy is drawn to networked games because it makes it possible to play against other humans, and he hopes his chapter encourages more people to write more multiple-player games. Send him email at `rrwood@io.org`.

And of course there's our skilled and dedicated support group at Hayden Books—Stacy Kaplan, Brad Miser, and Steve Mulder—and Paul Celestin, who put our CD-ROM together (you can reach him at Celestin Company, at `info@celestin.com`)....

Speaking of Internet address, if you're on the Web you'll want to make sure to drop by the Web site dedicated to this book. There, you'll find plenty of links to great programming resources on the net. It's at `http://www.mcp.com/hayden/mac_game-gurus`.

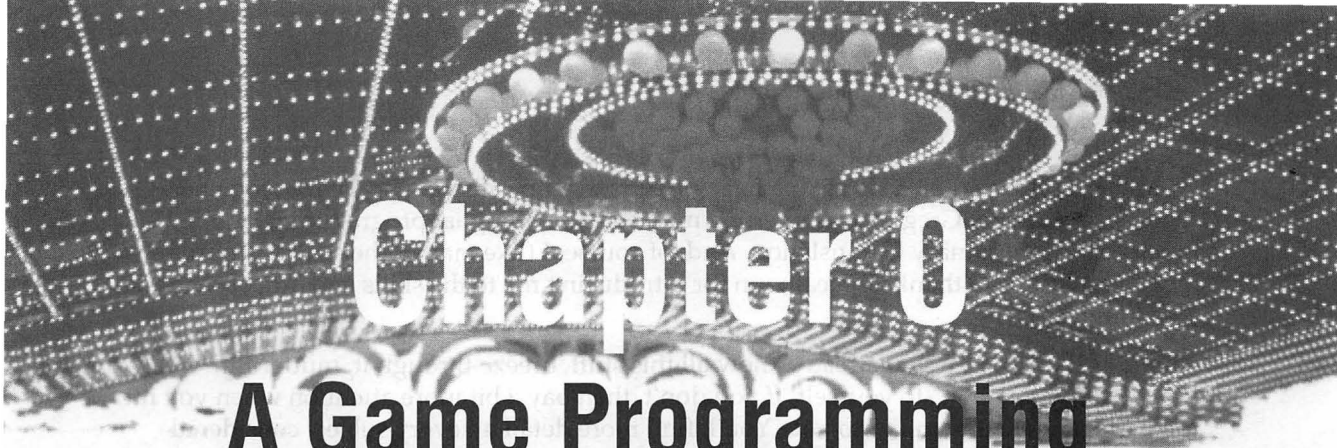
But enough about us. You have lots of pages of this book ahead of you and it's time you got started. By the time you're done, you'll be ready for your own Mac game project. Be sure to send a copy of your finished game to `macgamedev@aol.com`—we are already looking for sample games for the next book in this series.

Have fun!

Jamie McCornack

Project Editor

June 1995



Chapter 0

A Game Programming Primer

So maybe you took a look at the back cover and thought, "Wow, this is for Accomplished programmers. I'm not quite there yet." Or maybe you're a pretty hot programmer, but all you've written so far are relational databases because your Comp Sci thesis advisor thinks games are for sissies. Or maybe it's time for you to bite the bullet and switch yourself from Pascal to C, since that's the language all the tough guys use nowadays.

Or maybe you'd like a quick refresher on the basics, so you won't miss something important when you read the *real* chapters, written by the *real* masters of Macintosh game programming. This chapter covers what the cutting-edge people expect you to already know.

Here you will find rudimentary...

- Animation techniques
- Sound management
- Collision detection
- Error handling

...all in one little package. You'll find these rudiments scattered through the code and discussion, since in games, they are all supposed to happen at the same time.

In the tradition of programming primers everywhere, our first program is called HelloWorld. Unlike the traditional HelloWorld, it does not print the words, "Hello, world!" to the screen. Instead, this version shows an ambulatory clam running from left to right across a beach, stopping to say, "Hello, world!" when the mouse button is clicked, and being trampled by the next oncoming clam. Yeah, I thought I'd take an example from real life.

Mind you, this is not a game. Some of this program's features (user input, for example) are so rudimentary they don't deserve mention, and some features (score-keeping, menus, and Apple Events, for example) are nonexistent. You'll find those subjects, and many others, covered in the body of this book. This is merely a warm-up for animation and sound—two aspects of the Mac that Serious Programming Tutorials tend to miss. I wouldn't even have included error handling, except I hate to see demos crash.

This chapter is based on a series of magazine articles john calhoun and I wrote together. I would like to thank Greg Dow for getting me started in bilingual programming and for much of the preliminary C translation. And, of course, I (like many other game programmers) would like to thank john calhoun for introducing me to the skills and arts of Mac game programming.

And now, we're off. If you already know all this stuff, breeze through it, muttering, "Yeah, yeah, I know, I know," to yourself. If you don't, then pay a bit more attention when you hit the parts that are unfamiliar to you. You'll find more detail on every subject considered herein once you get into the chapters that are the heart of this book.

Animation Basics

Don't be offended by the "rudimentary" qualifier in the intro—this was rocket science a few years ago. For illustration, we'll be doing things the old-fashioned way, with off-screen bitmaps and pixmaps instead of GWorlds (I know GWorlds are easier; I'm teaching you this way for the same reason your Driver's Ed teacher trained you on a stick shift even though your dad's Dodge Viper had an automatic transmission), and we'll be using the `CopyMask()` routine to select which portions of our sprites we want to copy. There's no optimization for speed, no encoded sprites, no sophistication—you'll find all that in chapters 2 and 3 and appendix A—and still, this technique is more than fast enough for 90 percent of the most successful Mac games to date.

What's Animation?

Here's the basic principle of animation: If you put a still picture in one place, then quickly remove that picture and put a similar still picture in another place nearby, the object in the picture will appear to have moved from the first place to the second. In a manner not unlike Macromedia Director, your brain will fill in the blanks as needed to perceive smooth flow of motion.

So what's the problem? Why is animation on the Mac such a hassle? Why do one's first attempts produce fuzzy flickering images that strobe in and out, or look solid on one part of the screen and ghostly on others, or jerk from place to place?

It all centers on the word "quickly."

How Quick Is Quick Enough?

There are two conditions of quickness that must be addressed. First, how quick is the entire operation?

Twelve fps (frames per second) is about the slowest you can go and still get the illusion of continuous movement. Even 15 fps looks a bit jerky to most people (early movies ran at 18 fps, and they didn't call 'em "flickers" for nothin'), but 20 fps is more than satisfactory for

game purposes (modern movies run at 24 fps), and 30 fps (the speed of modern television) is usually indistinguishable from the real thing. Fortunately, as a game programmer, you're not in competition with the real thing—Wile E. Coyote chases the Road Runner at 12 fps, and nobody complains, "That's not a real coyote, his frame rate is too slow!"

But at slower than 12 frames per second, the illusion breaks down. Sure, your brain will simulate smooth motion from point A to point B and from point B to point C, but you'll be aware that the motion stopped briefly at point B—hence the image jerks.

Fortunately, it's only a game, and your game is not in competition with real life. In fact, if you want people aware that your game is Just As Good As Those Games In The Glossy Boxes, you may *want* it to be a bit jerky. The commercial games you see in arcades have an awful lot of pixels and tend to value elaborate graphics more than smooth graphics, and in order to have enough time for all the bells and whistles to be displayed and rung and tweeted, refresh rates are commonly lower than ten frames per second.

And there's the first part of our answer: When you're animating, don't ask your Mac to do more thinking between frames than it can do in a 12th of a second or so, unless jerky graphics will somehow add to the charm of your game.

Mind you, there are some awfully successful games that run slower than that. Flight simulators come to mind. They're jerky and the images they show are relatively crude. That's the price they pay for all that number-crunching between frames, but that's the current state of that particular art.

Instead, we're going to manipulate bitmaps and pixmaps, which the Mac does with alacrity. We're going to copy, not draw, and we'll use the word "copy" to mean "move a piece of existing artwork from one bitmap (or pixmap) to another."

Note

So, what's a bitmap*? A bitmap is a collection of bits organized into rows and columns for use in graphic display. On the Mac, a 1 value signifies a black pixel, and a 0 value signifies a white pixel. A pixmap is a bit more complex. Since a pixmap stores color information, 1 bit per pixel isn't enough. When folks speak of "8-bit color," for example, they're referring to pixmaps in which each pixel is represented by 8 bits. One bit is enough to distinguish between two colors (1 bit can be 1 or 0, black or white). Eight bits is enough information to distinguish between 256 colors (00000000, 00000001, 00000010,... 11111110, 11111111).*

We won't try to "draw" in the MacDraw sense of the word, which is to juggle a bunch of numerical information, calculate lines and areas, add patterns, and so on. You'll do the artwork long before anyone runs your program, since the Mac can hurl artwork out of memory gigagobs faster than it can dream up artwork on its own. We're going to use sprites to represent our game entities, and...

Sprites? We used to call the little-pictures-of-things-that-move-on-the-screen "screen objects," or sometimes just "objects," but nowadays, with OOP (object-oriented programming)

using "object" to mean something quite different, we call 'em sprites. The term "sprite" is a bit of a misnomer—sprite implies a hardware assist—but it has drifted into the vernacular, so we'll use it. To keep us all on track, let's define a few terms:

- A *sprite* is a piece of bit-mapped (or pix-mapped) art that can be superimposed over background art.
- A *sprite set* is a collection of sprites showing a single item in a variety of activities and/or from a variety of angles: running, jumping, turning, or whatever.
- A *face* is a specific member of a sprite set, usually given a descriptive name to help the programmer stay organized.
- A *frame* is one full loop through the animation cycle, with the screen redrawn where needed. It is rare to redraw the entire screen each frame (unless you're doing a 3-D game). In the interest of speed, one generally redraws only those portions of the screen where sprites were in motion. In its copious free time between frames, the program looks for user input, calculates the sprite positions for the next frame, and makes noises.

And now, back to the code....

How Smooth Is Smooth Enough?

The other condition of quickness is more complex, but also more important. After you've erased the image at point A, how quickly can you copy a new image at point B? Not quickly enough, sad to say.

Trouble is, the Mac can copy just about as fast as it can erase, so if you do your animation on the screen (copy at point A, erase at point A, copy at point B, erase at point B, copy at point C...), you'll find nothing on the screen about half the time. Below 15 frames per second, you'll see the object appear and disappear. At higher frame rates, it will look translucent, indistinct, ethereal, vague.

Can't you do your calculating while the image is on the screen, or maybe delay for a Tick or two and then erase the old image and *copyrealfast*?

Note

A "Tick" is 1/60th of a second and is the basic unit of Mac speed control. When a Mac is turned on, a variable `tickCount` (of type `longInt` in Pascal, long in C) begins incrementing. In around two years and three months, `tickCount` will overflow, so if you leave your Mac running that long, it will probably crash.

Sorry. It won't work. There's a finite refresh time to deal with: the time it takes the monitor to redraw the screen. If the screen refreshes 60 times a second (as the Macs with built-in monitors do) and your object remains undrawn for only one percent of the time, then every couple of seconds it will be blank when the time comes for screen refreshment, and it will

stay blank for 1/60th of a second. That's not long enough to register in your mind as a complete disappearance. Instead, the image will seem to have partially disappeared (about 1/4th transparency) very briefly (about 1/15th of a second). It's quite disconcerting.

Even more disconcerting is when the calculate-and-copy rate gets close to the screen refresh rate. If your program redraws at, for example, 61 frames a second, it will go in and out of sync with the screen refreshment once per second, and your image will fade in and out once per second. There's got to be a better way.

There is. Do all your calculating and copying in a `GrafPort`, which does not appear on the screen. When you're done, cover the old image with the new image. Thus, you'll never have a moment when the old image is erased from the screen but the new image hasn't yet appeared. Voilà! Flicker-free animation—no ghosts, no strobing. Anything less isn't good enough if your work is going to be compared with the current state of the art. And it will.

One more thing before we get into the code...

How Fast Is Too Fast?

Yes, there is such a thing as too fast. Monitors work by sweeping a beam of electrons across a matrix of phosphors. The phosphors glow when struck by the electrons. Switching the electron beam rapidly on and off, causing some phosphors to glow and some not, displays a picture. The phosphors fade fairly quickly, so the beam has to come back and refresh them (light them up again) fairly often. How often this beam comes by is called the refresh rate.

Mac monitors refresh (redraw the screen) 60 to 70-plus times per second, depending on the model, but you can't see it because the sensors in your eyeballs can't refresh that fast. However, as your game graphics redraw approaches the monitor refresh rate, images can get fuzzy and ghosty because the new image has appeared before the previous image could fade from your eyeball/brain interface.

If your game graphics redraw is within a few fps of the monitor refresh rate, there may be noticeable jerks as the redraw goes in and out of sync with the refresh. If your game runs at 59 fps on a monitor that refreshes 60 times a second, then every second 58 of your frames will be displayed once and one frame will be displayed twice. That frame may be noticeably brighter than the others or cause a jerk once per second in otherwise smooth movement.

If your game graphics redraw exceeds the monitor refresh rate, some of your frames will not be displayed at all. If they are critical frames, their loss will be sorely felt. For example, a shoot-'em-up game may have targets (asteroids or something) that explode and disappear when they are hit. If the explosion frame never makes it to the screen, the target will just... disappear. The players will feel like the programmer owes them an explosion, and they won't be impressed by a developer who says, "Hey, this is the fastest game on the market; this game runs so fast that half of the action never even makes it to the screen!"

This is not hypothetical. HelloWorld will run at over 200 fps if you take the brakes off it. Full color. No PowerPC chip required. And no tricks, either. No assembly language, no drawing

direct to the screen, no Apple-unapproved cleverness such as direct manipulation of memory. Just standard Mac Toolbox calls and standard Mac resources. If you have the imagination, you can write the next Glider, Troubled Souls, or Tetris without one whit of fancier graphics programming. Great games can be made without violating Apple's programming guidelines.

The HelloWorld Code

You'll find it on the CD-ROM, in the 00 - Game Programming Primer folder within the Book Chapters folder. As is my habit, I'm not filling the pages of this book with code, because you wouldn't care to type it anyway. Besides, there's a lot of code in all the iterations of HelloWorld, and the publisher would have to slaughter an inappropriate number of trees to put it all down in black and white. The code is thoroughly commented in the source code files, and if you have a smattering of C programming experience, I don't think you'll find it confusing.

So if any of this text seems fuzzy to you, open up the appropriate source code file and read the code on your monitor as you read the book in your hand.

HelloWorld1: Bare Bones

In this version, a black-and-white clam runs across the screen. When the mouse is clicked, its lips move, its voice speaks, "Hello, world!" and it's gone.

Now then: This is the 90s—why am I bothering you with black-and-white?

In shareware game programming, the rule of thumb has been if the author owns a black-and-white Mac, the game runs in black-and-white, and if the programmer owns a color machine, it runs in color and crashes on a Classic. In the commercial game world, it is common to provide a black-and-white version along with the color version, or at least an error message saying, "This game requires color," followed by a graceful exit.

I believe that any game should strive for "commercial quality," at least as far as a willingness to run on a variety of Macs and an unwillingness to crash mysteriously. Thus, we start at the bottom line, black-and-white 1-bit graphics, introduced over a decade ago and compatible with everything bearing the Macintosh logo.

For those of you who think the black-and-white Mac customer base is dwindling, well, you're wrong. The percentage may be going down compared with color Macs, but the quantity is still climbing. People aren't sending their Mac Pluses to the landfill yet, and every time your local dealer sells another noncolor PowerBook, the customer base toggles up. And when an SE/30 got replaced at the office by a Power Mac, it ended up at home in the family room, where it is likely to get more game use than it got when it was working for a living.

I can justify the use of black-and-white in this demo code despite its ring of antiquity:

- Black-and-white graphics take less code than color graphics, and I want to start off simple.
- We don't have to check for monitor depth, so I can put off error checking until HelloWorld2.
- Black-and-white illustrations are easy to print with black ink on white paper.

There are some good arguments for black-and-white graphics later on, too:

- When appropriate, your game should have a black-and-white version when it hits the streets, even if the color version is the primary draw. And if you're going to do two versions, which one should you do first? That's right: the easy one.
- One-bit PICT files (black-and-white) take up roughly 1/8th the RAM and disk space of equivalent 8-bit color PICT files. Thus, if RAM or disk availability is a factor—perhaps you want this game to run on a low-end 4/80 PowerBook while a spreadsheet updates itself in the background—you may want to write "tiny" games.

There are three source files in the HelloWorld1.π project. The first is MGWSound.c. We're going to ignore this for a little while—trust me for a bit; it makes sounds happen, and I'll get back to it.

The next is MGWGraphicsBWLite.c, which contains three routines:

```
void CreateOffscreenBitMapLite (Rect *theRect, GrafPtr *offScreen)
```

There is no error checking yet, but if all goes well, `CreateOffscreenBitMapLite()` assigns a piece of your Mac's memory to a `GrafPort` named `offScreen`, with a `GrafPtr` and `.portBits` as passed by the parameters. In a nutshell, a `GrafPort` is a structure containing all the fields QuickDraw needs to work with black-and-white graphic data, and it is called, directly or indirectly, by virtually every QuickDraw function. `CreateOffscreenBitMap()` assigns a bitmap to the field `.portBits`, and the size of the field is determined by `theRect`. If there is not enough memory available for 1 bit per pixel in the rect `theRect`, this program will crash. By the time we get to HelloWorld2, that won't happen anymore.

```
void KillOffscreenBitMap (GrafPort *wasPort)
```

No error checking is needed here. If the `GrafPort` we want killed is already dead, it stays dead, but doesn't get any deader. That's important, because killing an already-dead pointer or handle will cause a bus error. If it never existed, it stays nonexistent. So what's to check?

Presuming `KillOffscreenBitMap()` is passed a "live" `GrafPort`, that `GrafPort` and the contents of all its fields are released from memory, thus giving the memory manager more room to work with. One of those fields is `.portBits`, where the no-longer-desired bitmap is stored.

Once your game is done with a particular bitmap, you can call `KillOffscreenBitMap()` and get rid of it. The typical arcade-style game disposes of the bitmaps only when the program quits, but `KillOffscreenBitMap()` may be needed if you have a multilevel game. If you don't

get rid of the bitmaps you used in Level 1, you may not have enough memory for Level 2, and so on.

```
void LoadPictureLite (short thePictID)
```

This is a stripped-down version of `LoadPicture()`, which we will be seeing soon. It loads the 'PICT' resource selected by `thePictID` into the `.portBits` field of the current open `GrafPort`.

Now on to HelloWorld1.c, which contains the main program.

The HelloWorld1 Constants

My "regular" constants start with lowercase `k`, followed by an uppercase letter, so I can remember they're konstants, and they have descriptive names, so I can remember what they're good for.

By the way, I learned my spelling standards in the field. You'll find the other authors in this book use pretty much the same standards (in some cases, because I learned mine from them). If you do too, you can read our code, and someday, we can read yours.

- `kWaitTicks`—The number of 60ths of a second the program waits between one frame and the next. This is used to control the speed of the program and to make the program run at similar speeds on dissimilar Macintoshes.
- `kFrontFace` through `kRunRightFace`—The faces the sprite can show, enumerated for the program's convenience and named for the programmer's convenience. I'm using Ingemar Ragnemalm's convention here, from chapter 7, and using "face" to describe how a sprite looks—its size, shape, and appearance. The enumeration begins with 0, which is handy for future arrays.
- `kMaxFaces`—The next integer after the last face. This is how many elements in an array that holds one element for each face.

I like having the resource constants all up front with the other constants, and I like starting their names with a lowercase `r`. Real games have a lot of resources, and following this standard helps keep me from getting confused.

- `rFacesID`—The 'PICT' ID number where the sprite face images are found. I am fond of the lots-of-faces-in-one-big-'PICT' technique, but you'll find other techniques, such as filing faces in icon lists, throughout this book.
- `rMasksID`—The 'PICT' ID number where the masks are found. At this stage, we use masks to select which pixels of each face will be transferred to the screen.
- `rBackgroundID`—The 'PICT' ID number where the background picture is stored.
- `rMainWindowID`—The 'WIND' ID number of the main, and only, window. This is where we'll see the action.
- `rHelloWorldSndID`—The 'snd ' ID number for the voice of the clam.

Let's take a look at the 'PICT' resources before we go any further.

Here we see The Many Faces of Clem the Clam in a wide variety of activities (see figure 0.1). By our selecting which parts of this 'PICT' are transferred to the screen, the clam will appear to do different things from time to time.



Figure 0.1 *'PICT' 129—the sprite faces.*

The specific faces are selected using a small rectangle (32 x 32 pixels, in this case) that encloses one and only one face. If the contents of that rect are transferred to the main window via `CopyBits()`, we'll have problems with the white space. For example, let's select the first face in the 'PICT', which shows Clem standing still, turned toward the left side of the screen (see figure 0.2).



Figure 0.2 *A single face of Clem the Clam, as defined by a rect (0, 0, 32, 32).*

If we `CopyBits()` this rect onto the background in the `srcCopy` transfer mode, all the white will be transferred, obliterating a 32 x 32 section of the background (see figure 0.3).

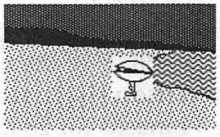


Figure 0.3 *Clem copied in `srcCopy` transfer mode.*

If we `CopyBits()` this rect onto the background in the `srcOr` transfer mode, none of the white will be transferred and the background will show through the white parts of Clem's body, giving him a ghostly, ethereal look (see figure 0.4).

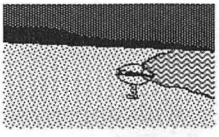


Figure 0.4 *Clem copied in `srcOr` transfer mode.*

So we'll use `CopyBits()`'s slow-paced sibling, `CopyMask()`, which, while not the fastest way to move a sprite, is by far the simplest. `CopyMask()` has three bitmaps (or pixmaps) as parameters: the source, the mask, and the destination. It has three rect parameters too, which

determine where on the source bitmap and mask bitmap the data come from and where on the destination bitmap the data go. The main reason `CopyMask()` is relatively slow (relative to the techniques you'll learn in chapters 2 and 3) is that it is extremely versatile—`CopyMask()`, like `CopyBits()`, can work with varying palettes on different pixmaps and can scale to a destination rect of different size and shape than the source rect, but these processes take time.

And here we have the masks (see figure 0.5).



Figure 0.5 *The masks for Clem the Clam and Clamity Jane, too.*

`CopyMask()` is like `CopyBits()` with a special transfer mode. `CopyMask()` transfers pixels from the source to the destination only where the corresponding pixels are value 1 (black) in the mask (see figure 0.6).

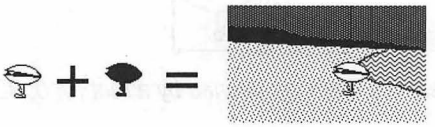


Figure 0.6 *The source, controlled by the mask, is copied to the destination.*

Okay, back to the code. We have a new type, `tFaceStruct`, a structure of one rect for face and another rect for mask. These two rect variables, when properly initialized, define how a particular face appears on the screen. If the 'PICT' of the masks were the same size and layout as the 'PICT' for the faces, we could use a single rect definition for both components. But the masks do double duty. We have 14 masks for 28 faces. Why more faces than masks? Note that the top and bottom rows of clam faces are identical in silhouette, though they have "internal" differences, such as eyes (clams have eyes?) open or closed. In this case, each mask is suitable for two faces. It could be more—a rear view of a clam would use the same mask as a front view, for example.

When we get to HelloWorld3 and have two clams of different colors, those 14 masks will serve 56 faces—28 for each clam.

The HelloWorld1 Variables

My variables start with a lowercase letter, with uppercase letters at word breaks—for example, `thisIsAVariable`. I clearly describe their purpose in the name. I used to abrvt 2 cnsrv kystrks, but found that six months down the road, I was losing more time deciphering `thsVrbl` than I'd gained from typing fast. Global variables are an exception; I start my globals with a lowercase `g` and carry on from there as usual.

We have pointers (`GrafPtr`) to `GrafPorts` named `workPort`, `facesPort`, `masksPort`, and `backgroundPort`. The `GrafPtr` `mainWindow` is where we will send the finished graphics for display on the monitor.

`facesRect` and `masksRect` define the shape of `facesPort.portBits` and `masksPort.portBits`. `bigPictureRect` is the shape of `mainWindow`, which is predefined as a 'WIND' resource. We use `bigPictureRect` to define the shape of `workPort.portBits` and `backgroundPort.portBits`. `CopyBits()` will cheerfully scale data to fit various shapes of bitmaps, but it takes a *long* time in doing so. By our using a single rect to describe these bitmaps, there is no risk of an accidental slowdown from scaling. `clamIsAtRect` describes where on the `workPort` and `mainWindow` the clam is displayed, `clamWasAtRect` describes where on the `workPort` and `mainWindow` the previous clam face needs to be erased, and `clamComboRect` describes both those rects combined, so we can erase and redraw to the screen in a single step.

Opening the HelloWorld1 source code and examining the code as you read this text is a good idea. Actually compiling and running HelloWorld1 is... well, I wouldn't say it's a bad idea to run it, but in the interest of simplicity, I've stripped out every iota of error checking, so if any little thing goes wrong, it will crash. And when it crashes, you'll lose any unsaved work on any applications running in the background. I should be ashamed of myself for showing you how to do off-screen graphics without checking for adequate memory and valid resources first. Oh well, one step at a time.

`sprite` is an array of `tSpriteType`, and that's where we'll store the locations of the seven faces and masks we'll be using in our animation.

`thisSprite` is a short variable; we'll use it to select which element of `sprite` to copy.

The HelloWorld1 Routines

My routine names start with an uppercase letter, and (surprise!) the names say what they do.

```
void SetTheRects()
```

In `SetTheRects()`, we initialize all the rect variables, including those in the `.face` and `.mask` fields of the elements of the `sprite` array. Okay, it's not too tedious in this sample, but it sure can be in a complex game with multiple levels and many views of many objects. GliderPro and Spaceway 2000 have pages and pages of `SetRect()` stuff.

How do you determine the borders of the various rects so you can set them properly? I'm personally fond of the Show Coordinates window of Studio1 (a fave amongst black-and-white paint programs; published by Electronic Arts) and Zeus (mostly glorious 32-bit color paint program, though slightly buggy; by Delta Tao), but others have similar features with similar names, such as Show Mouse or Mouse Coordinates.

Incidentally, you don't need a black-and-white art program to make black-and-white art. If you use only the black-and-white colors from a 4-bit (16-color) or 8-bit (256-color) palette and save your art in 1-bit (black-and-white) format, black-and-white is what you'll get.

After you have completed your artwork and packed your various screen objects together snugly onto one cozy picture, use your paint program's marquee tool to move the picture to the upper left corner of a paint document. That puts the top left point of the picture at (0, 0), and by using Show Coordinates, you can prowl around your picture using the marquee tool to highlight rectangles around the objects you want to locate. Jot down the point data at the top left and bottom right corners, and you have what you need to define the rect.

Most paint programs show coordinates in some fashion, and a variety of desk accessories show mouse position, but some are particularly slick in that they show the size of the rectangle as well as the coordinates of the cursor. Since `CopyBits()` works most efficiently when the horizontal dimension of the rect it's copying is divisible by eight, and since `CopyBits()` is called a minimum of four times per frame, this is useful information.

If you're going to do it the hard way with some sort of Show Mouse tool, expect it to return the coordinate of the point to the upper left of the mouse position. Note the point given by Show Mouse when the cursor is on the upper left pixel of the desired rectangle, then note the point given by Show Mouse when the cursor is on the lower right pixel of the desired rectangle, and then add one to the lower figure and the right figure, so you define the rect that *includes* the lower right pixel.

Another technique is to arrange your artwork neatly and calculate the positions of the rects. All the clam views in 'PICT' 129 are 32 x 32 rectangles, with no overlaps or spaces between them, so the first rect is (0, 0, 32, 32), the next is (32, 0, 64, 32), and so forth. You can also use 'DITL' user items in ResEdit, as Ingemar shows in chapter 4, "File Management."

```
void SetThePorts()
```

This routine creates the four off-screen ports. `facesPort` is loaded with the clam faces 'PICT', `masksPort` is loaded with the clam masks 'PICT', and `backgroundPort` is loaded with the background 'PICT'. These three `GrafPorts` are our sources of bitmap data—it takes too long to keep calling 'PICT' resources every time we need them, so we store their data in the `.portBits` fields of these three `GrafPorts`. `workPort` is where we'll assemble the data thus stored.

Then we create and show `mainWindow`. Then we load `mainWindow` with the contents of `backgroundPort.portBits` (so the user has something to look at) via `CopyBits()`.

Because this is the first use of `CopyBits()`, let's go through its operation one parameter at a time.

The first parameter `&((GrafPtr)backgroundPort)->portBits` is the source of the data to be copied.

The second parameter `&((GrafPtr)mainWindow)->portBits` is the destination bitmap.

The third parameter `&bigPictureRect` defines what portion of the source bitmap will be copied, and the fourth parameter `&bigPictureRect` defines where in the destination bitmap it will be copied to. These don't always have to be the same rect, as they are here, or have the same coordinates, but they do have to be the same size and shape. Otherwise, `CopyBits()` will squish or stretch the contents of the first rect to fit in the second rect. In general, such distortions are aesthetically undesirable, and they greatly slow down `CopyBits()`.

The fifth parameter `srcCopy` sets the copy mode. `srcCopy` takes everything in the source rect, whether white pixels or black, and copies it to the destination rect, erasing all previous contents of the destination rect.

The sixth parameter is the clipping region—anything falling outside this region will not be copied. We don't want anything clipped from this copy operation, so we set that to the `.visRgn` of the destination `GrafPort`.

Using the same technique, we copy the contents of `backgroundMap` into `&((GrafPtr)workPort)->portBits` so we'll have an unsullied background to work with.

```
void InitAll()
```

This is rather brief at the moment: `ToolBoxInit()`, and `InitializeForSound()`, a function in MGWSound.c (and which I'll explain later). `thisSprite` is set to `kFrontFace`.

```
void RunRight()
```

The value of `thisSprite` cycles from `kStepRightFace` to `kWalkRightFace` to `kRunRightFace` and back to `kStepRightFace` again, using the scissors-paper-stone algorithm, and `clamIsAtRect` is moved from left to right.

```
void LipSynch()
```

From `kFrontFace` to `kEehFace`, back to `kFrontFace`, then to `kEehFace` and `kOohFace`, etcetera, etcetera, `thisSprite` runs through a pattern of facial expressions. Meanwhile, `PlayASound()` plays `rHelloWorldSnd` in the background.

```
void ShowClam()
```

And now, at long last, we're actually going to animate, though in a grossly simplified manner.

There are four steps to this process. We'll presume the loop is already looping and animation is in progress, so let's call the current state Step 0, and work our way through the loop (see figure 0.7).

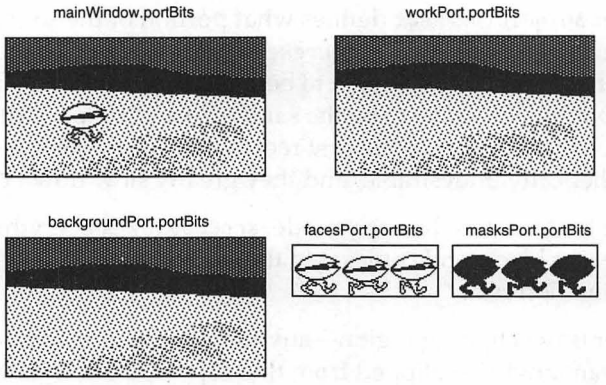


Figure 0.7 *Step 0: This is the state of* `mainWindow` *and* `workPort` *after the previous pass through the animation loop.*
`backgroundPort`, `facesPort`, *and* `masksPort` *are used as sources only, and are not modified during animation.*

Now we're going to leave `mainWindow` alone for a while and work off-screen (see figure 0.8). `workPort` is never visible to the player, "off-screen" is a computer analogy to "offstage" in live theater. The audience sees smooth action on stage; they don't see the pandemonium in the dressing room.

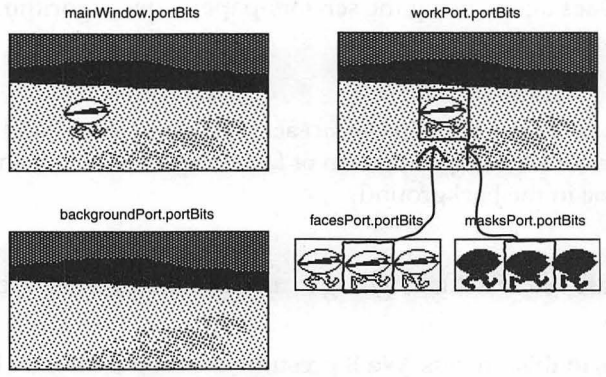


Figure 0.8 *Step 1: Using* `CopyMask()`, *copy the next face of the clam onto* `workPort.portBits`, *at the clam's next position.*
Now `workPort.portBits` *looks like we'll want* `mainWindow` *to look when we're done with this loop. For now,* `mainWindow` *still shows the old view of the clam at the old position.*

Now `workPort` is just the way we want the next frame to appear on the screen. We could copy the entire contents of `workPort.portBits` onto `mainWindow.portBits`, but that's killing flies with a sledgehammer. Copying more than we need to copy slows the operation. All we need to copy is the area in `mainWindow.portBits` where the last view of the clam was (that's `clamWasAtRect`), plus the area where the new view of the clam is destined to appear (that's `clamIsAtRect`). There is a Toolbox call that is perfect for the job—`UnionRect()` (see figure 0.9).

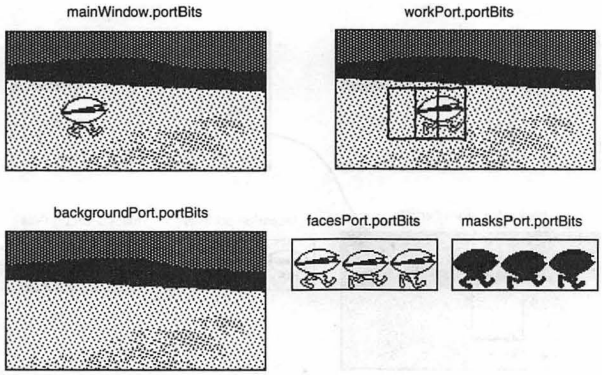


Figure 0.9 *Step 2: Calling `UnionRect()`, we set `clamComboRect` to the union of `clamWasAtRect` and `clamIsAtRect`; that is, `comxboRect` is the smallest rectangle that includes both `clamWasAtRect` and `clamIsAtRect`.*
(Note: Nothing actually happens in `workPort`. This drawing is just a reminder of `clamWasAtRect`'s and `clamIsAtRect`'s location in `workPort.portBits`.)

Now it's time to do the part the audience sees. The relevant portion of `workPort.portBits` is copied and pasted onto `mainWindow.portBits`. Wow, look at that! The clam moved (see figure 0.10)!

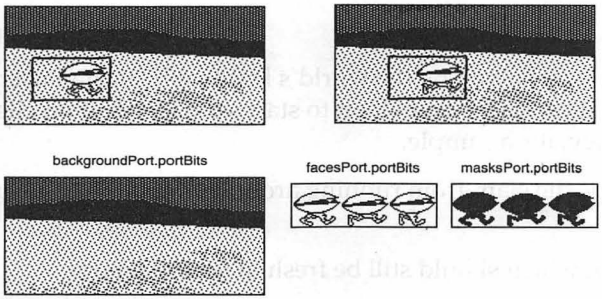


Figure 0.10 *Step 3: Using `CopyBits()` in `srcCopy` mode, we copy the entire contents of `clamComboRect` from `workPort.portBits` to `mainWindow.portBits`.*
This erases the old clam and draws the new clam in one action. No behind-the-scenes bitshuffling to distract the game player, and no flickering, strobing, or ghosts.

We'll need a clam-free workspace with a clean background we can copy onto for the next time we run through this loop. We could copy the entire contents of `backgroundPort` onto `workPort`, but again, that's killing flies with a sledgehammer. `workPort` is almost perfect already. It has the background we copied onto it in the last line of `SetThePorts()`. The only fly (clam?) in the ointment is that in Step 1, we copied a clam onto `workPort.portBits` at `clamIsAtRect`. So, using `CopyBits()` in `srcCopy` mode, we copy the background at `clamIsAtRect` onto `workPort.portBits` at `clamIsAtRect` (see figure 0.11).

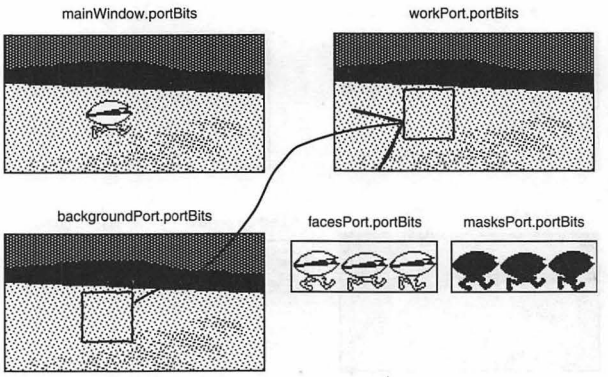


Figure 0.11 *Step 4: In* `workPort.portBits`*, the previous view of the clam is replaced with data from* `backgroundPort.portBits`*, using* `CopyBits()` *in* `srcCopy` *mode to copy the bits enclosed in* `clamIsAtRect`*.*
Now `workPort.portBits` *looks just like* `backgoundPort.portBits`*, and* `mainWindow` *still shows the clam.*

We've done quite a few things here, but the only operation that shows on the screen is Step 3, so there's no flicker. All the real gruntwork was done off-screen, and if we'd shown the game player what was happening in `workMap`, the scene would have flickered and strobed like crazy.

```
void Main()
```

The first call is `InitAll()`, followed by the world's least sophisticated user interface, three "while button" calls that cause the main loop to start up with a button click and quit with another button click. Hey, it's a sample.

The `DoDelay()` call keeps the clam from running around so fast the game player can't see what it's doing.

Then comes `ShowClam()`, which should still be fresh in your memory.

And when the mouse is clicked again, we call `LipSynch()`. Then we call `CloseDownSound()`, and it's over.

Now, even if you intend to create black-and-white games exclusively, you need to read the next section. It is primarily about color, but it is also about how to move your sprites from place to place on the screen, and it's about error checking too. Do not use `MGWOffscreenBWLite()` for any of your own programs. Even if you are writing a quick little ditty for your toddler's SE, you can save the kid some frustration by using `MGWOffscreenBW()`, which includes the error-checking routines you're about to meet.

HelloWorld2: Color and Complexity

Now we go from bitmaps to pixmaps, and from single sprites to multiple sprites, and from blind faith to error checking. This program builds on what we covered in HelloWorld1, and I

refuse to type it all over again. If you didn't read the HelloWorld1 section because you don't intend to do black-and-white games, go back and read it now. I'll wait.

Index Color vs. Direct Color

Bitmaps are single bit arrays, each bit representing a white (0) or black (1) point on the map. Each point on a pixel map (pixmap) is represented by a number of bits; that number is called its "depth."

Four-bit (16-color) and 8-bit (256-color) graphics are index color. A pixmap holds a number for each pixel, and the number refers to a position on a color look-up table (a 'CLUT', in resources) or palette ('pltt').

Sixteen-bit (thousands of colors) and 32-bit (millions of colors) graphics are direct color. The number in memory is divided among the three display colors (red, green, and blue on an RGB monitor), 5 bits per each display color for 16-bit color and 8 bits each for 32-bit color. Thus, 16-bit color provides 32 levels of intensity (2^5) for the red, green, and blue colors and 32-bit color (which is 24-bit color with 8 bits left available for an alpha channel) provides 256 levels of intensity (2^8) each. In layperson's terms (what the user sees in the Monitors control panel), 16-bit is called "thousands of colors" and 24-/32-bit is called "millions of colors."

If you have the tools for it, you'll want to do much of your art development in 16- or 32-bit color, but when you play the game, we recommend you use index color. It's not just me; the authors of the graphics chapters recommend index color too.

Why? Well, we know technology is changing rapidly, but as of this writing, here are the relevant facts:

- Eight-bit color graphics take half the disk and RAM space of 16-bit graphics, a quarter the space of 32-bit graphics. The result is more game in less storage space and a game that will run on more Macs.
- Eight-bit graphics transfer between pixmaps significantly faster than 16- or 32-bit graphics, because only half or a quarter as much data is being transferred. The result is a faster game or a game with larger and more complex sprites.

Maybe down the road, RAM and storage will be so cheap that size won't matter, and CPUs will be so fast that speed won't matter, but today, they matter.

Four-bit color runs even faster, requires even less storage space, and will run on even more Macs than 8-bit color, but it will always look like a cartoon. Sixteen colors aren't enough for photorealism.

Another point to consider: You can't show more colors than are available with the user's monitor and VRAM, and the quick and dirty way Color QuickDraw deals with scaling direct color down to index color is... well, actually, it's amazingly clean considering how quick it is, but it's not as good as you can do yourself with an imaging program. QuickDraw does it one pixel at a time, matching each direct color with the closest available index color, with no regard to neighboring pixels. If you use an imaging program or a 24-bit paint program, you can scale down to 8- or 4-bit art with dithering to preserve color gradients. So if your game is going to run on 8-bit or 4-bit Macs, you're not just wasting storage space when you provide 16-bit art, you're compromising picture quality, too.

Note

QuickDraw does have a dithering mode, which we won't get into here, because it is sloooow. It's much better to do your art dithering once in advance than to have QuickDraw do it over and over and over as the game is being played.

Okay, time to look at some code.

MGWUtilities.c

We have a new source file in the HelloWorld2.π project, because now that we're dealing with pixmaps and multiple bits per pixel, memory errors are more likely. Most of these routines will be useful for more than graphics; `RedAlert()` is a good example.

One task for MGWUtilities.c is to cover up compatibility problems in a graceful manner. Hey, if everything ran on 128K Macs, Apple would still be making them, right? The user's hardware and system may have to meet some minimums before your game will run. By your noting and announcing the problem instead of crashing without explanation, the user receives the subtle message of "user is too cheap to get a proper Macintosh," rather than the more blatant message of "programmer has made a poor career choice." MGWUtilities.c endeavors to deal with compatibility problems in a graceful manner, checking for hardware and System minimums and saying "Goodbye" politely if these minimums are not met.

MGWUtilities.c introduces a new variable type, `tMacEnvironment`.

```
  struct  tMacEnvironment
{
    short   wasDepth;
    short   isDepth;
    Boolean isColor ;
    Boolean hasColorQD;
    Boolean canSwitch;
    Boolean isSystem7;
}
```

`thisMac` is the only variable of type `tMacEnvironment`. `thisMac` contains information your program may need for compatibility checks. If your game switches to a more favorable color depth, use `wasDepth` to store what the depth was before your game started, so it can reset back to the old value when it exits.

```
void InitEnvironment()
```

`InitEnvironment()` sets the fields of `thisMac` to the results of the following functions.

```
Boolean MacHasColor()
```

This sets `thisMac.hasColorQD`. It's a quick call to `SysEnvirons()` to see if we have Color QuickDraw. If not, the user needs to know the color version of our game won't run on this

machine. If `thisMac.hasColorQD` is TRUE, `InitEnvironment()` calls `InitBlackAndWhiteColors()`, which sets the RGB components of `rgbBlack` as low as they can go (0) and the RGB components of `rgbWhite` as high as they can go (65535). `CGrafPorts` will not display a full range of colors and shades without this information.

```
Boolean MacIsAtLeastSystem7()
```

This sets `thisMac.isSystem7`. If our code makes any calls to routines or capabilities introduced in System 7, we need to bow out now if `MacIsAtLeastSystem7` returns FALSE. Incidentally, Apple encourages us to use the latest System and says there is no disadvantage to requiring it from your users. But there are still a lot of Mac IIs and the like running System 6.0x. So if your game is capable of running on some versions of System 6, you may want to demand less from your minimum System, such as...

```
Boolean MacCanSwitchDepths()
```

This code sets `thisMac.canSwitch`. That is, is it running System 6.05 or later? You'll need this information if you want to use a `SetMonitorColorDepth()` routine that resets the monitor if needed, without exiting to the Finder and requiring the user to open the Monitors control panel.

```
short   GetThisMacsDepth()
```

This sets `thisMac.isDepth`. This tells the program what depth the main Mac monitor (the one with the menu bar, if there are multiple monitors) is currently set to. This routine does not tell if this monitor is color or grayscale, since the user probably already knows that, and because index color runs fine on grayscale monitors.

`InitEnvironment()` sets the foreground and background colors to fully white and fully black, and also verifies that the current practical minimum hardware and software, a Macintosh II and System 7, is available to run a black-and-white game with low-hassle sound. Actually, you can go as low as a Mac Plus with System 6.02, but it is a fair amount more work. There has to be a limit somewhere, and many would argue that even going back to the Mac II is pretty extreme. One of our hip and knowledgeable tech editors estimates no more than 5 percent of new software goes onto System 6.x machines, and heaven knows that programming with System 7.x is more than 5 percent easier than attempting compatibility with System 6.x. If you want to ignore everything but color Macs on System 7.x, you're not missing much, and as for pre-Plus Macs running pre-System 6.02, most of them have been converted to fish tanks by now.

If it is important to you to support early Macs with early Systems, get in touch with me; I still have some antique code hanging around somewhere. I'll feel a bit like Chuck Wade being asked the best cam grind for a Studebaker Hawk, but it's available if you want it.

If your game does get into pre-Mac II hardware running pre-System 7, `InitEnvironment()` will call `RedAlert()`.

```
void RedAlert(short whatGives)
```

Have your program call `RedAlert()` or a similar routine of your own devising if any of the fields of `thisMac` indicate that your game demands more from the user than is possible to perform (e.g., 256 colors on a monochrome monitor). Believe me, this will not be the first time this user will have seen such a message. An explanation, followed by an exit to the Finder, is much friendlier than a crash, which is what happens if your program attempts to send the wrong-depth pixmap to your monitor. If there is a black-and-white version of this game, this is the place to mention it.

In honor of Apple's Human Interface Guidelines, `RedAlert()` calls `CenterAlert()` to place the alert window in the upper center of the main monitor. Since this is the last the user will see of your programming skills, leave a good impression (subliminal though it may be) by getting your exit right.

The parameter `whatGives` tells `RedAlert()` which item on the appropriate string list ('STR#' resource) to display in the alert box. A half dozen indexes are included.

```
#define kErrOldSystem     1
#define kErrOldMacintosh  2
#define kErrLowMemory     3
#define kErrNoResource    4
#define kErrNoColor       5
#define kErrNot8BitColor  6
```

The strings cover a fair range of fatal errors:

1. "This program requires System 7 or later to run."
2. "This program requires a Macintosh II or later to run."
3. "There is not enough memory available to run this program. Try closing other programs or setting the monitor to fewer colors."
4. "Resource not found, or damaged, or there is not enough memory to load the resource."
5. "The monitor must be set to color or grayscale for this program to run. A black-and-white version of this program is available."
6. "The monitor must be set to 256 colors for this program to run."

When your program discovers a difficulty that will otherwise lead to a crash, it calls `RedAlert()`, passes the index number of the appropriate error message, displays a single button dialog (the button says "Finder") with the error message emblazoned therein, and, when the button is selected, exits to the Finder.

You can add error messages by adding strings to the 'STR#' resource list—Ingemar shows you how in chapter 4, "File Management." You can also use ResEdit to modify the strings I've listed here, because not all of these are likely to apply to your program.

Indeed, as written, `kErrLowMemory` and `kErrNot8BitColor` would not both apply to any one program. While reducing the monitor's color depth will reduce memory requirements, it

would seem a trifle snide when paired with a demand for 8-bit color, and I would advise against flip-flopping between these two error messages as the game player sets the Monitors control panel from 8-bit to 4-bit to 8-bit to 4-bit.

There's more to MGWUtilities.c, but we won't need it for a while.

MGWGraphics.c

This is the real thing, with color and error checking.

These routines handle the off-screen pixmap creation, loading, and dissolution, and are quite similar to the bitmap routines in MGWGraphicsBWLite.c. Each off-screen pixmap is a field of a specific `CGrafPort` (not a `GrafPort`), and we'll be using at least four of them.

```
Boolean CreateOffScreenPixMap (Rect *theRect, CGrafPtr *offScreen)
```

This is much like `CreateOffscreenBitMap()`, except for checking `thisMac.isDepth`. If the monitor the game is playing on is set to a different depth than the pixmaps, the application will fail.

Color depth is a direct factor in memory needs. This routine checks for adequate memory, and if there is not a sufficiency, exits instead of crashing. This can be a problem if you're using many large pixmaps and deep color; over 300K of RAM is required for a 640 x 480 (Apple 13" monitor) pixmap 8 bits deep. If `CreateOffscreenPixMap()` fails, it calls `RedAlert(kErrLowMemory)`, and `RedAlert()` informs the user.

By the way, over a meg of RAM is required for a single full-screen pixmap in 32-bit color. No wonder Photoshop 3.0 insists on lots of memory. Don't forget, you're writing a game, not an upscale prepress graphics program. Keep a grip on your memory needs.

```
void KillOffscreenPixMap (CGrafPort *wasPort)
```

This is just like `KillOffscreenBitMap()`. If the `CGrafPort` we want killed is already dead, it stays dead, but doesn't get any deader, thus avoiding a bus error.

Don't let obsolete pixmaps hang around in memory. When you're done with your splash screen, get rid of it. People won't buy new 72-pin SIMMs in order to play your game, so be stingy with memory.

Let's run a sample program to see how color differs from black-and-white.

HelloWorld2.π: Color and Error Checking

The constants and variables are much like before, except for many `GrafPorts` turning into `CGrafPorts` (since we're working with color this time), a new constant `kColorBitsNeeded` (which equals 8 in this example), and a couple of new 'PICT' ID numbers.

Open Hello2.rsrc with ResEdit and take a look at the 'PICT's. Orange trousers and brown shoes on a green clam? I guess clams don't read *CQ* or *Clamsmopolitan*. Check out that background, done with KTP Bryce. It took me about half an hour to set up and it rendered

while I was cooking dinner; the Bryce file is in the ClamArt folder on the CD-ROM. Between the rendered background and the colorized clam, you wouldn't guess the same guy did the art, would you? Color makes a big difference.

Okay, close the resource and let's look at the routines.

```
void SetTheRects()
void RunRight()
void LipSynch()
```

You've seen these before, haven't you?

```
void SetTheCPorts()
```

Much as I eschew obfuscatory abbreviation, Apple has inserted the letter `C` for color into so many Toolbox calls that we might as well go with the flow. The difference between `SetTheCPorts()` and the previous program's `SetThePorts()` is small but important: We're working with `CGrafPorts` and `CGrafPtrs` now.

The 'PICT' resource ID numbers for the clam and background have been changed to 136 and 134, which is where I stored the 8-bit art for this program.

Note that the resource and bitmap for the masks remain the same as in HelloWorld1. Masks for `CopyMask()` are always bitmaps.

```
void InitAll()
```

This is the same as before, but with a check for `thisMac.isColor == kColorBitsNeeded` to ensure that the monitor's depth matches our art. If the monitor is set to a lower color depth than we demand, then the art will look crummy, and if the monitor is set to a higher color depth than we demand, the program will use more memory than it needs and may process the graphics more slowly than we like. So here, if the color depth does not suit our needs, we tell the player what to do about it and then exit to the Finder.

I did it this way mostly so you could see how `RedAlert()` works. If you want to see `RedAlert()` in action, set your main monitor to 16 Colors (4-bit) or Thousands (16-bit) or Millions (32-bit) of Colors and run HelloWorld2.

```
void ShowCClam()
```

This routine uses the same Toolbox calls as `ShowClam()` from HelloWorld1. Color QuickDraw doesn't have (or need) a `CopyCBits()` or `CopyPix()`; the basic copy routine kept the `CopyBits()` name.

And now...

```
void Main()
```

...which is word-for-word identical to the HelloWorld1 main program. When you run HelloWorld2, you will find no surprises. Unless, of course, your monitor is not switched to 8-bit (256 colors), in which case HelloWorld2 will chastise you and quit.

By the way, if you don't have 256 colors available, you can change this program into HelloWorld4Bit by changing the constant `kColorBitsNeeded` from 8 to 4. That's all you have to do. You won't get the prettiest picture, but the 8-bit art will load into 4-bit pixmaps without crashing. Don't let the art department see what you've done, because the automatic color conversion leaves much to be desired by the sensitive eye. Oh, go ahead, try it; pretty gross, eh? If for some reason you like that effect, add `kErrNot4BitColor` to your lists of constants and strings.

See, the switch to color wasn't so rough. Now for something more challenging.

HelloWorld3: Clams in Formation

We will be using MGWUtilities.c, MGWGraphics.c, and MGWSound.c as before. Open HelloWorld3.c and let's see what's new.

There's some new declarations. `kClamStepLength` is gone, as are some others. Now we have `kJaneStepLength` and `kClemStepLength`. There are numerous distinguishing features 'twixt Jane the Clam and Clem the Clam, the most obvious being that Clem has his own resource ID number for sprites (`rClemSprites` = 135).

Jane and Clem each have their own `janeIsAtRect` and `clemIsAtRect` (formerly `clamIsAtRect`), and `...WasAtRect` and `...ComboRect`, but they share masks. Jane and Clem are different colors (Clem's no fashion plate either) but the same shape. All previous `clamVariables` have been made into `janeVariables` and `clemVariables`, and there's a `janeSprite` and `clemSprite` in place of `thisSprite`.

The Routines

```
void SetTheRects;
void NewRightRunView;
void LipSynch()
void SetTheCPorts;
```

Nothing new here.

```
void InitAll;
```

`janeSprite` and `clemSprite` get set separately to `kStepRightFace` and `kRunRightFace`, so they won't run in step with each other like members of the Clam Marines. Chowder Fi!

```
void ShowClams;
```

Now we're into new territory. Note it's `ShowClams()`, plural.

If this were a typical arcade game in which no two sprites could occupy the same space on the screen, we could duplicate `ShowCClam()`, rename them `ShowJane()` and `ShowClem()`, and run them one after the other. However, we want more versatility than that. We want our sprites to be able to cross in front of each other, with the front sprite (Clem the Clam, in this case) superimposed over the rear sprite (Clamity Jane).

If we were to copy Jane to the screen and then copy Clem, any area where `janeRect` and `clemRect` overlap will flicker from Jane to Clem to Jane to Clem in a disturbing fashion. When Jane and Clem are together, we have to copy them both to screen at once.

`ShowClams()` `CopyMask()`s the image of Jane onto `workPort.portBits`, then `CopyMask()`s Clem onto `workPort.portBits`. If Clem is overlapping onto Jane, no problem; that's what we want—Clem superimposed on Jane, under control of Clem's mask. If Clem is not overlapping onto Jane, that's no problem either.

Then we do the usual `UnionRect()` to set `janeComboRect` and `clemComboRect` independently.

Now we need to find if there is a possibility of overlap. We call...

```
if SectRect(clemRect, janeRect, dummyRect)
{
    contactFlag = TRUE;
    CopyBothAtOnce();
 }
```

...which tells us whether `clemRect` and `janeRect` intersect. If they do, `contactFlag` is set to TRUE (I'll tell you about `contactFlag` in a minute) and the `CopyBothAtOnce()` routine is called. If they do not intersect...

```
{
      contactFlag = FALSE;
      CopyOneAtATime();
}
```

...`contactFlag` is set to FALSE and `CopyOneAtATime()` is called.

```
void CopyOneAtATime()
```

If the clams aren't in contact, we don't want to copy them both at once, because if they were far apart we'd be copying a great big `allComboRect`, and copying big things takes longer than copying small things. So `CopyOneAtATime()` is just the latter half of `ShowCClam()` run twice: once for Jane and once for Clem.

```
void CopyBothAtOnce()
```

If the clams are in contact, we set `allComboRect` to include `janeComboRect` and `clemComboRect`, and copy `allComboRect` from `workPort.portBits` to the screen. Then we "erase" the clam images from `workPort.portBits` by covering `allComboRect` with pixels from `backgroundPort.portBits`. Now `workPort` is ready for the next cycle.

```
void JaneLoop();
void ClemLoop();
```

These should be familiar from the HelloWorld main programs. By our breaking the main program routines down into `JaneLoop()` and `ClemLoop()` routines, the main program stays a presentable size.

Note

If you try to cram everything into the main program, you lose the main strength of Pascal and C—which is readable code. My rule of thumb is that any routine that won't fit on the screen is getting too big. And no, I don't use a full-page monitor.

```
void Main()
```

This has been streamlined by putting some of its calls in `JaneLoop()` and `ClemLoop()`, but otherwise it is unchanged from HelloWorld1 and HelloWorld2.

When you click to start, you see Clem overtake Jane (providing you leave `kClemStepLength` larger than `kJaneStepLength`) and pass on the right without flickering or fading. If you set `kClemStepLength` equal to `kJaneStepLength`, Clem and Jane pace each other. When you click again, Jane stops and says, "Hello, world," and Clem keeps running. If you time your click right, Clem runs by while Jane is speaking, just barely missing a collision. But not for long.

HelloWorld4: Clams in Collision

Most of this program is a repeat, but this routine in HelloWorld4.c is new:

```
void DoImpact()
```

This routine is called only from `JaneSpeaks()` and only if certain conditions are met. Calling from `JaneSpeaks()` assures us that impact occurs only when Jane is standing still.

Upon impact, Jane gets knocked forward (by calling `OffsetRect(janeRect, kJaneStepLength, 0)`) and Clem bounces backward (via `OffsetRect(clemRect, kClemStepLength * kBounce, 0)`). `kBounce` is a multiplier of `kClemStepLength` and must be negative for Clem to bounce backward.

Then there is a repeat loop in which `PlayLoopSound(rDizzySnd, kHighSoundPriority)` is called, `janeFaceCounter` is incremented from 0 up to `kJaneLastFace`, and Jane and Clem are posed on their keesters with thoughts spinning around their heads.

Also, there are a couple of new calls to `PlayASound()` at various priorities. To put this parameter in a nutshell, higher priorities can interrupt lower priorities. To help you get a feel for `PlayASound()` and to add even more realism to our illusion of clams running wild on the beach, I have added a few new sounds: `rFootstepSnd`, `rImpactSnd`, and `rDizzySnd`.

```
void PlayASound(short soundID, short priority)
```

This routine is found in MGWSound.c, which I'll cover in depth in a few more pages. The `soundID` parameter tells `PlayASound()` which 'snd ' resource ID number to play, and the `priority` parameter tells `PlayASound()` if it should interrupt the currently playing sound to play the requested sound.

Unlike Hollywood, the Sound API Steve and Jim provide in chapter 6, MGWSound.c uses only a single sound channel, which (almost by definition) can play only a single sound. Glypha III uses similar routines (like, guess who helped me with mine?) but with two channels instead of one. This is a primer, and given proper prioritizing, one channel is plenty.

`rFootstepSnd` is given `kLowSoundPriority`, which is one of the constants in MGWSound.c. `rFootstepSound` is played once for every step (which takes three frames) Clem takes. Sounds of the same priority can interrupt themselves, so if you increase the frame rate by changing `kWaitTicks`, the step sounds speed up, with the beginning of each new step sound interrupting the decaying sound of the previous step.

If `rFootstepSnd` is playing, it can be interrupted by `rHelloWorldSnd`, which is a `rMediumSoundPriority` sound. However, even though `PlayASound(rFootstepSnd, kLowSoundPriority)` is called every three frames, it will not interrupt the cry of "Hello, World!" Once `rHelloWorldSnd` is finished playing, `rFootstepSnd` is heard again.

`rImpactSnd` is played at `kHighestPriority`, so it can interrupt anything. Then `rDizzySnd` is played at `rHighPriority`, through a call to `PlayLoopSound()`.

```
void PlayLoopSound(short soundID, short priority)
```

In function, `PlayLoopSound()` differs from `PlayASound()` in that `PlayLoopSound()` will not interrupt a currently playing sound of equal priority. Instead, it erases everything but the current sound from the sound command queue and plays the requested sound immediately after the current sound is finished playing. In execution, it is quite different from `PlayASound()`, in that it wrestles around with `SndCommands()` and the command queue. For a full understanding of its operation, open MGWSound.c and look the code over.

```
void LipSynch
```

We have a new Boolean variable: `impactReadyFlag`. `impactReadyFlag` is initialized to FALSE by `InitAll()`. If Clem is in front of Jane when Jane turns, stops, and begins to speak, we do not want an impact, because the user will literally not have enough time to see what happened. We want Jane to make a full turn and at least begin to speak before the collision occurs. So we see whether `impactReadyFlag` is FALSE, and if so, we see if `clemRect` and `janeRect` intersect (via `SectRect()`), and if not, `setImpactReadyFlag` to TRUE. Thus, Clem and Jane can only collide if they have been apart first.

If `impactReadyFlag` is TRUE and `clemRect` and `janeRect` intersect, `LipSynch()` calls `DoImpact()`. Otherwise, it's `LipSynch()` as usual.

Ingemar explores more sophisticated techniques for collision detection in chapter 8, but this will do for now. We have other graphics issues to deal with.

HelloWorld5: Clams in Multibit

Not every color Mac has 8-bit capability, Many low-end models are limited to 4-bit (16-color). Many Macs can run 16- or 32-bit color, and their users prefer them that way. Insisting that users change for the convenience of the programmer is not the route to job security.

Instead, you can call the MGWUtilities.c routine `GetThisMacsDepth()` to find out what depth the monitor is set to. `NewOffscreenPixMap()` uses that information when creating the `CGrafPorts` and pixmaps. `SetBestPICTs()` can use this information to select the optimum art for a given monitor setup.

While any pixmap can be displayed on any depth of color or grayscale monitor (except 1-bit monitors, which show only black-and-white), best picture quality comes when art is tailored to the depth of the monitor displaying it. A 4-bit monitor will display a simplified version of 8-bit 'PICT' or even a 32-bit 'PICT', but there will be losses. If you use your own skills and a paint or imaging program to reduce the depth of a 'PICT', you will have better results. Select Save As from the File menu and choose the color depth desired. You will probably be given dithering options; try different things until you find what you're happiest with.

In the declaration for HelloWorld5.c, we have 'PICT' resource ID numbers for 1-, 4-, 8-, and 32-bit clams and backgrounds. You can see how they look at a variety of monitor depths. Also, feel free to open a copy of the resource HelloWorldBig.rsrc and look at the 'PICT's.

Program ClamsInMultiBit switches pixmap depth to match your monitor setting and loads its various pixmaps from the best 'PICT' resources for that depth. We use three new short variables—`bestJaneSpritesID`, `bestClemSpritesID`, and `bestBackgroundID`.

Note

Even though the Crash-O-Matic feature has allegedly been disabled on recent versions of ResEdit, and even though the resource file is on CD-ROM, working with copies of resource files is a good habit to get into. If you roll your eyes and say, "Man, you'd have to be crazy to open an original with ResEdit," folks will figure you're an experienced Mac programmer. And if you say, "Actually, I'm not that familiar with ResEdit, we use Resorcerer," folks will figure you're a professional Mac programmer. ResEdit is included on the CD-ROM in the back of this book, and it works fine, but even if it were perfect, I'd still do my resource work on copies. ResEdit may be stable now, but I'm not. To experience more Thrill of Victory and less Agony of Delete, work with copies of your resource files.

```
void SetBestPICTs (short pixelDepth);
```

Called by `SetTheCPorts()`, this routine takes the parameter `pixelDepth` and, case `pixelDepth` of this or that, sets `bestJaneSpritesID`, `bestClemSpritesID`, and `bestBackgroundID` to this or that best 'PICT' resource ID number.

There are a few oddities in this routine. You'll note that in the case of `pixelDepth` of 2...

Note

Wait a minute. There's no 2-bit color in Color QuickDraw. No, but there is 2-bit grayscale, and HelloWorld5 runs fine in grayscale, 2-, 4-, or 8-bit. Switch your Monitors control panel to gray and try it.

...in the case of `pixelDepth` of 2, the selected PICTs are 1-bit: black-and-white. I could have whipped up some optimized 2-bit art (even the name sounds pretty crummy), but the 1-bit art was already done and this program won't run in 1-bit mode, so if you'd like to see how a black-and-white version of this sample would look, switch your monitor characteristics to 2-bit (4 grays).

The latest version of `SetTheCPorts()` has an `if` statement to call `RedAlert()` and switch us out if the monitor is set to 1-bit. Most of the time, this is because the Mac in question is limited to black-and-white, and the user should be referred to the black-and-white version of this program (if any).

```
void Main()
```

...is unchanged from ClamsInCollision8Bit. The pixmap depth switching is transparent to the user, as well it should be.

Or should it?

HelloWorld6: 8 Bits Preferred

Open a copy of the resource HelloWorldBig.rsrc from HelloWorld5. Open its 'PICT's with Open Picker by ID, found in the Resource menu. Look at all that stuff, for one background and two sprite sets. The 'PICT' files fill over 700,000 bytes (not quite 700K, since a kilobyte is actually 1,028 bytes).

So while it's all well and good to optimize art for the current monitor depth, all those 'PICT's sure fill up the floppies.

Let's look at those 'PICT's and the bytes of disk space required for optimum art for each 'PICT' at each depth:

	1-bit	4-bit	8-bit	32-bit
background	3,280	82,190	144,498	221,394
clam1	3,362	6,772	16,894	45,670
clam2		6,854	16,932	43,853
masks	1,762	1,762	1,762	1,762
total art bytes	8,406	97,578	180,086	312,682

If we had included 2-bit and 16-bit art, it would have been even bigger. As it stands, offering 4-bit and 32-bit art along with our 8-bit art more than triples the disk space our art absorbs. That's fine for a sample on CD-ROM, but impractical for a game going out on floppy or modem. Including 'snd 's and code, an 8-bit ClamsInCollision application can fit in 200K without compression. With four sets of art, the program and resources fill well over 700K. In most cases, it's probably better to supply just the 8-bit art.

Also, deep pixmaps demand lots of RAM, and the memory required to load a big resource is greater than the memory required to load a small one. By doodling around with the zone size, we find this program uses...

- 160K RAM with 2-bit pixmaps, 1-bit 'PICT's
- 320K RAM with 4-bit pixmaps, 4-bit 'PICT's
- 580K RAM with 8-bit pixmaps, 8-bit 'PICT's
- 1,030K RAM with 16-bit pixmaps, 32-bit 'PICT's

Reducing the depth of the 'PICT' resources helps a bit (this program runs on 950K of RAM with 16-bit pixmaps and 8-bit 'PICT's), but the main memory hogs are the too-deep pixmaps.

So, if we're offering an 8-bit game (for example), we might want to encourage users to set their monitors to 256 colors. People using fewer colors are missing out on the fine points of our art, and people using more colors are wasting RAM.

However, we don't want to make color depth changes mandatory. If your game is being played on a 32-bit monitor, chances are that Mac has plenty of RAM and doesn't need to switch. If your game is being played on a 4-bit monitor, chances are that Mac is doing the best it can and would already be switched to a greater color depth if the owner could afford the VRAM. Or perhaps, due to a surplus of other open programs or a dearth of RAM, a particular Mac may not have enough memory to run your game in 8-bit color. So your game may prefer 8-bit color, but should run in 4-bit color for the people in the peanut gallery.

So I've introduced another routine in MGWUtilities.c.

```
void YellowAlert(short whatGives)
```

Big surprise! `YellowAlert()` is much like `RedAlert()`! However, the exit to the Finder is not mandatory—there is a Play button also. Call `YellowAlert()` when there is a significant but not fatal compromise in hardware, software, or setup. In HelloWorld.6, the significant choice is color depth.

`YellowAlert()`'s messages are fairly mild, such as "For best performance and least memory required, switch your monitor to 16 colors. Would you like to exit to the Finder or play with the current monitor setting?" and "256 colors are recommended for best picture and performance. Would you like to exit to the Finder or play with the current monitor setting?" You can change these messages ('STR#' resource ID 1001 in HelloWorld6.rsrc; you can open and change them with ResEdit), but don't, DO NOT insult your customer (e.g., "Only jerks have 16-color Performas. Get your mom to buy you a REAL Macintosh!"). Be nice.

HelloWorld6 uses only the 8-bit 'PICT' resources. If you'd like to make a copy of HelloWorld.rsrc and remove all the 'PICT's but ID numbers 130, 134, 135, and 136 (there's one on the CD-ROM named HelloWorld8Bit.rsrc), you can Build or Make Application (depending on your programming environment) and compare the size of the finished app with and without all those other 'PICT's. I got 212K versus 728K, which is a pretty significant difference.

The big change in `SetTheCPorts()` is a check for `thisMac.isColor == kColorBitsRequested`, a variable that replaces `kColorBitsNeeded` from earlier versions. That constant has been renamed `kColorBitsRequested` to reflect that it is no longer mandatory. If what we have is not what the programmer thinks is best, it calls `YellowAlert()`, which politely suggests a more optimum color depth setting. The call to `SetBestPICTs()` is gone, naturally, since we've taken away the choice in 'PICT' resources.

HelloWorld6.1: No Brakes

How fast does this thing go? Change `kWaitTicks` to 0 and see what happens. Your mileage may vary, but my 660AV can run two independent clams at 250 frames per second, with 8-bit color. I could have 25 clams on the screen, or any other 32 x 32-pixel sprites, each with its own pixmap and its own pathway in life, and this program would still run at 20 fps.

You can call me a lazybones, but I prefer to leave speed optimization for those programs that need it and keep things simple for those games with low demands. Tetris doesn't need to draw direct to screen. GliderPRO doesn't need a custom blitter. Myst has sold a whole lot of copies without cutting-edge programming. Don't be intimidated by the poundage of this book; it is very thorough, and you don't have to absorb it all at once.

Some Sound Advice

Sound handling is a fairly easy task nowadays because the designers of the Macintosh did such a fabulous job with the Sound Manager, which replaced the Sound Driver found in antique Macs. The sounds themselves (okay, the digital representations of the sounds) are stored in resources, in easy access to a few generic procedure and function calls.

The Sound Driver was considerably less fabulous and was extremely difficult to use. While I applaud the brilliant programmers who worked with sound on the 128 and early 512 Macs, only a masochist would do so today.

Later versions of the Sound Manager (Systems 6.0.7 through 7.x) are even more fabulous, featuring continuous play from CD-ROM and multiple sampled sounds playing at once. Unfortunately, these features demand at least a 68020 running at 16 MHz and an Apple Sound Chip, so they won't run on an SE, a Plus, or a 512E.

Even more unfortunately, when the AV Macs came out with their DSP (Digital Signal Processing) chips in place of the Apple Sound Chip, some perfectly legitimate *Inside Mac*-approved routines ran badly, such as accessing the hard drive every single time a sound played, causing many games to lurch and grind. A few clever but non-Apple-standard sound routines were left mute by the DSP, with bizarre results. Virtual Valerie, for example, was struck speechless. Though various inanimate objects in her environment continued to sound off, players wishing to interact with Ms. Valerie were reduced to reading her lips and guessing a lot.

MGWSound.c takes care of all that for you, though without the grace or style of Hollywood. Think of MGWSound.c as training wheels, and when you're comfortable with the concepts of sound management, chuck MGWSound.c and go Hollywood. You get multiple sound channels in Hollywood, for one thing, but most of the advice I'm giving here applies to whatever sound management system you use.

Sound Markers

Whether your game project is a solo or group effort, I don't recommend doing sounds first. If you're working by yourself, you'll want to tailor your sounds to the screen action, rather than vice-versa. If one of you is developing 'snd ' files and another is doing the programming, you won't want the programmer to have to wait for the finished 'snd ' files. Here's a nice alternative—use audible "sound markers" to represent the sounds you'll use in the finished product.

These sound markers are the audio equivalent to the thumbnail sketches used in graphics development. There are two ways to do this.

1. *The Generic Way*—If you predict your game will use ten sounds, load ten consecutive 'snd ' resource files with the spoken words "zero," "one," "two," "three," and so forth and so fifth up through "nine." Take a sheet of paper and write the words "zero" through "nine" down the left side of the paper. Write the 'snd ' resource ID numbers next to the words you stored them in (we suggest ID numbers 3000 and up to keep away from Apple's "reserved" IDs). Then use the rest of the row to describe the sound you'll use in the finished game.

"zero"	3000	crowd cheering
"one"	3001	fiery explosion
"two"	3002	auto horn
"three"	3003	tuba in torment

   etc.

2. *The Game-Specific Way*—Speak the sounds you predict your game will use and load them into the resource files. Take a sheet of paper and write a list like the one below.

"hooray"	3000	crowd cheering
"kablooey"	3001	fiery explosion
"honk"	3002	auto horn
"oompaarghh"	3003	tuba in torment

   etc.

The advantages of the Generic Way are that any game under development can use the same sound resources and that if you change your mind about a sound, all you have to change is

the written list. So if you decide the Ferrari shouldn't burst into flame after it plummets off the precipice, you can fix it with the stroke of a pen.

The advantages of the Game-Specific Way are that you won't have to remember (or look up) what the numbers stand for, and if you speak the sounds with feeling (oh, go ahead, this is no time to be shy), you'll find the sounds add to the mood of the game during development. The disadvantages are you'll have to dive into ResEdit to make any changes, and you'll need to create new 'snd ' resources to change them to. But if you have the hardware handy, the results are worth the extra trouble. When you're done changing your resources, update your list.

If ten sounds aren't enough, you can add new sounds as you find you need them.

Synchronous vs. Asynchronous Sound

`SndPlay()` is a remarkable Toolbox function introduced in System 6.0.2. If `SndPlay()` is passed a NIL `SoundChannel`, it will locate the 'snd ' resource that had its ID number passed in the second parameter, load that resource into memory if needed, allocate and open a `SoundChannel`, initialize the channel to the sampled sound synthesizer, play the sound synchronously, and close the sound channel. If the first parameter is NIL, it doesn't care if the last parameter is TRUE or FALSE, it'll play synchronously regardless. If you have a business program or a slow-paced strategy game, it's the only sound routine you need.

However, if you have a game that shows action on the screen, you'll find this routine is a big disappointment, because while it's locating and loading and allocating and playing, nothing else will be going on anywhere in your program. That's what synchronous sound means. All your game characters and screen objects will stop what they're doing and listen to the sound, the laws of inertia and gravity will be repealed, and time will halt until the sound is done. It puts a real damper on the flow of the game.

If the first parameter is a pointer to a valid and open sound channel, `SndPlay()` allows sounds and screen action to run independently (asynchronous), without distracting the player with unexpected disk access operations, and with a minimum of memory use, a minimum of errors, and graceful exits from problems that may arise. Of course, `SndPlay()` then considers the validity and openness of that sound channel to be Somebody Else's Problem, and will crash if you get it wrong. Like many things in the Macintosh world, it's complex for the programmer, so the finished product will be simple for the user.

MGWSound.c

With MGWSound.c, it's simple for the programmer (you) too, though the code of MGWSound.c is fairly long and involved. It does considerable checking of the user's Mac and system and selects the best way to provide sound in that particular environment—but it's all behind the scenes. The interface is simple and straightforward and all you need to work with.

The comments in the code are pretty good, so I won't go through them line by line. I'll just explain what they're good for, how to use them, and what we were thinking about when we wrote them.

The Environmental Routines

These get your program in and out of the sound environment and keep your program sounds from interfering with any other programs.

```
void InitializeForSound()
```

Call this routine when the game begins. It sets the sound channel to NIL and the sound priority to 0, checks the system environment (the system hardware and software) for a minimum of a Mac II running System 7, and updates the variable `canUseSound` accordingly.

```
void SwitchingOutSound()
```

This kills the sound channel when switching out of the game, but leaving it open in the background. If the sound channel is left live, the next program may not be able to play sounds, not even a `SysBeep()`. Since the `soundChannel` variable is set to NIL at some times during game play, it's difficult to predict the state of `soundChannel` when switching out. Call `SwitchingOutSound()` when moving to another application, or sound may or may not play.

```
void CloseDownSound()
```

This kills the sound channel when quitting or switching out to another program. Call this routine when leaving the game, or face justified wrath when the game player figures out that it's your game's fault that Excel won't beep anymore.

The Load Sound Routines

Without sound-loading routines, `PlayASound()` and `PlaySynchSound()` would read from disk each time a new sound is played. Loading the sounds into memory before they're needed makes your game run faster and smoother.

These routines return a Boolean value as an easy way to verify they worked. If all goes well, they return TRUE. What could go wrong? Running out of memory or attempting to load a corrupted resource (and now that everybody and his dog have ResEdit, resources can be quickly and conveniently corrupted) causes these to return FALSE, so you can send the user a dialog stating that something is awry. You'll probably want to shut off the sounds and release them, too, since your game isn't going to run well with an undersized heap packed chock full of 'snd ' resources.

```
Boolean LoadASound (short soundID)
```

When passed a valid 'snd ' resource ID number, `LoadASound()` loads the resource from disk to memory, moves it high in the heap, and locks it.

```
Boolean LoadARangeOfSounds (short firstID, short lastID)
```

This calls `LoadASound()` a number of times in succession, loading resources from the `firstID` to the `lastID`. One of the best reasons to number your 'snd ' resources sequentially is so you can use `LoadARangeOfSounds()` and its counterpart, `ReleaseARangeOfSounds()`. It returns FALSE if there's a resource missing, so be sure you haven't skipped any numbers.

```
Boolean LoadAllSounds()
```

Loads every 'snd ' in the resource fork. If you have a lot of disk space tied up in 'snd ' resources, you probably won't want to call this function. Sound storage takes lots of memory, and you usually won't want every sound at your disposal to be loaded and locked at the same time.

The Release Sound Routines

Since sampled sounds are such memory hogs—22K per second of play time at the most common sampling speed—you'll want a way to liberate locked sounds when they're no longer needed so the memory manager can clear them away and make room for others.

These routines unlock locked sounds; that is, they make them relocatable and purgeable. The memory manager can then shuffle them around as needed for optimum utilization of heap space, and if things get really tight in the heap, it can dispose of them until they're needed again.

If release sound routines are called for 'snd ' resources that are not in memory at all, these routines find them on the disk and load them as relocatable, purgeable blocks. If you're willing to let the memory manager do all the work for you, you can skip the load sound routines and call `ReleaseAllSounds()` right from the start.

Like the load sound routines, these routines return a Boolean value so we can verify they worked. It's useful during debugging, when you're trying to whittle the game's memory needs down, but there's not much reason to send messages to the user. If there were enough memory to load the sounds, there's surely enough to release them.

```
Boolean ReleaseASound (short soundID0
```

When passed a valid 'snd ' resource ID number, `ReleaseASound()` finds the resource on disk or in memory and puts it in memory as a relocatable, purgeable resource.

```
Boolean ReleaseARangeOfSounds (short firstID, short lastID)
```

This calls `ReleaseASound()` a number of times in succession, loading and releasing resources from the `firstID` to the `lastID`.

```
Boolean ReleaeAllSounds ()
```

This loads and releases every 'snd ' in the resource fork. This function will not overflow or clog up your heap, regardless of how many sounds and how small the heap. However, it does return FALSE if any one 'snd ' resource is larger by itself than the available heap.

The Utility Routines

```
Boolean    IsSoundOn()
Boolean    ASoundIsPlaying()
short      SoundPriorityPlaying()
Boolean    ThisMacCanPlaySounds()
Boolean    ThisMacHasSoundChip()
```

These allow the rest of your program to get information from MGWSound.c. They're simple one-line functions that allow global access to local information.

For example, you may have a menu item called Sound On that the users can check or uncheck to turn sounds in the game on or off. If they are running on a Mac on which it is impossible to play sounds, you would probably want to gray out the menu item entirely. A call to the `ThisMacCanPlaySounds()` function lets you know if sound is even an option. `ThisMacCanPlaySounds()`, `ASoundIsPlaying()`, and `SoundPriorityPlaying()` can probably tell your program everything it needs to know about the status of your sound system.

The Play Sound Routines

And about time we got here. These are the routines that the rest of the unit supports. They actually make the sounds come out of the box.

These routines are versatile and robust. If you pass one of them an ID number of a sound that is in memory, it plays it. If the 'snd ' resource is still on the disk, they'll load it as a relocatable and purgeable block and play it. If there is no resource with that ID number, they won't try to play it; they'll return control immediately to the program instead.

These routines exit themselves if `soundIsOn` is FALSE, which seems appropriate, because usually that indicates that the user doesn't want to hear the sounds. They also exit if they can't find the chosen 'snd ' resource, which sure beats crashing, and usually indicates ResEdit has fallen into the wrong hands again.

Note

There's another reason these routines might not find a resource: The program may have called a 'snd ' ID number for which no resource exists, or which exists only in the mind of the programmer. If it were an insignificant little add-on feature that never quite got added on, doesn't affect game play, and wasn't noticed by you or your beta testers or your customers... maybe it's a good thing these routines exit themselves without alerting the user. A surplus of "Sound didn't happen" alerts might eventually be misinterpreted as "Programmer made poor career choice" alerts.

The routines do not provide error messages. If a desired sound doesn't play, not much is served by alerting the users. If they didn't notice, why intrude on their blissful ignorance with an error message? If they did notice, they don't need an "A sound didn't happen, so we're going on without it" alert. They already know that.

```
void PlayASound (short soundID, short priority)
```

The A in the middle stands for "Asynchronous." We shortened the name because you'll be typing it a lot. Also, this is the routine you'll usually call when you want to play a sound.

This routine plays the selected sound in the background while processing and graphics continue undisturbed. Well, that's how it's supposed to work, but the truth is, calling `SndPlay()` slows other CPU operations to roughly 80 percent of normal on the Pluses and SEs, which is another good reason not to go crazy on backwards compatability. In ancient times, Mac shareware programmers would get embarrassing email queries from their users, such as, "I notice I get higher scores when the sound is turned on. Can I get even higher scores by turning the sound up real loud?"

There are five if-then-exit sequences in `PlayASound()`.

1. The first checks whether or not the sound is on. If not (`gUserWantsSound`), then exit.
2. The second checks if a sound of higher priority is playing. We don't need a separate check to determine whether a sound is playing at all, because a nonsound has 0 priority and is thus deemed "interruptable."
3. The third gets the chosen resource or exits if it fails.
4. The fourth purges the `soundChannel` queue and chokes off any lesser sound which may be playing—a sound that didn't kick this routine out in check number two. If `soundChannel` is NIL, it creates a new `soundChannel`. If this routine fails (and we can't imagine why it might, but we'll check just in case), it exits.
5. And now we call `SndPlay()` and start the sound aplayin'. Since the last parameter passed to `SndPlay()` is TRUE, it will play asynchronously. `SndPlay()` passes these instructions to `soundChannel`, and then it's done. If there's a problem, the procedure exits.

So five checks have been performed, and if the procedure got this far, we can be quite confident the sound is playing properly. So we tell `soundPriority` the priority of the sound now playing.

Other routines (such as your next call to `PlayASound()`) want to know when this asynchronous sound is through playing, so we call `InstallCallBackRoutine()` to add a callback command to the end of `soundChannel`'s command queue. We could check for an error here, but hey, we're ready to exit the procedure anyway, so why bother?

```
void PlaySynchSound (short soundID, short priority)
```

It's similar to `PlayASound()`, except everything else in your program will stop while the sound is playing. The code is a bit shorter than `PlayASound()` because we don't have to create or check a sound channel (NIL is NIL; what could go wrong with that?), we don't have to pass the sound's priority to `soundPriority` (the sound is done playing before anything else can happen), and we don't need a callback (if anything else is going on, we know `PlaySynchSound()` is done).

This is a nice elegant routine, and you'll find similar routines in use in communication programs, business programs, and many games, but I haven't used a synchronized sound routine in years. It's here if you want it, but....

Why would you want your game to play a synchronous sound? Perhaps you'd want a game character to hold a pose of surprise while the dragon roars or the maiden sighs or whatever would cause Larry the Barbarian to choke up like that.

Ah, but we wouldn't want Larry the Barbarian to gain courage and savoir faire when the user turns the sound off, would we? So it might be better to `PlayASound()` (asynchronously), followed by `DoDelay()` (a routine in MGWUtility.c—pass it a number of Ticks equal to Jane's reaction time). That way Larry the Barbarian does the same double-take whether he hears the dragon's roar or merely sees the flame and breathes the sulfurous fumes.

There are less frivolous reasons you might want to pause while sound is playing. If the user reaches some benchmark or is awarded an extra life or a plethora of bonus points or the like, you may want to display a notice to that effect and play an appropriate sound—a cry of "All right!" or some inspirational music, perhaps. If so, you'll want a brief pause while the user absorbs the joyous news. Again, `PlayASound()` followed by `DoDelay()` gives consistent results whether `gUserWantsSound` is TRUE or FALSE.

Of course, if you absolutely insist that the users listen to your entire splash screen concerto before they are allowed to play, `PlaySynchSound()` is the routine for you.

The Feeping Creature Routines

These can be useful, and convenient, but they're easily overused, and you sure don't need them. As shown in HelloWorld.π, you don't really *need* anything but `PlayASound()`. Go ahead and use these when appropriate, but keep them on a short leash, lest you entertain yourself more than you entertain your game players.

```
void  PlayLoopSound (short soundID, short priority)
```

This is a pretty handy sound routine. It darn near got listed above the Feeping Creatures line.

`PlayLoopSound()` is the same as `PlayASound()` except when `PlayLoopSound()` encounters itself (or a sound with the same `soundPriority` number) already playing. In that case, it calls `FlushSoundSoon()` instead of `FlushSoundNow()` and inserts the selected sound into the queue. The sound currently playing continues to play, followed immediately by the `soundID` passed to `PlayLoopSound()`.

This is useful when making repeated calls for background sounds, to avoid "stuttering" due to the sound interrupting itself at inappropriate times or due to pausing between loops. If you're looping a sound that's six Ticks long and you pass that 'snd ' ID number to `PlayLoopSound()` and call it every five Ticks, the sound will repeat itself seamlessly forever.

```
void PlayLoopSoundOften (short soundID, short priority, short howManyLoops)
```

Here's just the ticket for stuffing `soundChannel` with a loop that repeats over and over and over and over again. Use it sparingly or end users will get real, real sick of it. Real, real, real, real, real sick of it.

```
void AddSoundToQ (short soundID)
```

`AddSoundToQ()` does not flush the command queue from `soundChannel`, nor silence `soundChannel`, nor check the priority of the sound currently playing, nor establish its own priority. What it does is add a command to play the selected sound to the end of the `soundChannel` command queue and set `soundPriority` to 0. Any call to any other MGWSound.c play sound routine will interrupt `AddSoundToQ()`, and `AddSoundToQ()` will add its sound to the queue created by any play sound routine, including itself. This routine resides outside the entire `soundPriority` system and must be called with care.

So what's `AddSoundToQ()` good for? Stringing a series of sounds together, rather like "concatenate" does for text strings. Put individual words in resource files and play them back as sentences. Record assorted barnyard animal sounds, sort them by pitch, store them as 'snd ' resources, and play them back as music.

The standard sound queue holds 128 commands; that's enough for some quite complex dialogue from ground control, or for a flock of sheep to baa a full chorus of "In the Mood." But don't go over 128 or it'll play synchronously (that is, everything but the sound will lock up) until the queue works its way back down to 128 commands.

That covers the routines accessible through the MGWSound.c interface. This unit includes many other routines which are called internally, and we think you'll be happier if you don't add them to the interface. But if you're a compulsive tweaker, you need to be particularly warned about...

The `CallBack()` Routines

Sound channels have a queue of commands that tell the channels what to do. This unit uses only sampled sounds, but square-wave and wave-table synthesizers are also available via sound commands, as well as amplitude and pitch change commands, commands to report synthesizer version and CPU load... a total of 24 commands. Their use is well documented in *Inside Macintosh*, Volume 6. For now, we can ignore 23 of these—the Toolbox routine `SndPlay()` and the 'snd ' resources themselves take care of their necessary commands—but we'll want to use `callBackCmd` ourselves.

When `callBackCmd` works its way up to the front of the command queue, it calls the program's `SoundCallBack()` routine. The `CallBack()` routines install and respond to `callBackCmd` to determine when asynchronous sounds have finished playing.

```
void SoundCallBack (SndChannelPtr chan, SndCommand theCommand)
```

Do not call this routine. The sound channel itself calls this routine. Because it may be called at interrupt time, it cannot directly move or allocate memory—`SoundCallBack()` cannot `GetResource()` reliably, for example. Also, `SoundCallBack()` must be loaded in the main segment of your program, so we load the entire MGWSound.c in the main segment.

All this program does is check to be sure it's responding to the proper `callBackCmd`, then change `soundPriority` to 0 without damaging the program's A5 environment. That's enough.

```
OSErr   InstallCallBackRoutine()
```

`PlayASound()` calls this routine as soon as the sound starts to play. This routine sticks a `callBackCmd` at the end of the command queue. As the sound plays, commands are executed. When the sound finishes playing, the `callBackCmd` is the only command left in the queue, so when the sound channel executes `callBackCmd` (which calls `SoundCallBack()`), we know the sound is done.

That covers the nuts and bolts of sound handling and the operation of MGWSound.c. To see (hear, actually) these routines in action, open HelloWorld6 again.

Now that you have the mechanics under control, you can concentrate on the arts. And if you're still not convinced of the importance of sound enhancing action on the screen, drop anything starring Arnold in the VCR and turn off the sound. Be sure to have someone wake you when the movie is over.

What Do I Do Next?

I guess I just sit around the house collecting huge royalty checks for this best-selling book, and... oh, I get it, you want to know what *you* do next.

Once you're comfortable with the principles presented here in chapter 0, you get to read the rest of the chapters. Now that you know how to walk, these guys are going to teach you how to run.

In the process of advancing your game programming skills, the gurus are going to contradict some of what I've told you here—the same way your high school art teacher told you that you didn't have to keep your crayons between the lines anymore. I've shown you how to hit a wiffle ball with a plastic bat. Now you can let the other authors coach you into the big league.

One last thing: I've included all my clam art in the 00 - Game Programming Primer folder on the CD-ROM. If anybody out there comes up with a real game featuring ambulatory clams, I'd love to see it.



Chapter 1
Your First Game

This book is about games, so rather than starting with small code snippets and doing nothing, let's go headfirst into the subject and start with a *game*! This first game is a simplified dungeon-digging game that I've called Dungeon. In the coming pages, I will list some of the source code for the first incarnation of the game, Dungeon 1, and describe how it works.

In this chapter, I will also discuss some problems you may have when designing games, list the most common game types, and give you some hints on what you need to know to make them. Finally, I will return to the Dungeon game and improve on it a bit.

Dungeon 1 is extremely simple and far from good enough to be distributed, even for free. The game should have an Apple menu and menus with New Game and Quit, it should allow the window to be dragged, and so on. It also should allow saving and restoring games, and it needs much more detail, different monsters and items. Let's forget all that for a moment—we'll add these niceties later.

Dungeon 1

Dungeon is a top-view dungeon-digging game, like NetHack but with graphics, in which you control a fantasy character (a knight or something like that) walking around in a dungeon, fighting monsters, and collecting treasures. In a real dungeon-digging game, there are countless types of monsters and treasures, many levels, much larger levels, and a much more complex combat system. It should also have better graphics, preferably objects that do not have to jump from space to space but are smoothly animated.

Dungeon 1 is on the CD-ROM included with this book; you'll find it in the Book Chapters folder, then in the 01 - Your First Game folder. Also on the CD-ROM is the complete code for the discussion that follows. The "1" in Dungeon 1 denotes that it is a first step, an introduction to the game that we will later refine quite a lot. As it is, Dungeon 1 is a very simple game using only a few Toolbox calls.

The game window in Dungeon shows a 15 x 12 map of square spaces (tiles). The tiles are 32 x 32 pixels each, arranged in a grid 15 tiles wide and 12 tiles high. All tiles that you (the player) have not yet seen are black, and the others are drawn with the contents of the space (see figure 1.1).

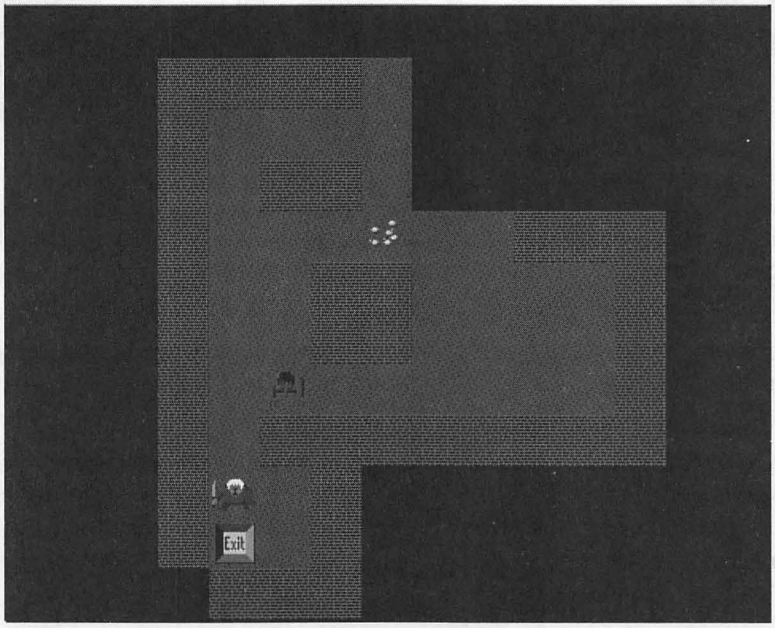


Figure 1.1 *Dungeon, our first demo game.*

Graphics

Dungeon uses a small number of pictures for the different contents a space can have:



The hero. I have drawn him from a top-view perspective so our game doesn't look too flat. He is facing in just one direction here. A real game would use at least four pictures, facing in four directions, and possibly different pictures depending on his equipment and health.



A troll. Purple skin and red eyes should make trolls look evil enough, don't you think? In a real game, the trolls, just like the hero, would be drawn in at least four directions, and of

course there should be many kinds of monsters. For making this demo short, we have only one.



Treasure. It's just a few coins scattered on the floor. Again, a real game would have many kinds of treasure.



The exit. This is where the hero should go to get to the next level.



Walls. This is an unpassable space. I chose to make this flat. We could use several different wall icons and pick one depending on its neighbors.



Floor, an empty space.

We use the `DrawPicture()` Toolbox call to draw everything. At this stage, it is perfectly appropriate. Few calls combine simplicity, flexibility, and compatibility as well as `DrawPicture()` does. We need to look for something better when we need more speed, but that is not critical at this point.

Sounds

Dungeon plays sounds when one of the following events occurs, with a different sound for each event:

- The hero attacks and hits.
- The hero attacks and misses.
- A troll attacks and hits.
- A troll attacks and kills the hero.
- A troll attacks and misses.

- The hero finds some treasure.
- The hero leaves for the next level.

For now, all sounds are played *synchronously* using the call `SndPlay()`. Playing a sound synchronously means that all action stops while the sound plays. Real games usually play sounds asynchronously, which means that the sound will play while the action continues. We will learn how to play sounds asynchronously in chapter 6, "Audio."

Moving Around and Fighting

The game polls the mouse button, and when it's pressed, the game program checks the cursor position. If the cursor is in a space next to the player's position, then the game inspects the space to see if the hero can move there (see figure 1.2).

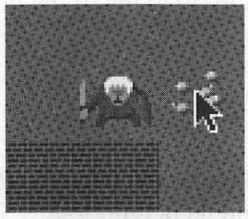


Figure 1.2 *Click on an adjacent space to move there.*

If you try to move to a space in which there is a troll, the hero attacks (see figure 1.3). When that happens, a random value is calculated, and if that number is within a certain range, the hero scores a hit and kills the troll.

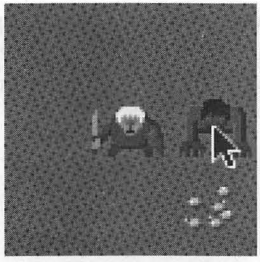


Figure 1.3 *Click on the troll to attack it.*

For each move the player makes, all trolls may also move. When a troll moves, it randomly selects if it should move toward the player or in a random direction. If the troll is close to the player, the chance of it moving toward the player is higher. If it tries moving into the player's exact position, it attacks. This differs from an attack by the player, because the hero has a number of hit points, so a single hit is not enough to kill the player.

The game ends when the player either dies or leaves through the exit.

Random Numbers

Most games need random numbers in some way in order to make the game not totally deterministic. The Toolbox includes a fairly good random number generator, named `Random()`, that returns a random short integer (16-bit). It is actually part of the QuickDraw drawing package. I will use this in the Dungeon demo in the function `Rand()`.

When using `Random()`, you should seed it by assigning the QuickDraw global `qd.randSeed` to a "random" value or you will get exactly the same random number sequence every time you use it. This may seem like a chicken-and-egg problem, but it's not. I simply use the `TickCount()` call.

```
qd.randSeed = TickCount();
```

When getting a random value from `Random()`, it is in the range -32,768...32,767. This is usually not what you want, but as long as you need numbers in a small interval, it's safe to do a modulo operation on the result.

For example, a six-sided die is quickly made as:

```
roll = (Random() & 0x7fff) % 6 + 1;
```

What I just did was call `Random()`, mask away the sign bit with a bitwise AND operation, take the result modulo 6, and add 1 to it. The result is an integer in the interval 1...6.

For small intervals, this gives us an even distribution. However, you can't use the modulo operation this way if you make random numbers in large intervals.

Take the following case as an example: You make random numbers in the interval 0...32,000. After the bitwise AND, you get a random number in the interval 0...32,767. The modulo operation takes the number modulo 32,001. This means that the numbers 32,001 to 32,767 are converted to 0 to 766, making these numbers twice as probable as others.

A solution to this problem is to call `Random()` repeatedly until `(Random()&0x7fff)` results in a value in the desired range.

Let's See Some Source Code!

Yes, yes, I'll get to the source code soon, but first I will suggest a flowchart for the program and list the source code piece by piece with explanations. Many of the calls and types I use will be discussed further in later chapters, but you will get a first look at them here.

The complete source code is listed on the CD-ROM, in the 01 - Your First Game folder. When I start explaining how it works, you may want to open the code files and see how this all fits together.

Building Dungeon

To write a program such as Dungeon from scratch, you would have to do the following:

- Draw all the graphics in a graphics program.
- Open ResEdit and create a new resource file. Copy each picture from the drawing program and paste it into the resource file using ResEdit. Check out what resource numbers the pictures get and change them if necessary. Copy all sounds from sound files into the resource file. Save and close the resource file. See chapter 2, "Basic Game Graphics," for more information.
- Open your development system. Create a new project. Add necessary libraries. Open a new code window and type the source code there. Save it and add it to the project.
- Add the resource file to the project. This technique varies with the development system. With Metrowerks CodeWarrior, you add it just like you add source files and libraries.
- Build and run!

Of course, you will go back and reiterate many steps as you develop the game. In particular, you will add more graphics, revise old drawings, add and change sounds, and last but not least, rewrite and debug the source code.

Understanding Dungeon

Figure 1.4 is a flowchart for Dungeon 1. After that, I will step through the code and describe how it works.

The flowchart is simplified quite a bit. In order to keep it reasonably simple, I do not include sound playing, nor do I include drawing. Drawing operations are made in the "Move" boxes, and sound playing as appropriate as part of some branches. As you can see, the main program is fairly simple—all the action is in `MovePlayer()` and `MoveEnemy()`.

In `MovePlayer()`, I first check if the clicked tile is a valid move. If it is, I take different actions depending on the contents of the tile the player clicked on: move, attack, or stay. Finally, `MoveEnemy()` is called for each tile that holds an enemy.

In `MoveEnemy()`, a move is decided, either towards the hero or randomly. Then, an action is taken depending on the tile the move leads to.

Declarations, Constants, and Variables

C programs tend to start by a number of #includes to define the library calls used. Dungeon uses a number of "managers" in the Macintosh Toolbox. However, most Toolbox units are included automatically (in Metrowerks C through MacHeaders), and for those, no #includes are needed. Out of all the managers Dungeon uses, only the Sound unit is not in MacHeaders, so it must be included. What you need to include varies with your development system and setup.

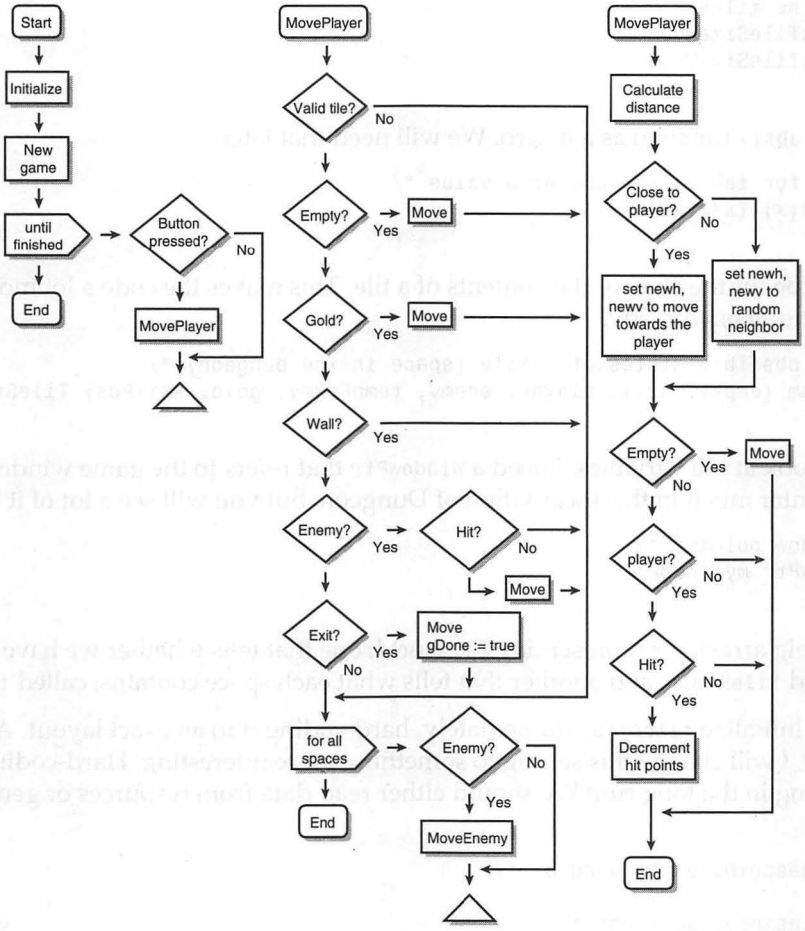


Figure 1.4 *Flowchart for Dungeon.*

```
#include <Sound.h>
```

I will use a game "world" consisting of 15 x 12 spaces and will later define two arrays representing them. In order to keep the actual numbers for the dimensions in as few places as possible, I define two constants for the size of the arrays:

```
/*Size of the array*/
#define   kArraySizeH 15
#define   kArraySizeV 12
```

I define constants for the size of the tiles. This may seem unnecessary, but is a very good idea in case I should decide to rescale the whole game one day.

```
/*Size of the tiles*/
#define   kTileSizeH 32
#define   kTileSizeV 32
```

I define the `abs()` function as a macro. We will need that later.

```
/* A macro for taking the abs of a value */
#define abs(x) (x>0?x:-x)
```

I define a type for the state or the contents of a tile. This makes the code a lot more readable than using numbers directly.

```
/* All the possible states of a tile (space in the dungeon) */
typedef enum {empty, wall, player, enemy, tempEnemy, gold, exitPos} TileState;
```

Now, let's look at the variables. I need a `WindowPtr` that refers to the game window. I will not use this pointer much in this incarnation of Dungeon, but you will see a lot of it later.

```
/* The window pointer */
     WindowPtr myWindow;
```

I need two big arrays for representing the world, one that tells whether we have seen a space or not, called `tileKnown`, and another that tells what each space contains, called `tileArray`.

This time, I initialize `tileArray` immediately, hard-coding it to an exact layout. At the end of this chapter, I will change this setting to something more interesting. Hard-coding is almost always wrong in the long run! We should either read data from resources or generate it randomly.

```
/* Arrays describing the dungeon */

/* What tiles have we seen? */
Boolean tileKnown[kArraySizeH][kArraySizeV];
/* What does each tile contain? */
TileState tileArray[kArraySizeH][kArraySizeV] = {
   {wall, wall, wall, wall, wall, wall, wall, wall, wall, wall, wall, wall},
   {wall,empty,empty,empty, wall, wall, wall, wall, wall, wall, wall, wall},
   {wall,player,empty,empty,wall, wall, wall, wall, wall, wall, wall, wall},
   {wall,empty,empty,empty, wall, wall,empty,enemy, gold, wall, wall, wall},
   {wall, wall, wall,empty, wall, wall,empty,empty,empty, wall, wall, wall},
   {wall, wall, wall,empty, wall, wall,empty, wall, wall, wall, wall, wall},
   {wall, wall, wall,empty, wall, wall,empty, wall, wall, wall, wall, wall},
   {wall, wall,empty,empty,empty,empty,empty,empty,empty,empty,empty, wall},
   {wall, wall,enemy,empty, wall, wall, wall, wall, wall, wall,empty, wall},
   {wall, wall,empty,empty, wall, wall, wall, wall, wall, wall,empty, wall},
   {wall, wall, wall, wall, wall,empty,empty,exitPos,wall, wall,empty,wall},
   {wall, wall, wall, wall, wall,empty, gold,empty, wall, wall,empty, wall},
   {wall, wall, wall, wall, wall,empty, gold,empty,empty,empty,empty, wall},
   {wall, wall, wall, wall, wall,empty,empty,empty, wall, wall, wall, wall},
   {wall, wall, wall, wall, wall, wall, wall, wall, wall, wall, wall, wall}
   };
```

The following variables tell where the player is (again, hard-coded according to the array above) and how many hit points the player has left.

```
/* Variables describing the player:*/
     Point playerPos = {1,2}; /* Must match the tile set to "player" */
     short playerHitPoints;
```

We need a Boolean variable that tells whether we should quit yet or not. It is initialized to FALSE.

```
/* A Boolean telling if we should quit yet or not */
     Boolean gDone = false;
```

For each picture, we need a `PicHandle` that can refer to it.

```
/* Pictures*/
     PicHandle floorTile;
     PicHandle playerTile;
     PicHandle enemyTile;
     PicHandle goldTile;
     PicHandle wallTile;
     PicHandle exitTile;
```

The following list shows vectors representing all eight directions in which we allow movement. Previously, I said that it is usually a bad thing to hard-code data in your programs. Here is an exception. The eight fundamental directions in the rectangular grid will never change, so we can safely hard-code them.

```
/*All eight directions as vectors*/
Point dirTable[8] = { { 0, 1 },
                      {-1, 1 },
                      {-1, 0 },
                      {-1,-1 },
                      { 0,-1 },
                      { 1,-1 },
                      { 1, 0 },
                      { 1, 1 }};
```

Subroutines

There are quite a few subroutines in our program. The first, `Rand()`, is a simple function for generating random values in a given range, as detailed previously in the section "Random Numbers."

```
/* A function that generates a value in the interval 0..range-1 */

static short Rand(short range)
{
```

```
  return (Random () & 0x7fff) % range;
}; /*Rand()*/
```

`DrawTile()` draws the appropriate picture for a given position. It starts by assigning the rectangle `tileRectangle` an area that encloses the tile. Note that I use the constants `kTileSizeH` and `kTileSizeV` for the dimensions of the game world rather than hard-coding the numbers to 15 and 12.

Next, we inspect `tileKnown` to see if the player is supposed to see what is in this position. If so, I inspect `tileArray` to find out what the tile should look like, and the program draws the appropriate picture for each case using `DrawPicture()`. If the tile is not known, or the value in `tileArray` is invalid, I fill the rectangle with black by calling `PaintRect()`.

```
/* Draw a tile */

static void DrawTile(short h, short v)
{
  Rect tileRectangle;

  SetRect(&tileRectangle, h * kTileSizeH, v * kTileSizeV, (h + 1) * kTileSizeH, ➥
(v + 1) * kTileSizeV);
  if ( tileKnown[h][v] )
    switch ( tileArray[h][v] )
    {
      case empty:
        DrawPicture(floorTile, &tileRectangle); break;
      case wall:
        DrawPicture(wallTile, &tileRectangle); break;
      case player:
        DrawPicture(playerTile, &tileRectangle); break;
      case enemy:
      case tempEnemy:
        DrawPicture(enemyTile, &tileRectangle); break;
      case gold:
        DrawPicture(goldTile, &tileRectangle); break;
      case exitPos:
        DrawPicture(exitTile, &tileRectangle); break;
      default:
        PaintRect(&tileRectangle);
    }
  else
    PaintRect(&tileRectangle);
} /*DrawTile()*/
```

`ShowAroundPlayer()` makes a double `for` loop over the nine tiles around the player. If any of those tiles are not known before (`tileKnown[h][v]` is not TRUE) then we set it to TRUE and draw it.

```
/* Set all tiles around the player to be known, and draw them if they were not ➥
known before. */

static void ShowAroundPlayer()
{
   short h, v;

   for ( h = playerPos.h - 1 ; h <= playerPos.h + 1 ; h++)
      for ( v = playerPos.v - 1 ; v <= playerPos.v + 1 ; v++)
         if ( ! tileKnown[h][v] )
            {
               tileKnown[h][v] = true;
               DrawTile(h, v);
            }
} /*ShowAroundPlayer()*/
```


`MoveEnemy()` makes one move with an enemy at the position `h,v`. It is called once for every enemy each time the player moves. This is a fairly long procedure.

```
/* Move an enemy */

static void MoveEnemy(short h, short v)
{
```

I declare a few variables...

```
   short dist;
   short newh, newv;
   short dir;
```

I calculate a simple distance measure from the enemy to the player. This is the "City Block" distance measure, which is the sum of the horizontal and vertical components. This is a poor distance measure, but good enough for now.

```
/*1: decide if we are close to enough to the player to "hear" the player*/

   dist = abs(h - playerPos.h) + abs(v - playerPos.v); /*City Block distance*/
```

The distance is used to determine whether or not the enemy hears the player. I call `Rand()` to get a random value, and if the distance is smaller than that value, the enemy hears the player and tries to move in that direction. If not, it moves in a random direction using `dirTable`. In both cases, we end up with the new position in the variables `newh, newv`.

```
/*2: Make a suggested destination, newh, newv*/

   if ( dist < Rand(15) )
      { /*Move towards the player*/
         if ( h < playerPos.h )
            newh = h + 1;
```

```
        else if ( h > playerPos.h )
          newh = h - 1;
        else
          newh = h;
        if ( v < playerPos.v )
          newv = v + 1;
        else if ( v > playerPos.v )
          newv = v - 1;
        else
          newv = v;
      }
    else
      { /*Move randomly*/
        dir = Rand(8);
        newh = h + dirTable[dir].h;
        newv = v + dirTable[dir].v;
      };
```

Now the troll knows where it wants to go. I get the `TileState` from `tileArray` and take different actions depending on what that particular tile contains.

```
/*3: Check what is in the destination and take appropriate action (move, fight)*/

  switch ( tileArray[newh][newv] )
    {
```

If the tile is empty, the enemy just moves to the new position. The old position is set to empty, the new is assigned to `tempEnemy`, and both are drawn. Note that we cannot set the new position to `enemy`, because we want to process each enemy only once, and the new position might not have been inspected yet.

```
    case empty:
        tileArray[newh][newv] = tempEnemy; /*We can't use "enemy," since then we➥
might process it again in the same          move!*/
        tileArray[h][v] = empty;
        DrawTile(newh, newv);
        DrawTile(h, v);
        break;
```

If the player is in the destination tile, the enemy makes an attack. It has a 40 percent chance to hit (if `Rand()` returns a 6, 7, 8 or 9—four out of ten!).

```
    case player:
        if ( Rand(10) > 5 )          /*Does it hit?*/
          {                          /*Enemy hits player!*/
```

If the troll hits, I decrement `playerHitPoints` and then check if the player has any hit points left. If there is still at least one point left, I play the sound for "player being hit."

To play sounds, I call `SndPlay()`. I pass a `SndListHandle` (a handle to a sound) to it, provided by `GetNamedResource()`. The sound is played asynchronously, so the program stops until the sound is finished.

```
            playerHitPoints--;              /*Reduce player hit points*/
            if ( playerHitPoints > 0 )
               {                            /*Player still lives!*/
               if ( noErr != SndPlay(0L, (SndListHandle)GetNamedResource('snd ➥
', "\pPlayer hit"), false) )
                  ;
               }
```


If there are no more hit points, the player dies, and I play the "player dies" sound and set the `gDone` flag to TRUE.

```
            else
               {                            /*Player died!*/
               if ( noErr != SndPlay(0L, (SndListHandle)GetNamedResource('snd ➥
', "\pPlayer died"), false) )
                  ;
               gDone = true;                /* Game over -- quit! */
               };
            }
```

On the other hand, if the enemy misses, we play a sound for that.

```
         else
            {                               /*Miss!*/
            if ( noErr != SndPlay(0L, (SndListHandle)GetNamedResource('snd ', ➥
"\pPlayer miss"), false) )
               ;
            };
         break;
```

In all other cases, I don't allow the enemy to move.

```
      case wall:
      case enemy:
      case gold:
      case exitPos:
      case tempEnemy:
         ;                                  /*Don't move to any of these!*/
   }; /*case*/
} /*MoveEnemy()*/
```

In `InitDungeon()`, I set up for the game by creating the game window and loading pictures.

```
/* Initialize -- create window, load graphics */
```

```
static void InitDungeon()
{
   Rect windowRectangle;
```

The window is created with `NewCWindow()`. The `C` stands for color, which means that it is part of the Color QuickDraw package and will not work on the oldest Mac models, such as the Mac Classic. There are a whole bunch of parameters that we will not dig into here; see appendix B, "Macintosh Environments," for more detail.

```
/*Set up the window*/
   SetRect(&windowRectangle, 50, 50, 50 + kArraySizeH * kTileSizeH, 50 + kArraySizeV ➥
* kTileSizeV);
   myWindow = NewCWindow(0L, &windowRectangle, "\pDungeon 1", true, 0, (WindowPtr)-➥
1L, false, 0);
   SetPort(myWindow);
```

We must seed the random number generator, or `Random()` will return the same value every time. I use `TickCount()` for this. `TickCount()` returns the number of "Ticks" (each Tick being a 60th of a second) since the Mac started. That is random enough for most needs. You need to seed the random number generator only once, at the beginning of your program.

```
   qd.randSeed = TickCount ();        /*Seed the random number generator*/
```

All pictures are loaded with the `GetPicture()` call.

```
/*Load all pictures*/
   floorTile = GetPicture(128);       /'PICT' resource #128.*/
   playerTile = GetPicture(129);      /'PICT' resource #129.*/
   enemyTile = GetPicture(130);       /'PICT' resource #130.*/
   goldTile = GetPicture(131);        /'PICT' resource #131.*/
   wallTile = GetPicture(132);        /'PICT' resource #132.*/
   exitTile = GetPicture(133);        /'PICT' resource #133.*/
} /*InitDungeon()*/
```

In `MovePlayer()`, I process a mouse click by trying to move the player to the position being clicked on.

```
/* Try to move the player to the position where we clicked. */

static void MovePlayer(Point clickedTile)
{
short h, v;
```

First of all, I check if the click is within the bounds of the arrays. You can never check for array bounds too often!

```
/* Valid tile?*/
if ( clickedTile.h >= 0 )
   if ( clickedTile.v >= 0 )
```

```
    if ( clickedTile.h < kArraySizeH )
      if ( clickedTile.v < kArraySizeV )
```

If the click is within bounds, I check if the tile being clicked on is within one step from the player, both horizontally and vertically.

```
/* OK, we are inside the game area, clicking in some space! Is it next to the ➥
player?*/
          if ( clickedTile.h >= playerPos.h - 1 )
            if ( clickedTile.h <= playerPos.h + 1 )
              if ( clickedTile.v >= playerPos.v - 1 )
                if ( clickedTile.v <= playerPos.v + 1 )
```


Once all those checks have passed, I get the `TileState` from `TileArray` and take different actions for each case.

```
/* Yes! What is there? */
                    {
                      switch ( tileArray[clickedTile.h][clickedTile.v] )
                        {
```

If the tile is empty, just move there. This is similar to how the enemies move.

```
                          case empty:
                               tileArray[playerPos.h][playerPos.v] = empty;
                               tileArray[clickedTile.h][clickedTile.v] = player;
                               DrawTile(playerPos.h, playerPos.v);
                               DrawTile(clickedTile.h, clickedTile.v);
                               playerPos = clickedTile;
                               break;
```

If it's a tile with treasure, I move there and play a sound. If I've been keeping track of how much treasure the player has found so far, it should be accumulated here. I skipped that feature in this version of Dungeon, but you will find it in Dungeon E. It's just a question of adding a score variable, adding points to it, and displaying it.

```
                          case gold:
                               tileArray[playerPos.h][playerPos.v] = empty;
                               tileArray[clickedTile.h][clickedTile.v] = player;
                               DrawTile(playerPos.h, playerPos.v);
                               DrawTile(clickedTile.h, clickedTile.v);
                               playerPos = clickedTile;
/* We could add score here */
                               if ( noErr != SndPlay(0L, ➥
(SndListHandle)GetNamedResource('snd ', "\pMoney"), false) )
                                  ;
                               break;
```

If it is a wall, I play a `SysBeep()` and don't move.

```
case wall:
  SysBeep(1);
  break;
```

If an enemy is here, the player attacks it. This works similarly to when an enemy attacks the player, with just a few differences:

- The player has a 50 percent chance to hit.
- The enemies have no hit points. They die from one hit.
- We use other sounds.

```
case enemy:
  if ( Rand(10) > 4 )
    { /*Hit! Play a "monster died" sound*/
      if ( noErr != SndPlay(0L, ➥
(SndListHandle)GetNamedResource('snd ', "\pEnemy died"), false) )
        ;
      /* Walk to the space where the monster was.*/
      tileArray[playerPos.h][playerPos.v] = empty;
      tileArray[clickedTile.h][clickedTile.v] = player;
      DrawTile(playerPos.h, playerPos.v);
      DrawTile(clickedTile.h, clickedTile.v);
      playerPos = clickedTile;
    }
  else
    { /*Miss! Play the "miss" sound. */
      if ( noErr != SndPlay(0L, ➥
(SndListHandle)GetNamedResource('snd ', "\pEnemy miss"), false) )
        ;
    };
  break;
  /*attack!*/
```

If the location is the exit tile, I move there, play the exit sound, and set the `gDone` flag. Happy ending—the player escapes the dungeon!

```
case exitPos:
  tileArray[playerPos.h][playerPos.v] = empty;
  tileArray[clickedTile.h][clickedTile.v] = player;
  DrawTile(playerPos.h, playerPos.v);
  DrawTile(clickedTile.h, clickedTile.v);
  playerPos = clickedTile;
  if ( noErr != SndPlay(0L, ➥
(SndListHandle)GetNamedResource('snd ', "\pNext level"), false) )
    ;
  gDone = true;
```

```
                    break;
                }; /*case*/
            };
```

After processing the player move, I call `ShowAroundPlayer()` in order to update any tiles the player hasn't seen before.

```
ShowAroundPlayer();
```


For each move the player makes—even if it fails—the enemies may move. I make a double `for` loop, looping over the entire grid looking for enemies. For each one I find, I call `MoveEnemy()` to let it move.

```
/* Monsters are allowed to move now!*/

/* Move all enemies!*/
for ( h = 0 ; h < kArraySizeH ; h++)
   for ( v = 0 ; v < kArraySizeV ; v++)
      if ( tileArray[h][v] == enemy )
         MoveEnemy(h, v);
```

When all enemies have moved, I loop over the array again to change `tempEnemy` to `enemy`.

```
/* After moving, replace tempEnemy by enemy.*/
for ( h = 0 ; h < kArraySizeH ; h++)
   for ( v = 0 ; v < kArraySizeV ; v++)
      if ( tileArray[h][v] == tempEnemy )
         tileArray[h][v] = enemy;
} /*MovePlayer()*/
```

`InitToolbox()` makes some initialization calls, the most important ones. You must perform the following function calls at the beginning of every Macintosh program. They initialize the various parts of the Toolbox and are required by the Metrowerks programming environment.

```
/* Standard inits */

static void InitToolbox(void) {
   InitGraf (&qd.thePort);
   InitFonts ();
   FlushEvents (everyEvent,0);
   InitWindows ();
   InitMenus ();
   TEInit ();
   InitDialogs (0L);
   InitCursor ();
}
```

The Main Program

The main program is very short, and that is the way it should be. Most programs follow the structure you see here: Initialize the Toolbox, make some calls to initialize the program, and then run a loop until the program ends.

```
/* Main program */

void main(void)
{
Point   clickPoint, clickedTile;
short   h, v;

InitToolbox();
InitDungeon();
```

As a part of the initializations, I draw all tiles and call `ShowAroundPlayer()` to open the eyes of the player, so to speak.

```
/*Draw all tiles!*/
for ( h = 0 ; h < kArraySizeH ; h++)
  for ( v = 0 ; v < kArraySizeV ; v++)
    DrawTile(h, v);

ShowAroundPlayer();
```

Now the program loops until `gDone` turns TRUE.

```
/*Initializations done! Run the game loop until the game ends.*/
do
{
```

The call `Button()` returns a Boolean that is TRUE if the mouse button is held down.

```
if (Button()) {
```

If that is the case, I put the mouse position into the point `clickPoint` using the call `GetMouse()`.

```
  GetMouse(&clickPoint);   /* Get the position of the click */
```

By dividing with the tile size constants, I get the tile coordinates.

```
  clickedTile.h = clickPoint.h / kTileSizeH; /* Convert to grid */
  clickedTile.v = clickPoint.v / kTileSizeV;
```

The coordinates are passed to `MovePlayer()`, which calls the other routines.

```
  MovePlayer(clickedTile); /* Try to move there */
```

Finally, I wait until the mouse button is no longer pressed, so the player won't make several moves by mistake.

```
   do {} while (Button());     /* Wait until the mouse click ends */
   };
} while (!  gDone);
```

Done! Before quitting, I call `FlushEvents()` to discard all the mouse events that are queued when the player clicks the mouse. If we don't, those events may go into the Finder instead.

In later chapters, I will show how to use the Event Manager to handle mouse clicks, rather than by direct calls to `Button()`.

```
FlushEvents(mDownMask, 0);     /* Get rid of mouse down events! */
} /*Dungeon main program*/
```

To summarize, I used the following Toolbox calls:

- `SetRect()`—Sets the fields of a rectangle.
- `NewCWindow()`—Creates a window.
- `SetPort()`—Directs drawing to the chosen port (i.e. window).
- `GetPicture()`—Loads a 'PICT' resource.
- `DrawPicture()`—Draws a 'PICT' resource.
- `Button()`—Checks if the mouse button is pressed.
- `GetMouse()`—Checks where the mouse cursor is located.
- `SysBeep()`—Makes a system beep sound.
- `SndPlay()`—Plays a sound.
- `Random()`—Returns a random integer.
- `FlushEvents()`—Discards events. Real programs process the events instead.

We also used a bunch of initialization calls that are not listed here.

We will visit this program again later and improve on it.

But How about My Own First Game?

Perhaps dungeon-digging was not what you had in mind for your own first game. Perhaps you want something nicer, like a puzzle game or an adventure game with some plot in it, or something more advanced?

Perhaps you want to try something harder as your first project? If you are a beginner to Mac programming and have some doubts about your ability to master the Mac, you should make

some simple game first, just to make the standard mistakes on a small project. If you are a beginner and still don't have any doubts... well, good luck. I won't stop you, but I did warn you.

Problem 1: A Good Idea

First of all, you need an idea. Even if you plan to make up the game as you program, you need some initial idea from which to start. If you have a good concept in your mind already, write it down and refine it, check that it makes sense, and then go on with the actual game design.

If you think you don't have any ideas, go out looking for them. Check out existing games on computers and in the arcades. Watch movies and read books and comics with your eyes open for that great idea. Have a look at real life—perhaps there are some unexploited gaming options right in front of you. After all, what could be more familiar?

Perhaps you feel like making a game based on an old game? Perhaps reviving a great game from the past? A large majority of games are based on some old idea with some new additions or changes. Anyone can come up with the idea of "Castle Wolfenstein with Barney as the hero"—it's up to you to make it fast and engaging and to push the envelope of the Macintosh... in short, to make it great.

Basing Your Game on an Existing Game

This is what most game makers do, at least to some extent, including myself. For example, you can decide that you want to make a Qix for the Mac (you know, that arcade game in which you chop away parts of the game area, but must do so without getting hit by the "Qix," which is moving around over the parts you haven't captured yet). In such a case, an important problem is how to mimic an old game without infringing on copyrights.

Wrong way: You make an exact clone of the original game and release it as Qix (or perhaps MacQix). This is bad. The owners of the original game might, if you are lucky, force you to destroy all copies and source code for your game. If you aren't lucky, they will sue your pants off.

Slightly better way: You make an exact clone of the original game and go to the owners of the Qix game and ask if you can sell it, or if they want to. If you are really lucky, they might buy it from you. If you are not so lucky, they still may force you to destroy all copies and source code for your game.

Questionable way: You make a game that works exactly as the original game, but change all graphics to something that looks different, and also change the name. This will probably work, but you can't be sure. Also, it isn't particularly innovative.

Right way: You make a game that has the same fundamental principle as Qix (cutting away parts of a larger free area with some dangerous enemy within it), but make some substantial changes and additions that makes the resulting game "a kind of Qix" but still far from a

clone. You might replace the "Qix" (the bunch of moving lines) by a couple of sprites. (This was actually done in an old shareware Qix-like game for the Mac named Iggy Iggopolis.) For example, Galaxian and Phoenix are variations on Space Invaders, but the changes in not only how the enemies look, but also how they behave and what options the player has, make these games into new games.

The perfect way to do your own version of a classic is to completely change the setting for the game. If you have a background story and plot that is completely different from the original game, it can appear as a completely new and original game—and in a way, it is. The plot is a big part of a game.

When making computer versions of board games, you should think along the same lines. If you make a game similar to Risk, don't call it Risk and don't use the same world map as Parker does. Change the rules, change the name, and change the way the map is divided. Perhaps put it in a setting that isn't about world conquest at all. The users won't think it's "real" Risk, and that is exactly why Parker won't sue you for making it.

Making a Completely New Game

But why base your game on an old idea? Why not come up with a new one? Well, new ideas that are really useful are hard to come by. It takes much more design effort, and it might not work out in the end. On the other hand, once it is made, the chance of getting rich from it is much greater than from yet another Space Invaders or another Wolfenstein.

Can You Do It?

Do you have all the knowledge and tools to actually make what you have in mind? This book will provide you with a lot of help, advice, and code for your game programming library, but it is still you who must put it all together into your dream game. Are you sure that you are ready to make the next big game hit now? Perhaps you should make a smaller project first to get some of the inevitable mistakes out of your way.

It takes a lot to make a really good game, especially a lot of *time*. A game of commercial quality cannot be produced in just a month or two.

Do you have the right tools? Do you have a compiler that you are comfortable with? You need a painting program such as Photoshop, and you might also need a renderer such as Infini-D. You need sound input (which might just mean getting a microphone for your Mac) but also a sound editing program. If you want music, you need the tools for putting that together—perhaps a MOD file editor and a driver with which to play the files.

Can the Computer Do It?

Even if you envision a great game concept, you have to consider if it is realistic on today's computers. If you have a Power Macintosh, you have plenty of processing power, but there is a limit to everything. A simulation game similar to SimCity but in which each and every

inhabitant of the city is simulated in detail could be fun, but there's no computer that would do that today with acceptable speed.

An even more important question is: Can your users' computers do it? If you make an ambitious game, it might be too slow on the low-end Macs. If it is, you should consider making it possible to switch to settings that give a lower CPU load, such as lower resolution, less detail, and so on.

The demand on the computer can be lowered a lot by designing the game properly. I have seen games that take forever to load and that eat enormous amounts of memory, just because they pregenerate a huge world that should have been built step by step, saving loading time and lowering the memory demand to what people actually have.

Mac Systems

Ask yourself: What is the minimum Mac system that is of any interest for me to support and make sure my game is enjoyable on? There are plenty of old Macs out there, but most of the software sales are for Macs that were bought fairly recently. If you start developing today, the current low-end will still create significant sales when you finish, so it is reasonable to make an effort to support them. Should you support older Macs as well, down to, say, the original LC? There are plenty of those old Macs and they do generate some sales, but not enough to make compromises that will make your game lame on fast Macs compared to your competitors.

With the Power Macs moving into the low-end market quickly, phasing out the 68040s, you can even consider skipping the 68K-based Macs. What you should support depends a lot on your game. Consider the following levels:

Power Mac 6100/60

PPC 601, 60 MHz, 8 MB of RAM, thousands of colors

If this is your minimum configuration, I hope your game will survive long enough to take advantage of the future larger quantities of Power Macs and third-party clones. You skip the entire installed base of 68K Macs. If you do, you better have good reasons.

LC 475

68040, 25 MHz, 8 MB of RAM, 256 colors

This is a rather ambitious minimum. The LC 475 is fast enough for rather impressive graphics. Running texture-mapping 3-D games is no big problem if you run in a small window. There is little reason to demand more as minimum configuration.

LC III

68030, 25 MHz, 4 MB of RAM, 256 colors

This is a reasonable level which gives fairly fast graphics, enough for many action games, though perhaps not the top-notch ones.

PowerBook 160

68030, 33 MHz, 4 MB of RAM, 16 grays

Here is an interesting configuration. The second generation of the PowerBooks isn't too old yet and has decent horsepower, but many of them have only 16 grays. It is not much of a game computer, but pretty interesting when making a networked game.

LC and Color Classic

68020/68030, 16 MHz, 4 MB of RAM, 256 colors, narrow data bus

Apple persisted for a long time to sell the crippled narrow bus design (16 bits wide rather than the usual 32). You find it in the LC, the LC II, the Classic II, and the Color Classic. That's a *lot* of Macs, and many of them are used for games! However, they are much slower than the LC III. Slamming picture data on the screen is so slow that you may decide to skip them.

Classic

68000, 8 MHz, 1 MB of RAM, black-and-white, no Color QD

At the extreme bottom level we find the old Classic design. This too is a level with a pretty big installed base, with all Mac Pluses, SEs, Classics, and PowerBook 100s. Still, if your game needs speed, it is hardly worth the effort. Most owners of the old "toasters" have given up buying new games (since there are so few new games supporting their platform) and use them exclusively for word processing.

There is one exception: networked games. All the homes with one old Mac and one newer one should not be ignored.

Screen Sizes

The dominating screen size on Macs used for games is 640 x 480. The smallest screen is the 512 x 342 "Classic" screen, with the 512 x 384 12" color and Color Classic screens being not too uncommon. Older black-and-white PowerBooks have 640 x 400 pixels.

If possible, support *both* the 640 x 480 and a smaller size (512 x 384 if you demand 8-bit color, 512 x 342 otherwise)—and perhaps even larger sizes. See if it's possible to cut down on some of the display around the primary game area in order to make it fit on a small screen, and still fill the 640 x 480 nicely.

The 640 x 480 monitor should be your primary target. Fill it completely. If your game demands that players with smaller screens get a compromise, with some displays gone, made very small, or switching back and forth, while players with bigger screens can see all the displays, so be it. As long as your compromise is playable, it is better than not being able to run your fun game at all.

Number of Colors

Most Mac gamers have 256 colors (8-bit color) nowadays. Obviously, you should make your game look as good as possible in 256 colors. Should you support other depths? Many games allow nothing but 256 colors for the following reasons:

- "Game mechanics depends on color." If you have played Oxyd or Diamonds, you know that this excuse doesn't hold. You can easily replace colors with patterns and symbols. It takes some extra trouble, but it usually can be done.
- "The graphics depend on some technique that works only in 256 colors, for instance color table animation or optimized ray-casting routines." I can accept that.
- "The game has direct-to-screen graphic routines that work only in 256 colors." This is a poor excuse, since all games that write directly to the screen should have a "safe" mode, allowing the user to switch to QuickDraw. Demand 256 colors when using the direct-to-screen, by all means, but if you use QuickDraw, you can allow other depths (such as thousands of colors) as long as the Mac is fast enough.
- "Graphics quality." The game looks too poor in 16 colors, and the author doesn't have the time/motivation to adapt the graphics.

The last point is perhaps the most important reason to demand 256 colors. Most graphics made for 256 colors look downright awful in 16. The result can be improved a little bit by dithering, but only up to a certain point.

There is one simple solution that will add a fair bunch of PowerBook owners to your potential buyers: Make it run in 16 grays! A 256-color game that looks awful in 16 colors often looks quite nice in 16 grays! Lode Runner: Legend Returns is an example of 256 colors and 16 grays supported for exactly these reasons.

Note
Sixteen colors are few—16 grays are plenty.

Of course, if your game doesn't look good even in 16 grays, stick to 256 colors or more.

Can Your Players Do It?

Before doing too much coding, ask yourself the question, "Will this game be playable?" This is a question of good game design and especially a well-designed user interface. Much can

be fixed as you go on, but you will save time if you think about it before getting too far. What controls should be used? Are there any indicators that the player needs? How can you make their meaning obvious?

If the game is complex, do what you can to simplify it. If the game is a shooter with many different weapons, either have one weapon selector key and one fire key or find some other way to make the weapon selection simple. If the player must memorize two or three keys for each weapon, chances are high that the game gets tiresome.

Flight simulators are typical games for which the complexity easily gets overwhelming. Some flight simulators are so complex that there is no chance to play without having the manual in front of you all the time. Even though the idea of manual protection may seem appealing to those who are wary of software piracy, this is not the right way to make the manual necessary.

Some game ideas can be tested by making quick prototypes and playing them without a lot of fancy graphics but with the game mechanics in place. Use that quick hack for evaluation, and make a new program when making the real game. Some ideas can be tested as HyperCard or Macromedia Director hacks.

All Those Games

Have you already decided on a game idea that you want to implement? Perhaps you are still looking for ideas? Let's browse the different types of games we have out there already, and look at how much you have to learn to make one of them.

In this section, I will list several types of computer games and discuss some implementation issues for each. Most games fall into one of the following categories, but you definitely can't categorize every game. Actually, the best new games are probably the ones that either can't be put in any category or fall into several.

I will make some notes on what techniques each kind of game uses for its graphics and what representation it must use (i.e., data structures). My statements here are not definitions of how you have to make each kind of game, but rather suggestions of how they normally would be implemented on the Mac. The techniques suggested will be discussed in detail in later chapters.

2-D Space Games

This is one of the oldest computer game types. It started with the ancient SpaceWar game. With Space Invaders and Asteroids, it reached the arcades. On the Mac today, we have a fair number of 2-D space games in the shareware market, such as Solarian, Rescue, Cyclone, Maelstrom, Missions Of The Reliant, Asterax, Slime Invaders (see figure 1.5), and many others.

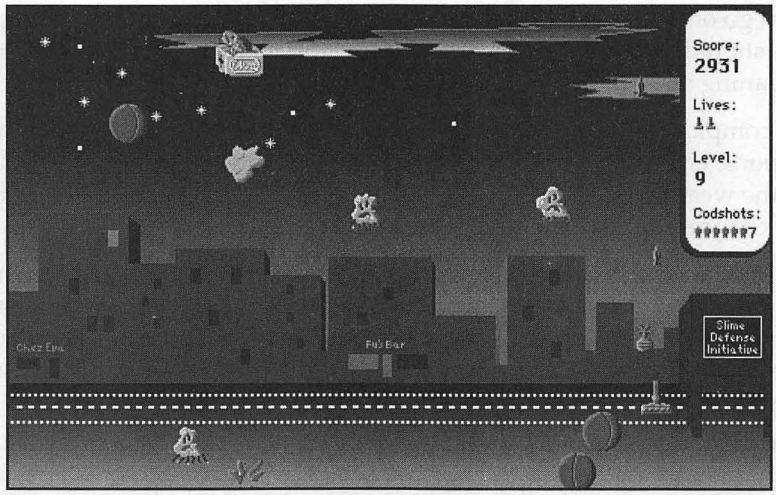


Figure 1.5 *Slime Invaders, this author's freeware contribution to the many 2-D space games for the Mac.*

Graphics

A 2-D space game can be made using sprite animation techniques using off-screen drawing, though that can be simplified a bit since you usually use a black background. You may also want stars moving in the sky. If so, you probably have to use direct-to-screen drawing for making the stars, or it will be too slow on low-end Macs.

Representation and Behavior

These games are rather easy to design. You need some kind of list of all the objects, so you can go through that list when drawing and making collision detection.

Platform Games

One of the first platform games was Donkey Kong, which has led to lots of followers, including Dark Castle and Prince Of Persia. Some shareware games in the genre are Glypha and Quagmire (see figure 1.6).

In a platform game, you get a side view of a 2-D world. Objects can move up, down, right, and left, but not in or out. The characters in the game move on platforms, often climbing on ladders.

Graphics

As long as you don't want your platform game to be continuously scrolling (see the "Side-Scrollers" section coming up), nothing stops you from making platform games that will

work just fine on any Mac. After all, Dark Castle ran just great on an 8 MHz Classic. Your approach should be to use sprite animation techniques, drawing off-screen and copying changed parts to the screen.

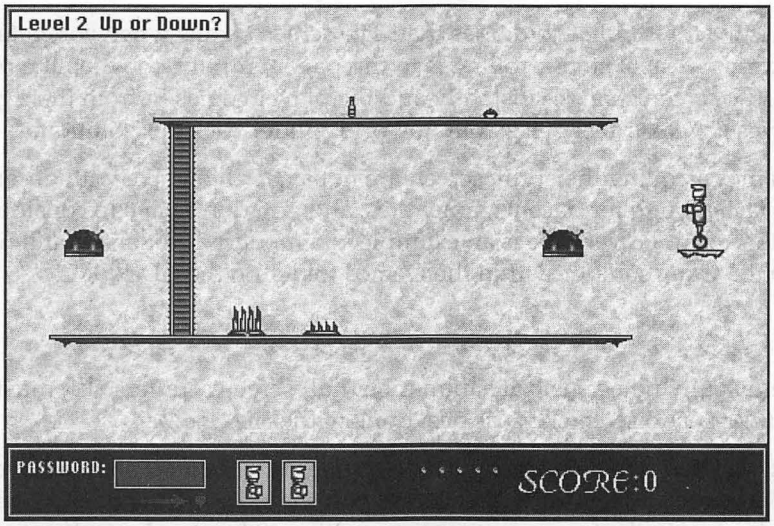


Figure 1.6 *Quagmire, a good shareware platform game.*

Representation and Behavior

A platform game uses sprite animation just like the 2-D space games, with a list of all moving objects, but it must also handle the platforms in some way. There are several ways to accomplish this:

- You can make each platform an object, and make other objects bounce off it.
- You can use an underlying grid that shows which parts are blocked and which are passable.
- You can use a set of rectangles or polygons representing all passable areas. When an object moves outside its current rectangle, you check if there's some other rectangle it can move into. If not, it should bounce back, or be destroyed, as appropriate for your game.

Side-Scrollers

In the arcades and on dedicated game platforms, you often find games that scroll the image, usually sideways. On the Mac, scrolling games have been few, mostly because a 16 MHz, 16-bit data bus LC can't move data to and from memory fast enough. There were still a few of these games, even when the LC was the home Mac of choice—for example, Sky Shadow, Karnov, and Iraq Attack.

Some side-scrollers can get by with less processing power if they don't require the entire screen to be copied for every frame. A typical example is Defender. Defender clones on the Mac, for example MacLanding or StarRunner/Delirium, clearly demonstrate that even yesterday's low-end Macs are fast enough for these games.

Another trick is to step-scroll. Oxyd does that: The screen scrolls when you get near the border. An extreme case of step-scrolling is to flip pages entirely, so when the player goes out to the left, the screen changes instantly and the player comes in from the right in the next scene. This gives a weaker feel of a scrolling world. Glider uses this solution.

Nowadays, games scrolling the entire screen are not as hard to make with decent results. The LC 475 is fast enough for decent frame rates, and you don't have to consider much slower Macs than that. Power Pete is an example of a modern scroller—you'll find a demo of Power Pete in the Games folder within the Demos folder on the CD-ROM.

Graphics

A side-scroller usually relies on sprite animation. Full-screen scrollers also must copy the image from off-screen to screen. This is best done with `CopyBits()`.

An important question when designing a side-scroller is whether it should have a larger off-screen (of which you can see a part) or whether you should generate the new parts as they enter the visible area. In the former case, your world can't be very large. If you use an off-screen more than about four times the size of the screen, it will take lots of time to initialize and eat so much memory that it prevents low-end users from running it.

In the situation in which you generate the world as you go on, memory consumption goes down a lot. On the other hand, the drawing process gets more complicated.

Representation and Behavior

If you use a larger off-screen with the entire world, the game is essentially just a sprite-using program that copies a part of its game area for every frame. This option requires nothing extra.

However, if you generate the world as it comes, you may need some representation for handling this ongoing world generation. If you make a game like Scramble, for example, you probably have a large list describing the shape of the ground as it appears.

A very simple representation would be a single integer variable that denotes the height of the ground at the right edge of the screen (assuming that the screen scrolls from right to left). When the screen scrolls, you paint the ground from the bottom up to the height that the variable represents. Then, you may want to modify the variable in order to vary the height of the ground.

Board Games and Puzzle Games

Board games are often easy to program, which is why they are common first choices for new game programmers. Such games include Memory, Tic-Tac-Toe, GoMoku, Master Mind, and Sokoban (see figure 1.7). The most popular first game project is obviously Mines, also known as Minesweeper. Did you know that there are more than 20 different implementations of this game released on the Internet?

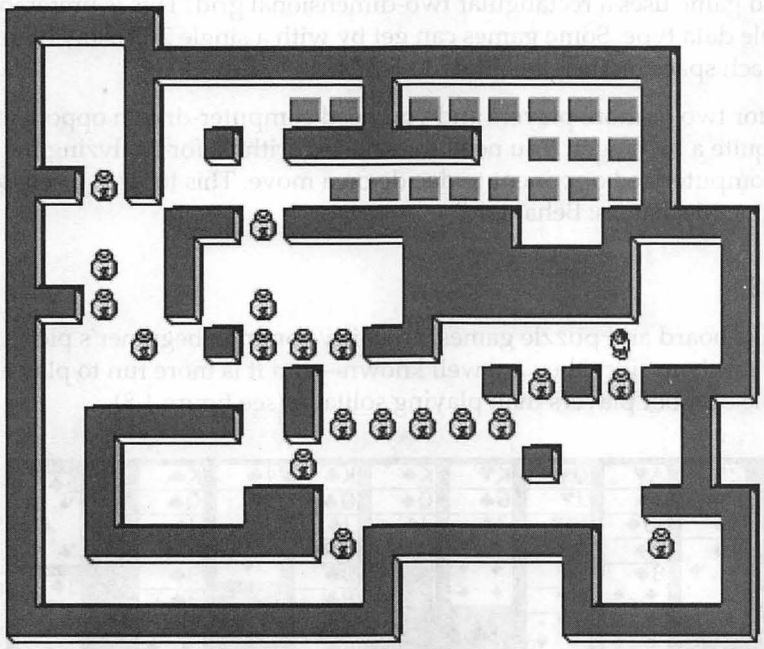


Figure 1.7 *MacSokoban, one of the Mac implementations of the Sokoban puzzle game.*

However, board games and puzzle games are not only for the beginner. In this category, you also find some of the most successful and entertaining commercial computer games ever made. The first big hit of this type on the Mac must be The Fool's Errand, with followers like 3 In Three and The 7th Guest. An important aspect of those games is that they are significantly more attractive to women than the average action game is.

Graphics

Most board games need nothing more than QuickDraw drawing, probably not even off-screens. As demonstrated by Dungeon, you get far by using some 'PICT' resources that you draw with `DrawPicture()`.

However, you can improve the quality of a board game a lot by using a little bit of sprite animation, allowing the user to click-and-drag the pieces from one space to another rather than just erasing a piece in one space and drawing it in another. A compromise that is fairly common in older games, but less common now when we can expect more memory and more speed, is to drag an outline of the piece.

Representation and Behavior

A typical board game uses a rectangular two-dimensional grid. This is preferably a 2-D array of some suitable data type. Some games can get by with a single 2-D array of integers representing each space on the game board.

If the game is for two or more players and you need computer-driven opponents, the problem gets quite a bit bigger. You need to make algorithms for analyzing the position in order for the computerized opponent to decide on a move. This topic is discussed further in chapter 8, "Game Mechanics: Behavior."

Card Games

Card games, like board and puzzle games, are fairly common beginner's picks, but there is a decent market for them since they are well known—and it is more fun to play a card game against a couple of robot players than playing solitaire (see figure 1.8).

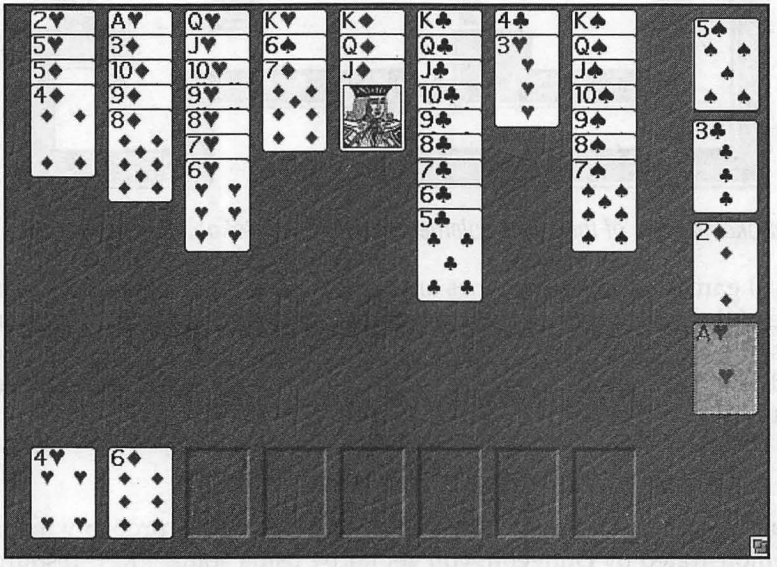


Figure 1.8 *Eric's Ultimate Solitaire, a top-notch solitaire card game.*

What makes card games different from board games is that most card games have no copyright owner, so you can't get sued for releasing a Poker game, while you can if you make a Monopoly rip-off.

Graphics

Card games can be implemented like board games. However, card games have even more need of dragging than board games do.

Representation and Behavior

You need to represent the deck of cards, of course, either as arrays or sets. You must also represent hot spots in the game window where it is allowed to put cards. You might want to draw rectangles on the screen at the hot spots so the player can see where the cards can be placed.

Just as with board games, a major problem is making good computer-driven opponents. However, card games are often not too hard to analyze.

First-Person 3-D Action Games

3-D action games with texture mapping are probably the hottest trend right now, especially among teenage boys. These games give you a first-person perspective and a very realistic feel due to smooth movement and the use of texture mapping. Wolfenstein 3D showed that it was possible, and many games have followed suit. On the Mac we have seen games like Pathways Into Darkness (see figure 1.9), Sensory Overload, Marathon, and of course the original Wolfenstein 3D.

However, 3-D first-person games sure didn't start with Wolfenstein 3D. Vector and polygon graphics games have been here for a long time, with titles such as Colony, BattleZone, and Spectre.

If you want to make 3-D games with texture mapping, you should probably forget about full-screen graphics on 68030 Macs and below. You will have to draw the game in a smaller window size (see the game Wolfenstein 3D: First Encounter on the CD-ROM for an example). Another possible solution for slow Macs is to include a way to turn texture mapping off and use flat polygon graphics instead. Is it worth the trouble? Make your graphics engine first, and then have a look if you can put that feature in for a low cost. If not, you aren't losing a lot.

Graphics

Ray-casting is one approach to this problem. On the ray-casted image you can then place sprites, preferably sprites that are rescaled on the fly.

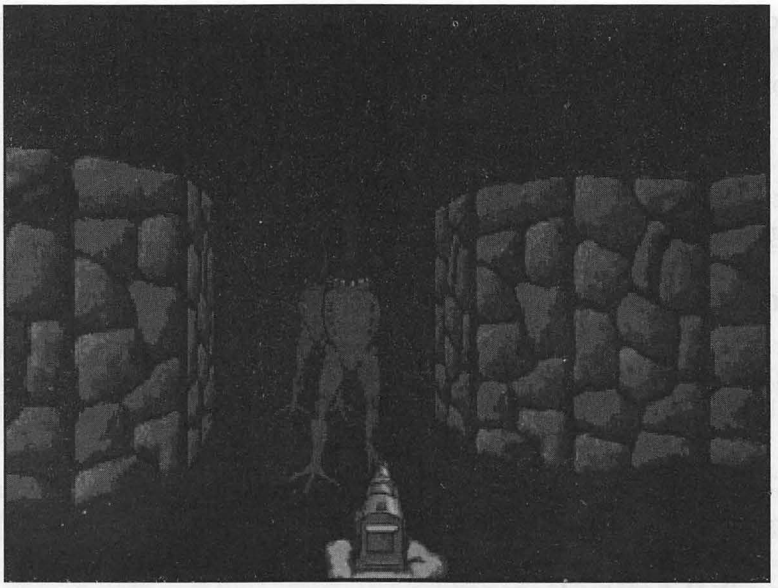


Figure 1.9 *Pathways Into Darkness, a 3-D game with texture mapping.*

Another approach is line or polygon graphics, as used by BattleZone and Spectre. This is not state of the art anymore. Consider it for a low-power option for extremely low-end Macs or for noncommercial projects for which you can't spend so much time optimizing.

Refer to chapter 9, "QuickDraw 3D," for the 68K.

Representation and Behavior

The world can be represented as a rectangular grid. That is the simplest solution, and it is what you find in Wolfenstein 3D. However, other representations can give a more flexible world, like the one you find in Marathon. Again, see chapter 9.

The creatures (the hero and monsters) and objects in the game, on the other hand, are straightforward to represent. You need a list of all active objects and creatures, plus their position and perhaps state (alive/dead and health). If the number of objects and creatures that can be active at one time is very large, you can add a mechanism for deactivating objects out of reach.

Adventure Games

An adventure game is generally not an action game, but rather a game with a story. In this type of game, you move a character through a world of adventure and exploration, overcoming obstacles by using your brain more than your reflexes.

The first well-known adventure game was Adventure and was made in the 70s. That was a text adventure. Many followed, most notably Zork and other Infocom games.

Nowadays, text adventures are no longer part of the commercial game market. They have been replaced by adventure games using graphics.

For a while, there were many Macintosh adventure games using mostly still images, with little or no animation. Déjà Vù was one excellent example, an adventure game that used point-and-click in a way that made the old text adventure seem very clumsy and tedious to play.

However, the big trend was animated adventures. King's Quest was the first, and there has been a steady stream of adventure games of this kind.

On the Mac, we have mostly seen ports in recent years (see figure 1.10). Unfortunately, this means that we get adventure games with poor graphics. The entire Sierra series has ugly, zoomed graphics. Doesn't this suggest that there may be room for some adventure games with high-resolution graphics?

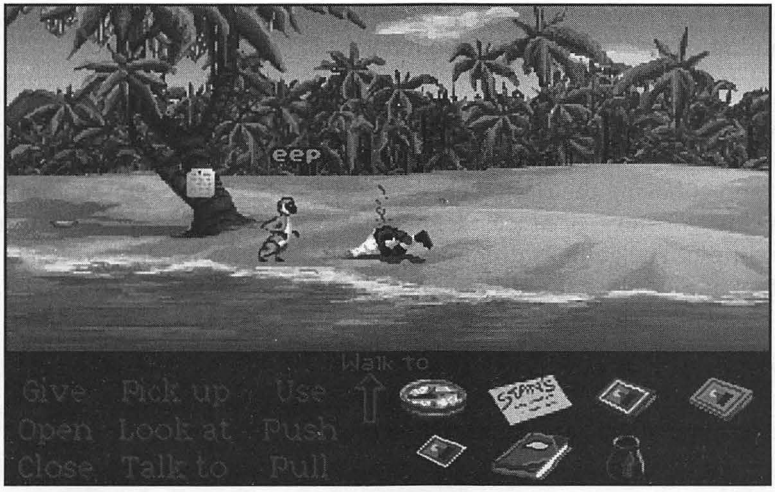


Figure 1.10 *Monkey Island, one of many low-res adventure games ported to the Mac—a good game with less-than-perfect graphics.*

Graphics

Adventure games come in many different shapes, and the graphics are totally different for each kind. For most kinds of adventure games, you don't have to worry about speed as much as you do in an action game.

A text adventure needs a text window. Simple, right? You can make a text adventure from the "vanilla" text shell that is part of almost every development system. In CodeWarrior, you have SIOUX.

An animated graphical adventure is like a platform game, but more. You should be able to point and click in the image, move around, and select, pick up, and manipulate objects. You may also want to consider the first-person 3-D approach for an adventure game.

Representation and Behavior

Adventure games have more need for smart characters than other games. In action games, the opponents can be simple finite-state machines, with a few behaviors for a few different situations. In an adventure game, there is no limit to how smart you make a character—and the players may still find it stupid.

An adventure game needs a fairly complex world representation. It must hold all information for where you can move from each place in the world, what objects there are in each room, and so on. See further chapter 7, "Game Mechanics: Environments and Entities."

Dungeon-Digging Games

By dungeon-digging games, I refer to games that are between shoot-'em-up games and adventure games. They have much less plot and storyline than an adventure game, but there is still more thinking to do than in, say, Wolfenstein 3D.

Dungeon-digging games are generally heavily inspired by role-playing games like Dungeons & Dragons, using combat systems similar to those and often the concept of gaining experience points through fighting. A big part of the games is combat, but combat that you win via tactics and choosing the right weapons and moves, rather than just reflexes. There usually is some kind of story, or at least a goal that should be accomplished, but the story is usually rather simple. The game is dominated by random encounters, in which you meet various monsters more or less at random—just as in a beginner Dungeons & Dragons scenario.

ASCII-based games such as Rouge and Hack (and later NetHack—see figure 1.11) were quite popular in the early 80s. Graphical games like Dungeon Of Doom, Ultima, Wizardry, Theldrow, and Realmz are fairly similar to them, but with graphics.

Graphics

Graphically, a dungeon-digging game can be anywhere on the scale, including simple top views on a grid, just like in Dungeon 1; smoothly animated top-view games using sprite animation; polygon 3-D graphics; and texture-mapped first-person 3-D games. Pick a level that suits you and the amount of time you have for the project.

Representation and Behavior

The typical dungeon-digging game uses a rectangular grid that you move around in. Some, such as Wizardry and Theldrow, give a first-person view, while others, such as NetHack, give a top view.

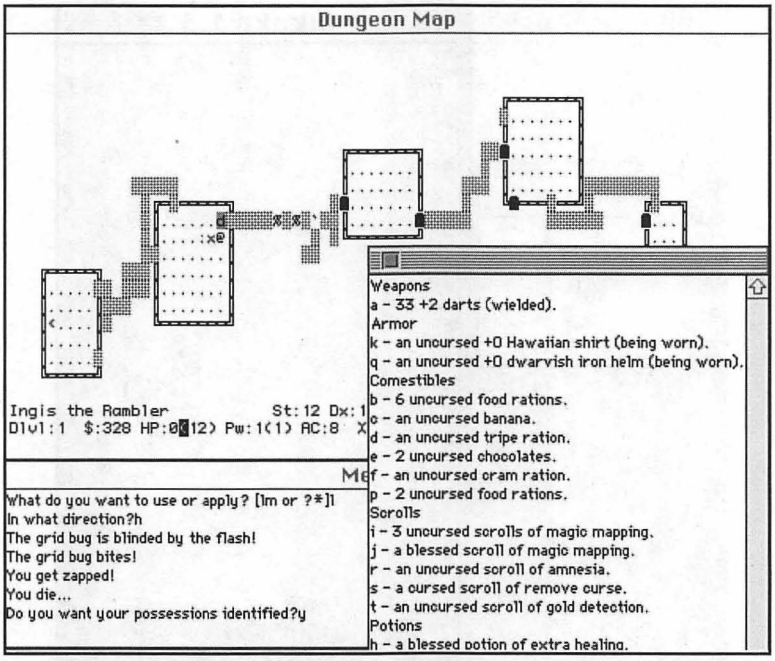


Figure 1.11 *NetHack, a game that, despite primitive character-based graphics, is perhaps the most complex dungeon-digging game around.*

In a dungeon-digging game, the combat system and the data associated with each monster and the player are of great importance. The combat system often mimics that of Dungeons & Dragons, with a probability to hit that depends on your skill, your weapon, and the opponent's dexterity and armor. There are also hit points that are reduced when you get hurt.

The creatures the player encounters may need to be smarter than the ones you find in first-person 3-D action games. There is more need for a plot, creatures that are not hostile, and some possibility of communication. State-based behavior may fit this kind of game well. See chapter 8, "Game Mechanics: Behavior."

Abstract Action Games

In the early 80s, many arcade games were highly abstract and didn't try to simulate anything at all, but rather focused on an interesting game mechanism. Such games include Snake, Breakout, and the later but extremely popular Tetris. These days we have a number of Tetris descendants (especially in the shareware domain—see figure 1.12) and games such as Spin Doctor and Diamonds.

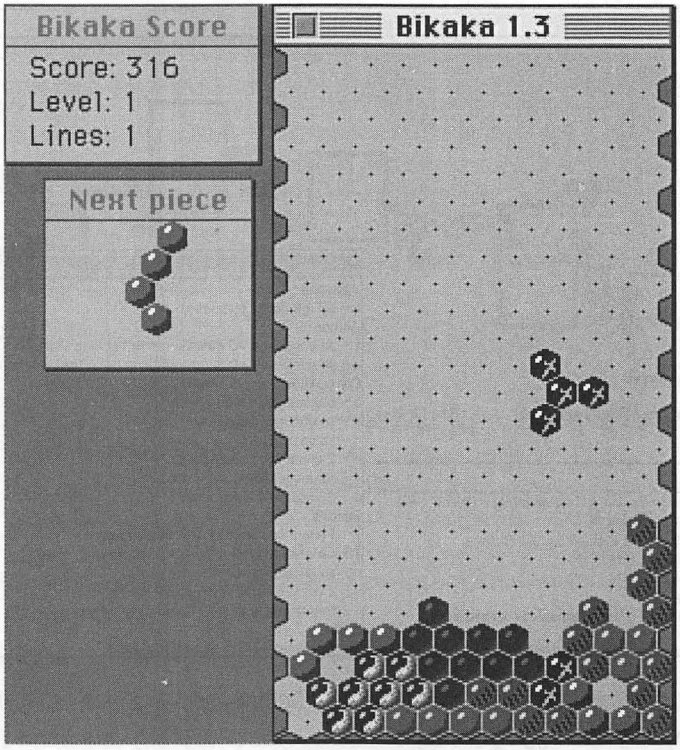


Figure 1.12 *Bikaka, my hexagonal version of Tetris.*

An abstract action game can be extremely simple, since there is no real-world original that it is simulating. It doesn't matter too much if objects look plastic or if movements are realistic or not, since there is no realism in it.

Graphics

The simpler abstract action games can be made with straightforward QuickDraw. They are often as easy to make as the simpler board games.

Representation and Behavior

Let's take Tetris as an example. You need a 2-D array representing the "well," telling which spaces are filled and which ones are free. That could be an array of Booleans, but you should probably make it an array of integers so you can remember what color or pattern each grid has.

Then, you need a representation for each piece that can fall into the well. In classic Tetris, each piece is built from four blocks. A possible representation is to use four points; each one indicates an offset from the center point. You need four sets of four points for each piece.

Simulation Games

Many games can be called simulation games, including flight simulators (which are really first-person 3-D action games), but with the game SimCity we got a completely new category of games: games in which you build something and try to make it work, and in which the result is rather complex and partially runs by itself. Maxis has made several more "sim" games, and games such as Railroad Tycoon also fall into this category.

Graphics

Sim games are perhaps the most complex sprite-based games you can find. They don't have to use any particular graphics, though. You could write a SimCity clone using some straightforward QuickDraw calls without any off-screen techniques, with a top view on a grid.

Representation and Behavior

Most simulation games can base the simulation on a world represented by a 2-D array. For example, imagine that you want to simulate a city, like SimCity does. For each grid space, you need to store several pieces of information, such as how that space is used (road, building, open, park, etc.), how many people live there, what the pollution levels and crime rates are, and so on.

The 2-D grid is easy to handle when making all the algorithms needed for a good simulation. You may want to spread good and bad effects (pollution, crime, wealth, general happiness) from each space to its neighbors, or follow roads, power lines, and water pipes to make sure they are connected. I'm not sure I would want to do that in a world in which all objects are separate objects in a huge list.

In the simulated world, there may or may not be individual beings. If there are, you must also take behavior into account. In SimCity, you can find trains, airplanes, and boats. The train follows its track and picks a random direction when the railroad forks. Boats seem to go around aimlessly, and so do the airplanes.

You could make simulations in which individuals, be they vehicles or living beings, appear smarter. However, you must limit yourself to the scope of the simulation. What good does it do to make the boats go on scheduled routes if it doesn't affect the rest of the simulation?

Strategy Games

Strategy games are next of kin to board games, and it is often difficult to tell the groups apart. Just to have a definition, let's say that a board game is not extremely complex, perhaps on level with Risk, and doesn't take more than an hour or two to play. A strategy game, however, is far more complex and takes lots of time to play—several hours and possibly as much as several days.

Empire was a classic strategy game seen on VAX computers in the early 80s. A fairly close Mac implementation is Strategic Conquest, a game that is still being worked on, and the shareware game Empire Master. A very popular newer game is Civilization (see figure 1.13).

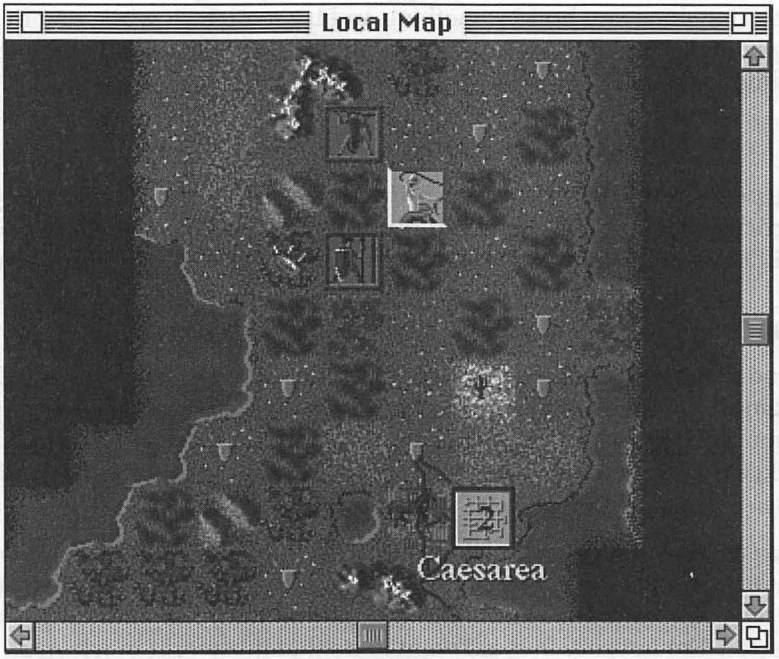


Figure 1.13 *The popular strategy game Civilization. I'm sure you've seen it before!*

Graphics

Strategy games, like board games, need no advanced techniques for graphics. What you learn in the first graphics chapter may be enough to make what you need.

Representation and Behavior

Most strategy games use a grid, just as board games. The pieces should usually be represented by a list of records, so each piece can have a fair amount of private information. Very simple strategy games, however, may use numbers in the grid telling how many armies (or whatever) you have in that space.

As with board games, making algorithms for enemy players is a major problem. It is often harder to analyze the position in a strategy game than in a board game. There are countless possible moves, and the map is often complex. The computer player must distribute its power to guard different parts of its territory and also make some kind of progress. As you know if you have played strategy games, the computer players often appear stupid.

Programmer's Games

Programmer's games make up a very special niche of computer gaming which is rather small but fascinating. In a programmer's game, you as a player write a program controlling something in the game, which should then perform better than the program of your competitor's.

These games are rather few in number, but some are really spectacular. A classic that is probably only of interest to real programmers is CoreWar. CoreWar simulates a computer in your computer, and you put programs against each other: The programs try to make the other program fail and stop. The game is extremely abstract, and useless for the non-programmer.

The most popular kind of programmer's game is robot battle games. You write a program that controls a robot, dictating what it should do when in the battlefield, and your robot tries to destroy other robots before they destroy it.

A very different, alas short-lived game, was ChipWits, published by BrainPower. ChipWits, which to my knowledge runs only on the oldest Macs, used a graphical programming language, which made it the most nonprogrammer-friendly programmer's game I've ever seen.

Graphics

You can use any kind of graphics that suits your game idea. The concept of a programmer's game doesn't imply any specific kind of graphics it uses—if any.

In one way or another, the player should enter a "program." If you let the player type in the program as text, you need a text editing window and should probably use the TextEdit package, which is in the ROMs of all Macs. However, if you use some graphical programming method, the game can be a lot more appealing to nonprogrammers.

Representation and Behavior

The only data structure of special interest here is the program being executed. If the player enters it as text, you need to have an interpreter that can decode it. You can choose to interpret the text as the game runs. In that case, the text is the representation. However, that will probably get pretty slow.

A more effective option is to compile it into statements in a compact format, for example with an 8-bit integer (char) denoting the instruction, and some arguments to it. The point is that you should need no translating from a more verbose format (like text) once the game is running.

If the player enters the program in some other way—graphically, for example, as in ChipWits—the problem of compiling disappears. The compiled format can then be created directly from the player's actions.

Dungeon 1 Plus

Before I end this chapter, let me take you on a second visit to the Dungeon game. I will add a little to it, making it a bit more to build on later. I will add one thing here, a feature that would have made it too big for a first demo, but that we can afford adding now—namely to get away from the hard-coded single level and generate maps randomly instead.

This improvement does not introduce any important programming techniques, just a fairly large routine in plain code, so this is not enough for a change to "Dungeon 2" yet—just a "Plus." The big changes will come later.

Map Generation

Dungeon 1 Plus uses a simple algorithm for generating the level layouts. It places a few rectangles, or "rooms," on the map, and fills them with "floor." Then it makes sure all rooms are connected. Some treasure and monsters are placed in the rooms, and finally the player and the exit are placed.

The algorithm doesn't check if rooms overlap, so they often do. This gives maps that sometimes have a few rooms with several corridors between them, and sometimes one irregularly shaped room.

Figure 1.14 shows how two rooms with a corridor between them relate to what you see in the game. Note that part of the corridor goes along the side of Room 2; the corridor adds to the room rather than looking like just a corridor between the rooms.

Source Code Additions

The following changes have been made to the source:

The routine `CreateLevel()` has been added. This is the biggest change. Its source code follows:

```
/* The level generation routine */

static void CreateLevel()
{
```

I use an array of rectangles, named `roomRect`, for storing the position of each room I create. Except for that, I just use a bunch of short integers for more or less temporary use.

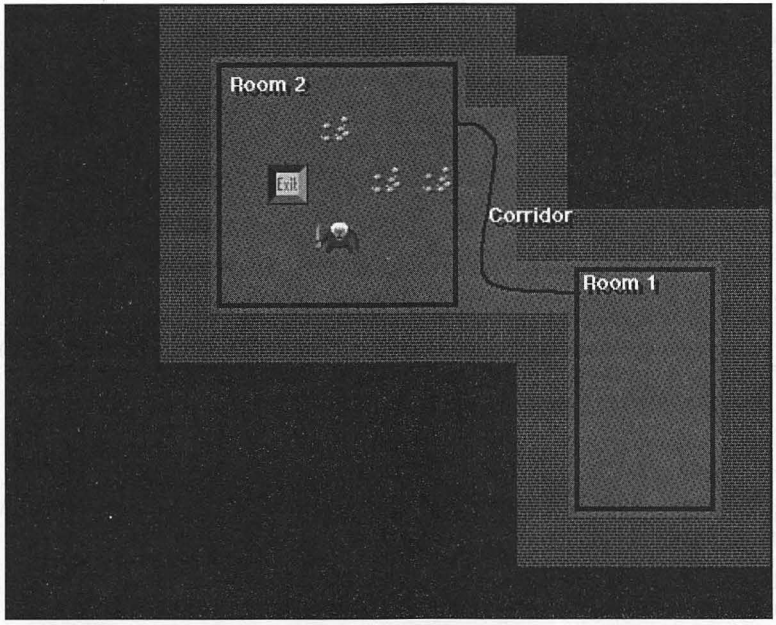


Figure 1.14 *Two rooms and a corridor in the map generator (black rectangles and black curve) and how they look in the game.*

```
Rect roomRect[10];
short room;
short height, width;
short h, v, h1, v1, h2, v2;
short numRooms;
short numTreasures, numMonsters;
short i;
```

First of all, we fill the entire dungeon with walls by assigning each item in `tileArray` to wall.

```
/*First fill the entire dungeon with walls!*/
    for ( h = 0 ; h < kArraySizeH ; h++)
      for ( v = 0 ; v < kArraySizeV ; v++)
           tileArray[h][v] = wall;
```

Then, I pick a number of rooms, anywhere from three to six.

```
numRooms = 3 + Rand(4); /*3 to 6 rooms*/
```

Each of the `numRooms` first entries in the `roomRect` array is now set to a random rectangle. I keep the rectangles fairly small. If I didn't, I would almost always end up with one big room, since the rooms would often overlap.

```
/*Create each room*/
     for ( room = 0 ; room <= numRooms - 1 ; room++)
       {
          height = 1 + Rand(5 - numRooms / 2);
          width = 1 + Rand(5 - numRooms / 2);
          roomRect[room].top = 1 + Rand(kArraySizeV - height - 2);
          roomRect[room].bottom = roomRect[room].top + height;
          roomRect[room].left = 1 + Rand(kArraySizeH - width - 2);
          roomRect[room].right = roomRect[room].left + width;
```

After picking the size, I set all the tiles in the room to empty.

```
          for ( h = roomRect[room].left ; h <= roomRect[room].right ; h++)
            for ( v = roomRect[room].top ; v <= roomRect[room].bottom ; v++)
                 tileArray[h][v] = empty;
       };
```

All rooms must be connected! For each room except the last one, I make a path to the next room in the array.

```
/*Make paths between all rooms*/
     for ( room = 0 ; room <= numRooms - 2 ; room++)
       {
```

First, I select a random position in each room to move from and to, `h1,v1` for the first room and `h2,v2` for the second one.

```
/*We make a path from h1,h2 to h2,v2*/
          h1 = roomRect[room].left + Rand(roomRect[room].right -roomRect[room].➥
left + 1);
          v1 = roomRect[room].top + Rand(roomRect[room].bottom - roomRect[room].➥
top + 1);
          h2 = roomRect[room + 1].left + Rand(roomRect[room + 1].right -➥
roomRect[room + 1].left + 1);
          v2 = roomRect[room + 1].top + Rand(roomRect[room + 1].bottom -➥
roomRect[room + 1].top + 1);
```

Then, I walk from `h1,v1` to `h2,v2`, first along the horizontal axis...

```
/*First move along the h axis*/
          if ( h1 < h2 )
            for ( h = h1 ; h <= h2 ; h++)
                 tileArray[h][v1] = empty;
          else
            for ( h = h1 ; h >= h2 ; h--)
                 tileArray[h][v1] = empty;
```

...and then along the vertical axis.

```
/*And then along the v axis*/
          if ( v1 < v2 )
            for ( v = v1 ; v <= v2 ; v++)
                 tileArray[h2][v] = empty;
          else
            for ( v = v1 ; v >= v2 ; v--)
                 tileArray[h2][v] = empty;
       };
```


For each room, I select a number of treasures, zero to two, and a number of monsters, zero to one, and put them in random places.

```
/*Now populate the rooms!*/
     for ( room = 1 ; room <= numRooms - 1 ; room++)
       {
          numTreasures = Rand(3);     /*0 to 2 treasures*/
          for ( i = 1 ; i <= numTreasures ; i++)
            {
              h = roomRect[room].left + Rand(roomRect[room].right - ➥
roomRect[room].left + 1);
              v = roomRect[room].top + Rand(roomRect[room].bottom - ➥
roomRect[room].top + 1);
              tileArray[h][v] = gold;
            };
          numMonsters = Rand(2);      /*0 to 1 monsters*/
          for ( i = 1 ; i <= numMonsters; i++)
            {
              h = roomRect[room].left + Rand(roomRect[room].right - ➥
roomRect[room].left + 1);
              v = roomRect[room].top + Rand(roomRect[room].bottom - ➥
roomRect[room].top + 1);
              tileArray[h][v] = enemy;
            };
       };
```

Now I put the player in the first room.

```
/*Player position:*/
     playerPosition.h = roomRect[0].left + Rand(roomRect[0].right - roomRect[0]. ➥
left + 1);
     playerPosition.v = roomRect[0].top + Rand(roomRect[0].bottom - roomRect[0]. ➥
top + 1);
     tileArray[playerPosition.h][playerPosition.v] = player;
```

Finally, I must put in an exit. This must not be the same as the player position.

```
/*Exit position:*/
     h = roomRect[numRooms - 1].left + Rand(roomRect[numRooms - 1].right - ➥
roomRect[numRooms - 1].left + 1);
```

```
     v = roomRect[numRooms - 1].top + Rand(roomRect[numRooms - 1].bottom - ➥
roomRect[numRooms - 1].top + 1);
```

If the exit position is not the same as the player position, everything is fine, but if it is, I must find some other place!

```
/*But please don't overwrite the player with the exit!*/
     if ( h == playerPosition.h )
       if ( v == playerPosition.v )
          {
/*Try another room:*/
            h = roomRect[numRooms - 2].left + Rand(roomRect[numRooms - 2].right - ➥
roomRect[numRooms - 2].left + 1);
            v = roomRect[numRooms - 2].top + Rand(roomRect[numRooms - 2].bottom - ➥
roomRect[numRooms - 2].top + 1);

/*Still failure? Darn. Just take a space next to the player.*/
            if ( h == playerPosition.h )
               if ( v == playerPosition.v )
                  if ( h < kArraySizeH )
                     h = h + 1;
                  else
                     h = h - 1;
          };
/*OK, that's enough. Set the exit!*/
     tileArray[h][v] = exitPos;
```

I set all tiles to be unknown...

```
/*All tiles are unknown when we start*/
     for ( h = 0 ; h < kArraySizeH ; h++)
       for ( v = 0 ; v < kArraySizeV ; v++)
          tileKnown[h][v] = false;
/*...except the ones around the player*/
```

...except the ones around the player!

```
/*Make the spaces around the player known*/
     ShowAroundPlayer();
```

Finally, I redraw all tiles.

```
/*Draw all tiles!*/
     for ( h = 0 ; h < kArraySizeH ; h++)
       for ( v = 0 ; v < kArraySizeV ; v++)
          DrawTile(h, v);
} /*CreateLevel()*/
```

The routine `NewGame()` has been added. It sets `playerHitPos` to its start value and calls `CreateLevel()`. It is not entirely necessary to have this in a separate routine yet, but it will be.

```
/* Set up for a new game */

static void NewGame()
{
/* Fill in the tileArr array */
     CreateLevel();

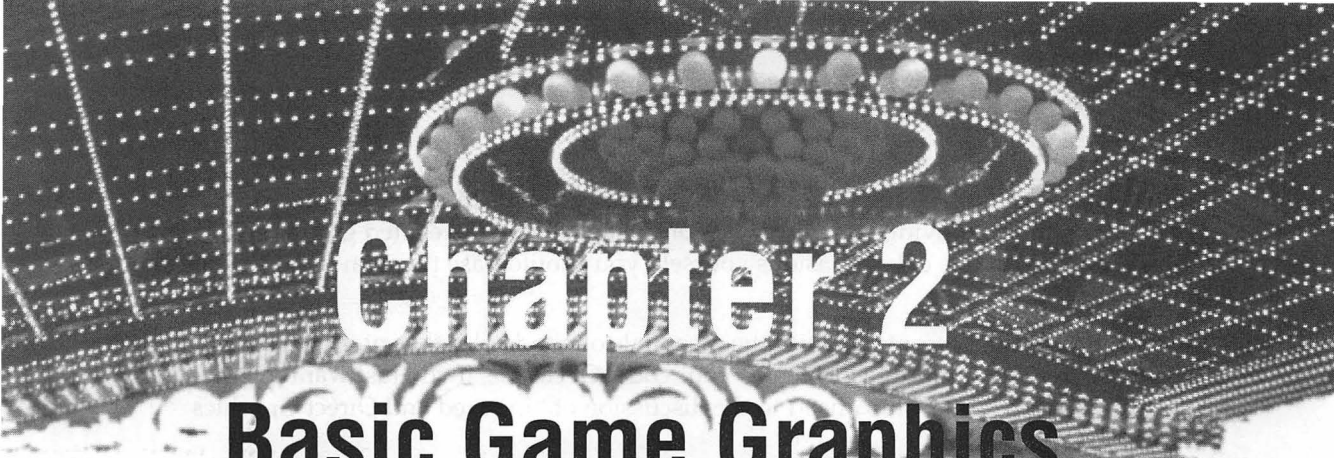
/* Start with a healthy player */
     playerHitPoints = 5;
}
```


When the player moves to the exit, the game does not quit, but generates a new level instead.

```
case exitPos:
   tileArray[playerPosition.h][playerPosition.v] = empty;
   tileArray[clickedTile.h][clickedTile.v] = player;
   DrawTile(playerPosition.h, playerPosition.v);
   DrawTile(clickedTile.h, clickedTile.v);
   playerPosition = clickedTile;
   if ( noErr != SndPlay(0L, (SndListHandle)GetNamedResource('snd ', "\pNext ➥
level"), false) )
   ;
   CreateLevel(); /* Don't quit -- make a new level instead */
   break;
```

Leaving the Game

In Dungeon 1, the game ends when the player dies or leaves the level. In Dungeon 1 Plus, it ends only when the player dies. There is currently no other way. A real game should have Quit in the File menu and respond to Command-Q. We will add that later in the book, when we deal with events, menus, and the other Macintosh niceties. If you are in a hurry to quit Dungeon 1 Plus in the meantime, move next to a troll and stand still (clicking on the hero) until he gets gobbled up—or hit Command-Option-Escape.



Chapter 2
Basic Game Graphics

Graphics and computer games are inextricably linked—think about it: When was the last time you saw a *text-based* adventure game make a top ten list *anywhere?* In order to create compelling games that keep your players coming back for more, you must have a thorough understanding of computer graphics.

This chapter addresses the fundamental issues of Macintosh graphics as they relate to computer games. Graphical solutions tend to be elegant and simple, but difficult to derive. This chapter will identify the most important factors for you and guide you to tried-and-true solutions.

You should already have a reasonable grasp of the basic Macintosh application architecture, as well as fundamentals such as opening windows and drawing shapes.

The first part of this chapter will dwell on color. From a theoretical standpoint as well as in practical implementation, color is an area of programming that is often poorly understood. Color on computers is a tricky area of programming, but certainly one you can master—especially since we supply the basics here, all geared toward fast, polished game graphics.

After color, we move on to other interesting areas of graphical programming as it relates to games. We will discuss finding the best available screen for your game, fading screens in and out gracefully, hiding the menu bar correctly, making `CopyBits()` *really* fast, handling multiple monitor setups, and many other interesting nuggets for you to rub your brain on.

Programs demonstrating important concepts and processes discussed in this chapter are included on the CD-ROM. These programs make certain bold assumptions about your system—such as assuming that your main screen is a 256-color-capable Indexed Device—and therefore may make poor skeletons for your next great game. However, by using techniques presented here, you can modify the programs to avoid these assumptions gracefully. Study the programs, play with them, experiment with them—just don't make Pong with them.

Color Theory

Color is a key element when creating mood, tension, and excitement in games. Though it's rarely covered in computing books, a good grasp of color theory is an essential arrow in the

quiver of any serious game programmer. If you have an art director, you can and should depend on him to provide you with well-conceived and well-executed graphics. However, when you must make color decisions yourself, you should base them on more than just an intuitive feel.

In this section, we present you with the essentials of good color theory, so that your graphics may be as refined and sophisticated as your code. Of course, if you just want to get to the technical stuff, you can skip ahead to the discussion of "Indexed and Direct Graphics Hardware."

How to Dress Better

It is unfortunate but true that many people would be very happy if they made Garanamals for adults: Just match the animals and—*voilà!*—your clothing matches perfectly every time. Clothes-matching will probably never be as simple as that for us hapless adults, but there are some areas of couture you can manage pretty easily with just a little bit of knowledge. A large part of the foundation of dressing well (as well as creating gorgeous graphics) is color theory.

First of all, let's forget RGB (Red, Green, and Blue) color space for now. If you've been dealing with color on computers at all, you know that RGBs are the primary colors in which your computer thinks. Unfortunately, the rest of the world thinks that Red, *Yellow*, and Blue are the primary colors—that's what they were taught in school, so that's the way it is. Of course, the rest of the world isn't entirely crazy: Red, Yellow, and Blue comprise the three primaries of the *subtractive* color model.

Computers, on the other hand, use a different color model: the *additive* model, in which Red, Green, and Blue are the primary colors. This color model best describes the way different colors of light blend—it's also the way our eyes see. While the additive model is practical (it's what the hardware understands best), it isn't what people are used to.

Subtractive and Additive Color

In color theory, the term subtractive *is used when a color seen is the result of reflecting white light off a surface. Surfaces can absorb (and thus subtract) some of the components of white light, so what is perceived as the surface's color is actually whatever colors weren't subtracted from the white light. Notice that as you mix in more and more colors, the result gets closer and closer to black. The primary subtractive colors are red, yellow, and blue.*

Additive *color describes what happens when you add various colors of light together. Note that as you add more and more colors together, you begin to approach white light (think of the rainbow going back* into *your sixth-grade science class prism). The primary additive colors are red, green, and blue.*

For purposes of comparison in this section, let's call the subtractive, Red-Yellow-Blue model, the one most people are familiar with, the *RYB* model. We'll call the additive, Red-Green-Blue model the *RGB* model.

Most designers use the RYB color model when defining palettes of harmonizing color. Various rules have been developed that govern the selection of pleasing combinations of colors. People have been using these rules for centuries, and most are very comfortable with the resulting color schemes. I strongly recommend that you use the RYB model and rules (arbitrary as they may seem) when designing graphics for your projects. If you don't, your graphics may wind up looking unsophisticated at best—and jeepers, if that's what your art looks like, what do *you* look like?!

Comparing RGB and RYB

RGB and RYB appear to be similar, but don't let that fool you (see figures 2.1 and 2.2—you'll also find color TIFF graphics of these figures in the 02 - Basic Game Graphics folder in the Book Chapters folder on the CD-ROM). In RYB, you go across the color wheel to get to a complementary color. For instance, green would be the complement of red, and that's just peachy with everyone. Try that trick with Apple's RGB-style Color Picker and you'll get cyan; the Artists Guild will revoke your beanie if you try to pass cyan off as the complement of red.

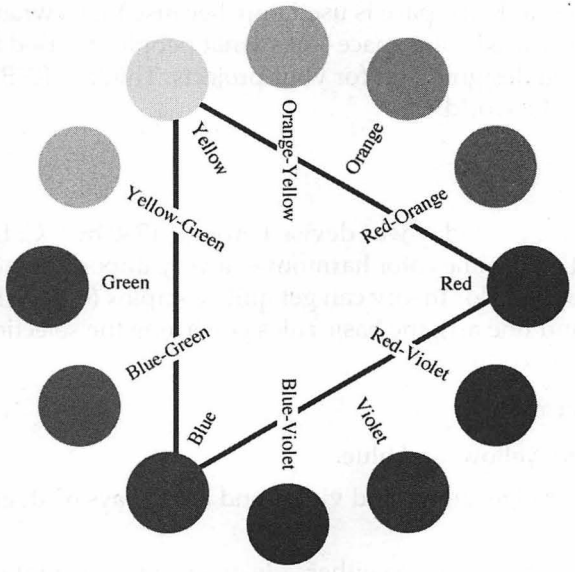


Figure 2.1 *Classic, 12-part RYB color wheel.*

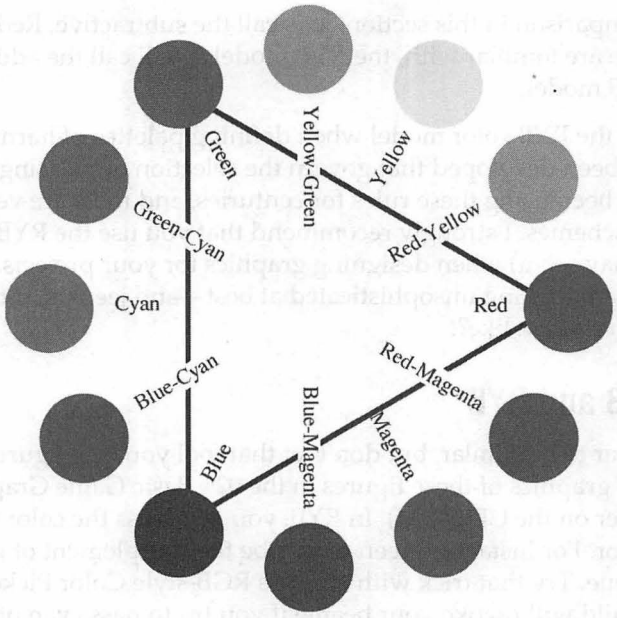


Figure 2.2 *RGB color wheel.*

Again, let's make it clear that RGB space is used only because that's what the hardware uses. RYB is the most commonly used color space—it's what people are used to seeing, and that's what you should use when designing art for your projects. Think in RYB, express in RGB, and all will be right with the world.

The Color Wheel

The 12-part color wheel we use today was devised around 1730 by J. C. Le Blon. It is the tool used to select colors and determine color harmony—a very important factor in creating polished graphics. Although color theory can get quite complex (especially in the rarefied worlds of high fashion and fine art), the basic rules governing the selection of harmonious colors are relatively easy to codify.

First, let's define some terms:

- *Primary colors* are red, yellow, and blue.
- *Secondary colors* are orange, green, and violet, and are always 60 degrees away from one of the primaries.
- *Adjacent colors* are the two colors on either side of a particular color on the color wheel. For instance, the adjacent colors for RYB green are yellow-green and blue-green.

- *Complementary (opposite) colors* exist on the opposite sides of the color wheel. If you have a red, you will find its complement (green, in the case of RYB) on the exact opposite side of the color wheel.
- *Split complements* are the adjacent colors of a complementary color. For instance, blue's split complements are red-orange and yellow-orange.

Let's look at the attributes associated with these color relationships:

- Complementary colors usually pair a warm color with a cool one. Together they are unsubtle, lively, vibrant, and pleasing, without appearing frenzied or demented. The red-green color scheme of Christmas is a classic complementary pairing, as is an orange sunset against a deep blue ocean. Use of secondary colors with complements (red-orange, for example, with blue-green) is thought to be a little more elegant than primary complementary mixes.
- Adjacent colors normally favor either the warm or cool sets of colors and seem somehow emotional: Sunsets are comprised of adjacent colors, from red into orange, from yellow into blue. Highlights and shadows are normally comprised of adjacents mixed with white or black.
- Split complements provide some harmonies that are a little more advanced. There is more variety when using split complements and, with judicious application, that can mean more interesting graphics.

Using Color Wisely

If you look at the games out in the marketplace, you will notice that the most popular ones use color effectively. Look at the use of color in, say, Brøderbund's Myst and Living Books' Just Grandma and Me. Myst, an adult title, uses color subtly to convey a moody atmosphere, while Just Grandma and Me, a children's title, uses a bright and happy color palette which kids find very appealing.

The makers of those and other popular games did not select colors haphazardly. Art directors get paid a lot of money to come up with useful color palettes. They analyze story, setting, time of day, character, mood, hardware limitations, and many other factors to derive a palette that works.

If you need to do some color scheming, here are a few general guidelines:

- Younger audiences like brighter colors, while older audiences can appreciate more subtlety.
- Avoid using reds and greens that have low saturation and low brightness qualities for important graphics. The most common form of color perception deficiency is red-green, and those individuals will have a hard time seeing your hard work.

- Characters that are seen as a group should have similar colors. Costumers use this technique to differentiate actors or groups of actors from the unimportant background and from each other. (For example, the Jets and the Sharks in *West Side Story* were given very different color clothing, not only because they were different gangs, but also so the audience could readily identify them, even during a rumble.)
- Manipulation of color saturation and value are classic depth-cueing devices, helping to differentiate foreground and background elements. You can use bright, highly saturated colors for foreground elements, and dim, washed-out colors in the background.
- There is no such thing as a flat black shadow. Use absolute black if you must, but try to use a darkened adjacent color if you can. For instance, if the shadow falls on baby-blue wallpaper, make the shadow a dark blue-green.
- Although white "pops" well as a highlight color, overuse can decrease its effectiveness. Try using lightened adjacents instead of pure white.
- If you're doing a cross-platform game for Macs and PCs and intend to use the same art on both machines, you may be in for a surprise: Often the art has to be tuned for the gamma of each platform.

Note

In this context, gamma *refers to the way colors actually show up on a computer's monitor. Gamma varies somewhat from computer to computer and varies quite a bit from platform to platform. For instance, a full-on blue looks roughly similar in color on different Macintoshes, but looks substantially different (usually darker) on PCs. Bear this in mind if you're creating graphics in a multiple-platform situation; avoid depending on only one set of art for the platforms—you may wind up with several.*

HSV and the Color Picker

The RGB color model is practical for the computer, but not particularly appropriate from an aesthetic point of view. The subtractive, RYB model is aesthetically useful, but cumbersome to implement in hardware. There is a third option: the HSV (Hue, Saturation, and Value [also called Brightness]) color model (see figure 2.3).

- *Hue* refers to any of the colors in the rainbow. They are given on the face of the hexcone and, starting at red and moving counterclockwise, proceed around the perimeter in rainbow order. Roy G. Biv is a famous fictional nonentity whose name can be used to remember the components of the rainbow. Each letter of his name is a color: Red, Orange, Yellow, Green, Blue, Indigo, and Violet.
- *Saturation* is the amount of hue in a given color. Saturation is at full intensity at the perimeter and drops off as you move toward the center. More saturated colors are more lush, while less saturated colors tend to be more pastel.

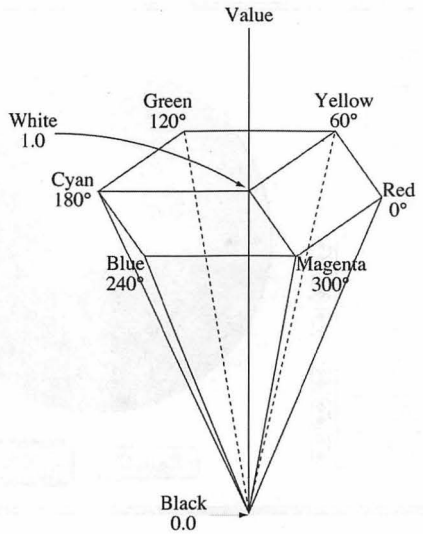


Figure 2.3 *HSV hexcone diagram.*

- *Value* measures the amount of whiteness in a given color. A lower value means a darker color, a higher value means a lighter color. (Value is also often called Brightness.)

Apple's standard Color Picker uses the HSV color model. The entire visible spectrum is presented around its color wheel, and a scroll bar controls the Value parameter (see figure 2.4—you'll also find a color TIFF graphic of this figure in the 02 - Basic Game Graphics folder in the Book Chapters folder on the CD-ROM).

Apple gives us a simple call to put up the familiar Color Picker dialog:

```
Boolean GetColor(Point where, ConstStr255Param prompt, const RGBColor *inColor, ➥
RGBColor *outColor);
```

`where` can be either the top left corner of the dialog box or one of two special values: `0,0` positions the dialog nicely on the main screen, while `-1,-1` displays the dialog on what the Color Picker determines to be the best screen, optimizing for depth and color.

`prompt` is a Pascal-style string and is displayed in the dialog's upper left corner.

`inColor` is a pointer to the starting color which is displayed for comparison.

`outColor` is a pointer to the last color value the user selected before clicking on the OK button.

This call returns a Boolean value: TRUE if the user clicked on the OK button, FALSE if the user canceled.

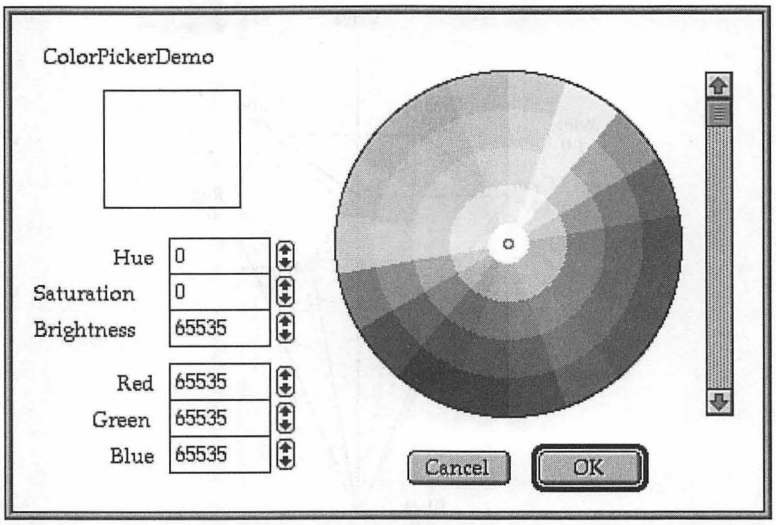


Figure 2.4 *Apple's standard Color Picker dialog.*

The following little program demonstrates how simple this call is:

```
//Simple Color Picker Demo
#include <Picker.h>    //for GetColor() function
//
// main
//
// For this simple program, this is where the action is. All
// this will do is throw up the Color Picker dialog and exit.
//
void main( void )
{
Point       where;
RGBColor    originalColor = {65535, 65535, 65535};  //absolute white
RGBColor    newColor;

  //standard initialization
  InitGraf(&qd.thePort);
  InitFonts();
  InitWindows();
  InitMenus();
  TEInit();
  InitDialogs(NULL);
  InitCursor();
  FlushEvents(everyEvent, 0);

  where.h = where.v = -1;  //Position of -1,-1 tells Color Picker logic
                           //to center Picker dialog on best screen.
```

```
  if (GetColor(where, "\pSimple Color Picker Demo", &originalColor, &newColor) )
  {
      //a real program would do something with newColor!
  }
}//main
```

This demo program doesn't do much, but it does illustrate that getting a color from a user is pretty easy, and it lets you examine Apple's HSV color wheel.

Included on the CD-ROM is a sample program called ColorPickerDemo, which is a little more useful than the previous code. You'll find it in folder 02 - Basic Game Graphics in the Book Chapters folder of the CD-ROM. ColorPickerDemo also pops up the Color Picker dialog, but when you click on OK it takes the resulting RGB value, converts it to a string, and puts it on the Clipboard. Quite simple, really, but pretty handy when you need to know the exact RGB values of a color.

You can exploit the fact that HSV's hues are in the familiar rainbow order. HSV makes it simple to create a highly saturated, rainbow-hued palette, for instance.

Also on the CD-ROM are a couple of programs that demonstrate HSV colors: HSV Demo 1 and HSV Demo 2. They show how to draw in HSV color space as well as use one of the System palettes and create a color table from scratch.

The structure of an HSV color is:

```
typedef struct {
  unsigned short    hue;         //Fraction of circle, red at 0
  unsigned short    saturation;  //0-65535, 0 for gray, 65535 for pure color
  unsigned short    value;       //0-65535, 0 for black, 65535 for max intensity
} HSVColor ;
```

RGBs are defined on the Macintosh thus:

```
typedef struct {
  unsigned short    red;         //magnitude of red component
  unsigned short    green;       //magnitude of green component
  unsigned short    blue;        //magnitude of blue component
}RGBColor, *RGBColorPtr, **RGBColorHdl;
```

The Macintosh provides several Toolbox routines for converting between various color spaces. Two of the most useful are:

- `void HSV2RGB(const HSVColor *hColor, RGBColor *rColor)`
- `void RGB2HSV(const RGBColor *rColor, HSVColor *hColor)`

Indexed and Direct Graphics Hardware

Why do we use RGB color? Because RGBs are what the hardware uses: Three electron guns shoot out streams of electrons exciting red, green, and blue dots on the screen. Video cards inside the computer drive the video hardware, and they use RGBs too, though there are a few different flavors available. Video cards on the Macintosh may be grouped into three main categories:

- *Fixed devices* may be able to use color or grayscale, but the colors the hardware can show cannot be changed. (In this context, "Fixed" means "Constant.")
- *Indexed devices* limit the number of distinct colors that can be displayed at one time (typically 256) in order to work with limited video RAM. Pixmaps, instead of storing a 48-bit RGB for each and every pixel, simply store an index into a color table. (Note that though `RGBColor` structures occupy 48 bits, the Macintosh currently uses only the high 8 bits of the Red, Green, and Blue components, providing 24 bits of color information.)
- *Direct devices* have a one-to-one match between an RGB value in a pixmap and what they display—these are often called true-color devices.

Direct Devices

Direct devices are normally 16- or 24-bit video cards, usually purchased with high-end graphics capability in mind. Often, these cards come with built-in hardware acceleration in order to deal with the large amounts of data in true-color images. Most recent Macs come with 16- or 24-bit video circuitry built in.

These devices are fantastic for editing images but they are, unfortunately, too slow to handle game graphics (even the accelerated ones don't allow you to really push the pixels fast enough). However, we can tell them to behave like Indexed devices which, even with their limited number of available colors, can produce excellent graphics and are sufficiently speedy for games. We'll show you how to do that later.

Fixed Devices

Once, it was possible for games to ignore the existence of Fixed devices, which were used to describe devices such as printers, but with the advent of Fixed color on some Macintosh PowerBooks, you should be aware of them and deal with them properly. We'll go over this in more detail when we talk about `CopyBits()` and `ctSeeds`.

Indexed Devices and Color Tables

For each pixel described by an Indexed device, there is a simple index into a table of colors. A video card's table is limited by video RAM, but typically holds 256 RGB values. These tables are called CLUTs, or *Color Look-Up Tables*. Color tables show up elsewhere, too, so to avoid confusion we'll call a video card's CLUT a *hardware CLUT*.

While the vast majority of hardware color tables can address 256 different colors, there are many cards that address sixteen, four, or two different colors. Or, using the Monitors control panel, you can change the index depth of a particular monitor.

Note that color table colors are full, 48-bit RGB colors—the limitation imposed by Indexed devices is the number of colors that may be shown *simultaneously*, not the accuracy with which the color is defined.

With color tables, we can define graphics that are comprised of tokens pointing to color table colors. We can specify any of those colors with only a 1 byte index. That's a *six-to-one savings* over Direct devices! Since we can blast indexes representing pixels over to the video hardware so much quicker than we can RGB values, is there any wonder that 99 percent of all games use Indexed colors?

The following is the definition of a `ColorSpec`, an element of the CLUT structure:

```
typedef  struct {
  short   value;          //index or other value (reserved
                          //when used in GDevices)
  RGBColor  rgb;          //true color
}ColorSpec ;
typedef ColorSpec CSpecArray[1];
```

Here is what a CLUT looks like:

```
typedef struct ColorTable {
  long    ctSeed;         //unique identifier for table
  short     ctFlags;      //flags describing the CSpecArray
                          //high bit: 0 = belongs to pixmap; 1 = belongs to GDevice
  short     ctSize;       //number of entries in ColorTable minus 1
  CSpecArray   ctTable;   //array contains ctSize plus one ColorSpec structs
}ColorTable, *CTabPtr, **CTabHandle;  //note that 'clut' resources are exactly
                                      //described by the ColorTable struct
```

Though this data structure may appear daunting, it is really nothing more than a list of RGBs and a couple of other interesting items:

`ctSeed` is an identifier for a given `ColorTable`. Should any two pixmaps have identical `ctSeed` values, QuickDraw will consider them equivalent and skip time-consuming color conversions, something we will explain and exploit later.

`ctFlags` describe the `ctTable`. You can determine if a CLUT belongs to a gDevice or simply a pixmap by examining the high bit of this field.

`ctSize` enumerates the number of elements of the `ctTable`, minus one (for instance, for a 256-entry CLUT, `ctSize`'s value would be 255).

`ctTable` is an array of `ColorSpec` records. The array must contain `ctSize` + 1 `ColorSpecs`.

There are several functions provided by Apple for dealing with CLUTs:

- `CTabHandle GetCTable(short ctID)`—Given the ID number of a 'clut' resource, this will return a handle to a CLUT initialized with the data from the resource. Returns NULL if the resource could not be read. Note that your 'clut' resources must be numbered above 128 and below 1,024; all other IDs have been reserved.
- `void DisposeCTable(CTabHandle cTable)`—Properly releases memory occupied by a color table.
- `void CTabChanged(CTabHandle ctHdl)`—Gets a new `ctSeed` for given color table and notifies QuickDraw that a change has occurred.
- `void CTab2Palette(CTabHandle srcCTab, PaletteHandle dstPalette, short ➥ srcUsage, short srcTolerance)`—This function copies values from a color table to a palette. Palette will be sized to `ctSize` if it isn't the correct size to begin with. Any animating entries in `dstPalette` are relinquished before the copy occurs; you may have to reset animating colors with `AnimateEntry()` and `AnimatePalette()`. The function does nothing if either the source or destination is NULL.
- `void Palette2CTab(PaletteHandle srcPalette, CTabHandle dstCTab)`—Copies colors from an existing palette into an existing color table, resizing the color table if necessary. If either the source or destination is NULL, `Palette2CTab` does nothing.

Color tables are generally stored in resources of type 'clut'. One might think that the `GetCTable()` function simply goes out to the resource file and grabs it, but it is not quite that simple. `GetCTable()` allocates space big enough for the 'clut' resource specified and initializes that space with values from the resource. What does this really mean? Well, for one, always make your 'clut' resources purgeable, and for another, *always* use `GetCTable()` when getting 'clut' resources—Apple isn't just being nice and providing you with another flavor of `GetResource()`; this is the interface for getting 'clut' resources and is meant to be the only way to get a 'clut'. Additional functionality is provided by this call when, for instance, you ask for a default color table.

IDs for default color table resources for grayscale devices can be found by adding 32 to the desired pixel depth (see table 2.1).

Table 2.1
Default Grayscale CLUT IDs

Pixel Depth	ColorTable ID	Colors
1	33	2 colors: black and white
2	34	4 colors: black, 1/3 gray, 2/3 gray, white
4	36	16 colors: black, 14 shades of gray, white
8	40	256 colors: black, 254 shades of gray, white

Likewise, IDs for default color tables for Indexed devices can be found by adding 64 to the desired pixel depth (see table 2.2).

Table 2.2
Default Color CLUT IDs

Pixel Depth	ColorTable ID	Colors
2	66	2 colors: black, 50 percent gray, highlight color, white
4	68	16 colors: black, 14 colors including highlight color, white
8	72	256 colors: black, 254 colors including highlight color, white

Color tables are used extensively by Color QuickDraw, and we will use them quite a bit as we develop graphical techniques.

Palettes and the Palette Manager

Palettes and color tables are similar in that they both contain lists of colors. However, while color tables are lists of colors with some general usage information, palettes are more sophisticated. Simply put, palettes are *requests for colors* for a window—at the OS-level, the Palette Manager arbitrates requests for colors of all the visible windows (and the desktop), ensuring that all clients get the colors they need or the closest available approximation. This is not an easy job, but the Palette Manager does surprisingly well, especially when everyone cooperates and avoids hogging all the colors. We should note that the Palette Manager is superfluous on Direct devices; since all color requests will be fulfilled, there is no need to arbitrate among available resources.

When a window comes to the fore, the Palette Manager looks at its associated palette and compares what it is asking for to the current color environment. If the palette is requesting colors that already exist (or are within a specified tolerance), the Palette Manager avoids making any changes to the color environment. If there are colors the palette specifies that aren't available, the Palette Manager starts taking colors away from other windows and applications until all requests by the frontmost window are fulfilled. This can cause a color update for the other windows, which will not have all of their colors available and therefore will not look as nice. Oh well.

The Palette Manager's job is difficult, but it hides a lot of nasty situations from applications: multiple screens, multiple resolutions, various levels of exactitude for colors for any number of windows, and so on. Although many programmers loathe the way color is implemented on the Macintosh *vis-à-vis* the Palette Manager, it has grown over time into a powerful and elegant solution to an ugly situation. Imagine if there were no Palette Manager and every application had to handle color itself—yuck!

Here are the important data structures used by the Palette Manager:

```
typedef struct {
   RGBColor     ciRGB;          //true RGB values
   short      ciUsage;          //color's usage
   short      ciTolerance;      //tolerance value
   short      ciDataFields[3]; //private fields
} ColorInfo ;

typedef struct Palette {
   short      pmEntries;        //number of entries in pmTable
   short      pmDataFields[7]; //private fields
   ColorInfo    pmInfo[];       //array of ColorInfo records
} Palette *PalettePtr, **PaletteHandle;
```

As you can see, a palette is quite similar to a color table—there's a simple header followed by an array of color definitions. Here, however, instead of an array of simple `ColorSpec` records, we have an array of slightly more complex `ColorInfo` records.

Notice that `ColorInfo` records contain an RGB, a usage describing the color's properties, and a tolerance telling the Palette Manager how exacting it must be in matching this color. Since a palette is essentially an array of `ColorInfos`, you can see that each and every RGB in a palette has usage flags as well as a tolerance. This is how we tell the Palette Manager how to treat a window's colors.

Warning

You should never access the values of palettes directly (except for reading `pmEntries`). Always use Palette Manager calls (`SetEntryColor()`, for instance) when setting up or modifying palettes.

These are values that can be combined to form usage information in the `ciUsage` field of a `ColorInfo` record:

```
enum  {
   pmCourteous     = 0,       //this color has no special properties
   pmTolerant      = 0x0002, //use ciRGB if ciTolerance is exceeded by best match.
   pmAnimated      = 0x0004, //reserve an index on each device touched and render ➥
ciRGB.
   pmExplicit      = 0x0008, //no reserve, no render, no record; stuff index into ➥
port.
   pmWhite         = 0x0010, //in B&W port, draw this color as white
```

```
    pmBlack        = 0x0020, //in B&W port, draw this color as black
    pmInhibitG2    = 0x0100, //don't use this color on 2-bit grayscale devices
    pmInhibitC2    = 0x0200, //don't use this color on 2-bit color devices
    pmInhibitG4    = 0x0400, //don't use this color on 4-bit grayscale devices
    pmInhibitC4    = 0x0800, //don't use this color on 4-bit color devices
    pmInhibitG8    = 0x1000, //don't use this color on 8-bit grayscale devices
    pmInhibitC8    = 0x2000, //don't use this color on 8-bit color devices
      //Update  Constants used in NSetPalette
    pmNoUpdates    = 0x8000, //no updates
    pmBkUpdates    = 0xA000, //background updates only
    pmFgUpdates    = 0xC000, //foreground updates only
    pmAllUpdates   = 0xE000  //all updates
};
```


The first six entries in the above enumeration are of special importance to the Palette Manager; they describe categories of usage a color may belong to. You can use these when you create a palette to control the way the Palette Manager allocates colors to your window.

- `pmCourteous` colors are the most cooperative color type; they accept whatever value the Palette Manager can give them. On Direct devices, they always render as specified. Courteous colors aren't particularly useful.
- `pmTolerant` colors specify a tolerance threshold defining a range for color matching. If no available color falls within that range, the Palette Manager loads the required color. Using a value of about `0x5000` for `ciTolerance` will usually get you a color pretty close to the one you're requesting without causing undue color environment updating. Note that `pmTolerant` with a value of zero means, "Give me these exact colors," and that's what the vast majority of games want.
- `pmAnimated` colors are used for changing a color on the hardware CLUT for color table animation. These colors lock out all attempts to match to them, so use them wisely. Animated colors cannot be used on Direct devices. If your window spans an Indexed and Direct device, the Palette Manager is smart enough to animate only that portion on the Indexed screen. Once colors are set to `pmAnimated`, they cannot be used by `RGBForeColor` or by objects copied to the screen, such as PICTs or color icons. `pmAnimated` colors can be used for drawing by calls to `PmForeColor()` or `PmBackColor()`. Calls to `AnimateEntry()` and `AnimatePalette()` actually do the color table animation.
- `pmExplicit` colors specify an index value and always generate the corresponding RGB in the device's hardware CLUT. Explicit colors are useful if you want to display the exact contents of a CLUT or want to be sure what index a color occupies. Explicit colors can be used to give you an indexed interface to color usage.

The `pmBlack`, `pmWhite`, and `pmInhibited` modifiers allow you to specify sets of colors to be used at certain monitor depths. This is not often done for game programming due to speed considerations. When game programmers are concerned about having good-looking graphics at different depths, they will create different graphics sets with different bit depths. Lower color-depth graphics have the additional advantage of lower RAM requirements.

Many of the modifiers listed can be combined by adding their values together. For instance, you can define a particular palette entry to be `pmExplicit + pmAnimated`, which means that the window gets the color you've specified at the index you want across all devices the window touches, but windows that don't share that palette can't use that index. Furthermore, the entry can be animated by a call to `AnimateEntry()`.

Note

We should point out that games are notorious for not sharing their colors nicely with other applications. Paint programs suffer from this too, and for good reason: Both are graphically oriented and absolutely demand that their colors are available. Luckily, we can use the Palette Manager to ensure that whatever colors we require are there for us by setting the usage of the colors of a window's palette to `pmExplicit+pmTolerant`, with a tolerance of zero. This mode is just like `pmTolerant`, but it guarantees that the exact palette colors are moved to the hardware CLUTs of all devices the window touches when that palette is activated. Furthermore, the colors appear in the same order that they appear in the palette.

Using Palettes

The System will provide a default palette for any window that doesn't have one of its own (the same goes for dialog boxes). The default palettes provided by the System are good for only the simplest of tasks. In general, you want to assign your own customized palettes to your windows and dialogs.

The easiest way to assign a particular palette to a window is to make a 'WIND' resource and a 'pltt' resource both with the same resource ID. Then use the `GetNewCWindow()` call to create your window. The Window manager will load the window's description via the 'WIND' resource, and will cause the 'pltt' to be assigned to the window.

Note

You can also define a default palette for your application by creating a 'pltt' resource with an ID of zero. This is useful if all of your windows will be using the same palette. Then, whenever a window or dialog is opened without a 'pltt' resource matching its ID, your default palette will be used. (This is a nifty trick to use when upgrading precolor Mac programs.)

There is another, more dynamic way to create a default palette for your application. Instead of assigning a palette to a particular window, you assign it to the application as a whole by passing a `WindowPtr` with a value of -1: `SetPalette((WindowPtr)-1L, appPalette, kFalse)` designates `appPalette` as the default palette for any windows or dialogs that are not explicitly assigned their own. The effect is the same as having booted your program with the pallet as a 'pltt' resource with an ID of zero.

You will probably be provided 'clut' and/or 'pltt' resources by your artist or art director. If you're doing your own artwork, you must generate one or the other yourself. (You can also

use Palette Manager routines to invent palettes at run-time, but situations requiring this are rare; normally, color information is created by the art team from the original art and stored as resources.)

Some of the most useful Palette Manager traps are:

```
PaletteHandle NewPalette(short entries, CTabHandle srcColors, short srcUsage, ➥
short srcTolerance);
```

This function allows you to create a new palette from within an application. Space for a palette long enough for `entries`-number of colors is allocated, and that many colors are copied into the new palette from the `srcColors` color table. Usage for all entries in the new palette is set to `srcUsage`, and all entries' tolerances are set to `srcTolerance`. If `srcColors` is NULL, all entries in the palette are set to 100 percent black.

```
PaletteHandle GetNewPalette(short PaletteID);
```

This is the preferred method for getting palettes. Given the ID of a 'pltt' resource, this will create a palette from information supplied by the 'pltt' resource.

```
PaletteHandle GetPalette(WindowPtr srcWindow);
```

This returns a `PaletteHandle` pointing to a window's palette. It will return NULL if the window is not a color window or has no palette. It's easy to forget that while `GetNewPalette()` creates a new palette, `GetPalette()` simply points you to an existing one.

```
void DisposePalette(PaletteHandle srcPalette);
```

`DisposePalette()` properly disposes of a palette. Entries set to `pmAnimate` are properly relinquished before the palette is disposed of.

```
void CTab2Palette(CTabHandle srcCTab, PaletteHandle dstPalette, short srcUsage, ➥
short srcTolerance);
```

This useful function copies an entire color table into a palette. The palette will be resized if it is larger or smaller than the color table. Entries set to `pmAnimate` are relinquished before the copy takes place. The usage and tolerance for all entries are set to `srcUsage` and `srcTolerance`. If either `srcCTab` or `dstPalette` is NULL, `CTab2Palette()` does nothing. Changes to a palette by `CTab2Palette()` don't take effect until the next `ActivatePalette()` call. Note that there is an inverse call: `Palette2CTab()`.

```
void ActivatePalette(WindowPtr srcWindow);
```

`ActivatePalette()` ensures that the color environment for `srcWindow` is suitable. If not, it forces a change in the color environment and sends out update events for all affected windows. This is called automatically any time a window's status changes: when it opens, closes, moves, or becomes frontmost. You should call `ActivatePalette()` when you change a window's palette (like calling `SetEntryColor()`) since those changes won't take effect until the next call the `ActivatePalette()`. This does allow you to make a series of changes to a window's palette and cause only a single update event. Note that calling `ActivatePalette()` with a pointer to an off-screen port has no effect.

```
void SetPalette(WindowPtr dstWindow, PaletteHandle srcPalette, Boolean cUpdates);
```

This function changes which palette is associated with a window, and allows you to specify whether the window should receive update events whenever the color environment changes. When `SetPalette()` is called, the palette for `dstWindow` will be changed to `srcPalette`, and, if `cUpdates` is TRUE, then the window will receive update events on color environment changes.

```
void NSetPalette(WindowPtr dstWindow, PaletteHandle srcPalette, short nCUpdates);
```

`NSetPalette()` (the "N" stands for "New") is virtually identical to `SetPalette()`, but instead of a Boolean specifying whether the window should receive updates when the color environment changes, a short is passed which can have one of four different update states:

- `pmNoUpdates`—Don't want update events when color environment changes; similar to FALSE for `cUpdates` in `SetPalettes()`.
- `pmBkUpdates`—Want update events when color environment changes and window is *not* the active window.
- `pmFgUpdates`—Want update events when color environment changes and window is the active window.
- `pmAllUpdates`—Always want update events when color environment changes; similar to FALSE for `cUpdates` in `SetPalettes()`.

The following is a function which demonstrates many of these Palette Manager calls. A common operation is to grab a 'clut' in a resource and copy it into a window's palette:

```
//pass in a pointer to an existing window and the handle to a color table, and
//this function will install the color table's colors into the window. The function
//will, if necessary, create a palette from scratch and install that in the window.

void SetWindPaletteFromClut( WindowPtr theWindow, CTabHandle theClut )
{
  PaletteHandle  thePalette;

  // Create a palette from the 'clut'.
  thePalette = GetPalette( theWindow ); //get the window's existing palette
  if ( thePalette == NULL )             //GetPalette() returns NULL if not color ➥
window or no palette found
  {
    //create a palette for the window; use indicated usage and tolerance.
    //Remember: ctSize is zero-based!
    thePalette = NewPalette( (**theClut).ctSize + 1, theClut, pmTolerant + ➥
pmExplicit, 0x0000 );
  }
  else
  {
    //transmogrify the window's palette to match the input color table, using
    //indicated usage and tolerance
    CTab2Palette( theClut, thePalette, pmTolerant + pmExplicit, 0x0000 );
  }
```

```
    //install the new palette into the window; nothing visible happens yet
    NSetPalette( theWindow, thePalette, pmAllUpdates );

    //update color environment; may cause update events
    ActivatePalette( theWindow );
}
```

Drawing with Palettes

In most situations, when you want to draw, say, a red rectangle, you specify an `RGBColor` with lots of red in it, pass that to `RGBForeColor()`, and draw a merry little red rectangle. Of course, if you've been a thoughtful citizen and defined your colors as `pmTolerant`, when you call `RGBForeColor()` QuickDraw will find only the closest match to that color on the current device. This is fine for most things: The rectangle shows up nice and red in the window, and everybody's happy. However, in the context of games and other graphically precise applications, more control over color is necessary.

For instance, what if your artist says something like, "You'll need to cover up the treasure chest with a red rectangle until the player rolls a seven. You should use the color at index 42 so the rectangle will blend perfectly into the wall color." Just what is the best way to go about getting color 42?

The simplest answer is to have the artist provide you with the RGB values for that color and hard-code them into your program. Another answer is to keep a copy of the color table around and grab the RGBs out of index 42 when the situation warrants, then make your call to `RGBForeColor()`, etc. Both of these methods are cumbersome at best.

A better alternative is to specify the color at position 42 as `pmExplicit`. Now, instead of calling `RGBForeColor()` with an `RGBColor`, call `PmForeColor()` with 42. This is handy for several reasons: It makes communication with your art crew simpler (they think in terms of indexed color already), your code is a lot cleaner because several operations are replaced with one, and should you ever want to do color animation on that red rectangle, you'll be all set.

Note

When you talk to your artists about color indexes and they look at you like you've gone mental when you tell them that color indexes start at zero, just relax and let them think that indexes start at one. We know they really start at zero and go to wacky values like 255, but there's no need to worry their pretty little right lobes about that.

Let's recap: For game programming, we want utmost control of our color environment. In order to get the level of color control we want, we may have to sacrifice civility towards other programs which may be running. For ease of use, we'll use `pmExplicit` so we can refer to specific colors by index. Here's the best way to go about it:

1. Load your color table using something like: `yourCTable = GetCTable(yourCTableID)`.
2. Create a new palette:
   `yourPalette = NewPalette( 256, yourCTable , pmTolerant + pmExplicit, 0x0000 ).`
3. Give the palette to the window via
   `NSetPalette( yourWindow, yourPalette , pmAllUpdates ).`
4. Call `ActivatePalette(yourWindow)` to tell the system all of your palette manipulations are done and that it's time to update the color environment.
5. If you won't need your color table later, dump it nicely via `DisposeCTable(yourCTable)`, though it is a handy thing to have around if you're going to use GWorlds for off-screen drawing.

There are several permutations of this, but they all involve getting some colors and slamming them into the window and then notifying the hardware of the dirty deed.

Note

Using color tables and generating palettes on the fly is handy because you can use the color table should you need to create an off-screen buffer with those colors. Also, keeping a palette and *a clut around duplicates data and effort and increases the likelihood of error. Use cluts; practice safe hex.*

Palette Animation

This is actually a cool hack that is quite useful. Though Palette Animation is somewhat esoteric in nature, its effect can readily be appreciated. If you look at the Palette Anim project on the enclosed CD-ROM (in the 02 - Basic Game Graphics folder), you'll see a variety of effects: an annoying, flashing square; a slowly cycling rectangle; and several dots chasing themselves in a circle. Please note that all of these elements were drawn only once; thereafter, any perceived motion or animation is achieved simply by changing values in the hardware CLUT, a very quick operation.

Caution

Note that Palette Animation only works in Indexed devices; if you're going to use it, make sure your port is on an Indexed device!

You perform animation by setting the usage elements of palette to `pmAnimated`, then using `PmForeColor()` and `PmBackColor()` to draw with those indexes. Once your image is drawn, you can use the following calls to change the colors of those indexes in the hardware:

```
void AnimateEntry(WindowPtr dstWindow, short dstEntry, const RGBColor *srcRGB)
```

`AnimateEntry()` substitutes a particular RGB into a particular index (`dstEntry`) of the palette associated with a window. Of course, since `pmAnimating` colors are reserved from other applications, this will not cause a change in the color environment—though the hardware color changes, no applications will receive update events. Each device touched by the window's rect will have the reserved index modified.

```
void AnimatePalette(WindowPtr dstWindow, CTabHandle srcCTab, short srcIndex, short ➥
dstEntry, short dstLength)
```

`AnimatePalette()` is similar to `AnimateEntry()`, but allows you to change several colors to new, specific RGB colors obtained from a color table: `dstLength` number of colors are copied from the `srcCTab`, starting at `srcIndex`, and they are placed into the palette starting at `dstEntry`. This is handy, for instance, when you want to move a group of colors up or down, creating that familiar "rolling" effect seen in many fractal programs. Note that `srcIndex` is zero-based, as is `dstEntry`.

As a simple example of Palette Animation, let's consider the following spooky scenario: You have a black screen and you want to have a lightning bolt flash on and off very quickly. Here's what you might do:

1. Set up your window's palette with `pmExplicit`, so we can use indexes to refer to colors. Because we're animating via indexes, this is key.
2. Paint the screen with normal black.
3. Select a particular index and use `SetEntryColor()` to set its color to black.
4. Using `PmForeColor()`, set your forecolor to the index you just made black. Draw your lightning bolt. Since it's black, it sneakily blends right into the background.
5. Now, any time you want, you can call `AnimateEntry()` for your chosen index, stuff a suitable lightning bolt color in there, and *voilà!*—lightning! Of course, stuffing black into that index quickly will probably enhance the lightningesque quality of the animation, but you get the idea....

Note that the lightning had to be drawn only once, and the animation was simply a matter of making calls to `AnimateEntry()`. Simple. No hassling with off-screens or bothersome calls to `CopyBits()`. This is the kind of animation that programmers with tight deadlines really appreciate!

Several effects can be achieved with Palette Animation (the names of these effects are somewhat arbitrary since no one can decide once and for all what they're called):

- *Flashing*—Repeatedly stuff contrasting colors into an animated index.
- *Smooth Cycling*—Continuously stuff slowly changing color into an animating index.
- *Rolling Cycle*—With a range of animated colors, move the colors up or down so that what's drawn with them on the screen changes a notch. Anyone who's played around with fractal generating programs has seen this effect. (It's also how the chasers are animated in the Palette Anim project.)

FYI: Off-Screen Palette Animation

Setting bit 14 of an off-screen port's color table (`(**someColorTableHdl).ctFlags |= 0x4000;`) gives us a way to refer to palette indexes instead of RGB values for off-screens—this means you can use `PmForeColor()` and `PmBackColor()` when drawing in off-screen ports. This makes it possible to draw using animated indexes in an off-screen world. For an excellent discussion of the topic, see "Palette Manager Animation," by Rich Collyer, in the Winter, 1991 issue of develop *from Apple (if you have access to* develop *on CD-ROM, it's in issue 5).*

Of GWorlds and GDevices

Graphics Devices and Graphics Worlds are constructs which will help you immeasurably when doing graphics on the Macintosh. These objects and the Graphics Devices Manager make it very easy to control and use off-screen graphics. All of this, along with the incredibly versatile and speedy `CopyBits()` routine, will give you the power you need to quickly, efficiently, and easily write visually supple games.

A Graphic Buffer, Pixmap, and `CopyBits()` Primer

The term *graphic buffer* refers to chunks of RAM used to store digital images. On the Macintosh, this usually means a pixmap. A pixmap can be thought of as chunk of RAM containing a rectangular collection of pixels. A pixmap's pixels may have various color depths, from 1-bit black and white pixels to "true-color," 24-bit pixels.

Your computer's monitor is a rectangular collection of pixels and is, in fact, a special pixmap. It's special because a) it's associated with the hardware that drives the monitor, and b) you can see it. If you've opened a window on the screen, you may modify this pixmap with commands such as `FrameRect()` or `CopyBits()`.

What is `CopyBits()`? In its most basic form, `CopyBits()` is like `BlockMove()` in that it moves chunks of RAM from here to there. But `CopyBits()` is really a routine specifically tailored to copy all or portions of pixmaps around. Used well, `CopyBits()` is very smart and very quick. `CopyBits()` is used in virtually every application on the Macintosh and it's worth learning to use well.

But what does one use `CopyBits()` on? Sure, you can copy one part of your window to another, but how fun is that? More useful is to copy something, say, from some hidden window onto your on-screen window. Why, you could do all kinds of things with that! But what's this "hidden window" mumbo-jumbo? Well, it's really called an "off-screen buffer," usually shortened to simply "off-screen."

So, to recap:

- Graphic buffers are areas of RAM in which digital images can be stored. A graphic buffer may be on-screen or simply exist in RAM.
- Pixmaps are cool, Mac-specific graphic buffers. Your Mac monitor displays an example of an on-screen pixmap.
- `CopyBits()` is used to copy all or part of a pixmap from and to different parts of itself, or to another pixmap (including the screen via a window).

Why Use an Off-Screen?

No, it's not a dumb question, and it's worthwhile to review the reasons for employing an off-screen:

- You are in complete control of the off-screen color environment—no matter what's going on with the screens, your off-screens remain untouched.
- Doing complex drawing off-screen and blasting the result on-screen will appear to the user a lot faster than simply drawing on-screen.
- Sprites would be absolutely impossible without off-screens—off-screens allow us to use a technique called "double-buffering," which eliminates the dreaded problem of animation flicker.

Okay, we're all nodding our heads. Yes, these are good reasons for drawing off-screen, but why the heck do we have to know all this GDevice stuff? Well...

- GDevices encapsulate information about the monitors connected to a machine, which is a very handy thing: You can find the optimal screen for your game by walking through the list of screen GDevices.
- GDevices provide a consistent context for graphics on monitors, printers, and off-screens.
- A GDevice you set up to define an off-screen graphics environment can be used by several off-screen `grafPorts`.

Hmmm..., you think, but what about those GWorlds? The answer is that while they sound awfully unwieldy, they are simple to use, and you get these magnificent features for free:

- GWorld routines take care of a lot of yucky math for you. Do you really want to remember stuff like `offPort->portBits.rowBytes = ((offBounds->right - offBounds->left ➥ + 15) >> 4) << 1;`?
- `CopyBits()` *adores* GWorlds—you will grow to appreciate `CopyBits()` in a deep and meaningful way, so making it happy is definitely worthwhile.

- Pixmaps associated with GWorlds are handle-based (as opposed to old-fashioned, pointer-based bitmaps) and your heap will get less fragged because you can allow your off-screens to float around in memory.
- The `UpdateGWorld()` call makes it particularly easy to manage off-screens.
- Apple will maintain compatibility for all routines associated with GDevices in the future. Know them. Use them. Love them.

Sold! These things are great! Now if Apple would only give the Macintosh a screaming fast hardware blitter for its next birthday....

GWorlds

GWorlds are part of 32-bit QuickDraw, available as an init for System Version 6.07 and up and an integral part of System 7 and up. We're in the 90s now, and while backwards compatibility is a Good Thing, at some point you have to ring out the old and ring in the new. By now, any serious Mac user has System Version 6.08 or 7.x installed. (Less than five percent of the software-buying public uses anything less than System 7.)

GWorlds contain one each of a `CGrafPort`, a pixmap, a ColorTable, and a GDevice. For the most part, you can deal at a high level of abstraction using GWorlds, and only occasionally will you have to delve into their depths. However, they aren't awful monsters and are really quite manageable, and QuickDraw provides routines that provide a nice interface (heck, `UpdateGWorld()` will even align the pixmap it's updating to optimize `CopyBits()`, if you ask nicely!).

The *blit* (short for *Block Transfer*) is what GWorlds were invented for: If you just want to blast an image to the screen (say, a morph between Godzilla and your favorite politician), you can simply create a GWorld with the same characteristics as your screen, draw into it, and blit it to the screen. A look at the Simple GWorld project on the CD-ROM will show you a reasonably simple use of an off-screen GWorld.

For Simple GWorld, the idea was to do fancy and complicated (if ugly) drawing off-screen, and blast it to the screen all at once rather than force you to stare at its laborious drawing process. This is quite common, and something you should have intimate knowledge of.

Off-screen GWorlds encapsulate a lot of tricky operations and are probably the most useful thing brought to life in QuickDraw 1.2. They will certainly be around for a long time, so if you don't know them well already, isn't it about time you got cozy and familiar?

The most important routines regarding GWorlds are:

```
QDErr NewGWorld(GWorldPtr *offscreenGWorld, short pixelDepth, const Rect ➥
*boundsRect, CTabHandle cTable, GDHandle aGDevice, GWorldFlags flags)
```

Here, in one not-too-overwhelming call, is how you allocate an off-screen GWorld (also more generically called an off-screen buffer). Old-time Mac jocks will whine a lot about how

tough it was in the old days, but don't let that bother you. `NewGWorld()` allocates an off-screen port and pixmap; it will also allocate a new off-screen Graphics Device unless you specify that an existing device should be used instead—you may supply one explicitly or indicate that the GDevice with the deepest pixel depth intersecting `boundsRect` be used. Things to remember are:

- If all goes well, `offscreenGWorld` will contain a handle to your brand new off-screen.
- `pixelDepth` may be 0, 1, 2, 4, 8, 16, or 32; 0 signals that the depth and color table of the deepest device touched by `boundsRect` should be used, and `cTable` will be ignored.
- `boundsRect` is to be given in *global* coordinates.
- A copy of `cTable` is made and stored in the off-screen GWorld. `cTable` must specify a color table valid for the depth set with `pixelDepth`—if `cTable` is NULL, the default System color table of the appropriate depth will be used. However, if you specified 0 for `pixelDepth`, the color table of the deepest device intersecting `boundsRect` is used and `cTable` is ignored.
- If you specified `noNewDevice` in the `flags`, you must pass a valid device in `aGDevice`. If `pixelDepth` was set to 0, `aGDevice` is ignored and should be set to NULL.
- `flags` parameter provides some control over certain options. The `flags` may be a combination of `pixPurge`, `noNewDevice`, `keepLocal`, and `useTempMem`. Here are descriptions of these `flags`:
  - `pixPurge`—If you set `pixPurge` in the `flags`, `NewGWorld()` makes the off-screen buffer a purgeable block. Purgeability is checked when you call `LockPixels()` (described in a bit). Before you perform drawing or copying operations with any off-screen buffer, your application should ensure that the call to `LockPixels()` returns TRUE; if `LockPixels()` returns FALSE, the off-screen buffer has been purged and you should either call `UpdateGWorld()` to reallocate it or draw directly to the window; *never* draw into a purged off-screen!
  - `noNewDevice`—`NewGWorld()` will attempt to create a GDevice for your off-screen buffer unless you set this flag. Instead, the depth and color table from the `aGDevice` parameter you pass in are used, unless `pixelDepth` is 0, in which case the deepest device intersecting `boundsRect` supplies the depth and color table. If you need to use a custom color table in an off-screen GWorld, you will need to create the associated off-screen GDevice so QuickDraw can use certain data structures.
  - `keepLocal`—This keeps your off-screen buffer in the Mac's main RAM and prevents it from being cached to a graphics accelerator card. Avoid setting this unless you have a good reason (like poking around the pixmap by hand), since you lose any speed benefit the accelerator may afford.
  - `useTempMem`—This allocates your off-screen in temporary memory, something to avoid if you can. You should allocate the buffer, use it, and release it as soon as you can, and if at all possible make it purgeable (via `AllowPurgePixels()`).

```
void DisposeGWorld(GWorldPtr offscreenGWorld)
```

This call disposes of all structures associated with a particular off-screen GWorld, including color table, pixmap, and GDevice if one were created.

```
PixMapHandle GetGWorldPixMap(GWorldPtr offscreenGWorld)
```

This function returns a handle to the pixmap of an off-screen GWorld. Use this any and every time you need to reference the pixmap of an off-screen GWorld; dereferencing the GWorld pointer is a faux-pas to be avoided. Luckily, we can use this call in order to code in a civilized manner, such as: `LockPixels( GetGWorldPixMap(myGWPtr) );`

```
Boolean LockPixels(PixMapHandle pm)
```

This function ensures that off-screen pixmaps stay put (are "locked down") when you're doing any draw or copy operations. Otherwise, off-screen pixmaps are allowed to move around in memory (that's why they're handles). Always call this before drawing into or copying from an off-screen GWorld. Get the `PixMapHandle` to your off-screen using `GetGWorldPixMap()`. Note that if the off-screen is purgeable and has in fact been purged, this routine will return FALSE; either call `UpdateGWorld()` to reallocate your off-screen, or draw directly into the window it represents. Call `UnlockPixels()` as soon as you can to avoid spiking your heap.

```
UnlockPixels(PixMapHandle pm)
```

This is the reciprocal of `LockPixels()`; it allows your off-screen GWorld's pixmap to "float" in memory, just like a good handle should.

```
void AllowPurgePixels(PixMapHandle pm)
```

When you want to mark a GWorld's pixmap as purgeable, pass in a valid `PixMapHandle` (hopefully obtained using `GetGWorldPixMap()`) to this function and it will mark the pixmap as purgeable.

```
void NoPurgePixels(PixMapHandle pm)
```

Unsurprisingly, this marks the pixmap as not purgeable.

```
void SetGWorld(CGrafPtr port, GDHandle gdh)
```

This function sets the current port. You can pass several different flavors of port: `GrafPtr`, `CGrafPtr`, or `GWorldPtr`, if you coerce them when necessary. If you are passing either a `GrafPtr` or `CGrafPtr`, the port and GDevice are set as specified; if you are passing in a `GWorldPtr`, the current port is set as specified, but the GDevice parameter is ignored (therefore, you may pass NULL) and `SetGWorld()` will use the GDevice attached to the `GWorldPtr`.

```
void GetGWorld(CGrafPtr *port, GDHandle *gdh)
```

The inverse of `SetGWorld()`. Returns current port and current GDevice.

There are several other routines of interest (take a gander at `UpdateGWorld()`, for instance, especially if you'll be changing the depth or color table of your off-screen) but the ones given here are the basic set you cannot do without.

GDevices

There can be many windows on a screen, and each is described by a `CGrafPorts`—what describes the device (the screen) they all reside on? What structure describes the physical characteristics of the screen's drawing environment? GDevices, of course!

GDevices describe drawing environments and provide a uniform layer between your program and a physical display device (when Apple says QuickDraw is device-independent, it means it!). At startup, the Graphics Devices Manager queries the Slot Manager for each graphics device attached. A GDevice is created for each device capable of display, and added to a GDevice list of available GDevices.

If none of the GDevices meets your off-screen graphics needs, it is easy enough to create one of your own private GDevices. These private devices should not, however, be placed in the GDevice list.

GDevices provide a simple interface to a bewildering array of hardware out there. Using the functions provided by the Graphics Devices Manager, you can issue standard device control calls to a card's device-specific driver (e.g. "Yo! Set your pixel depth to 8").

```
typdef struct {
   short        gdRefNum;          //driver's unit number
   short        gdID;              //client ID for search procs
   short        gdType;            //fixed/CLUT/direct
   ITabHandle   gdITable;          //handle to inverse lookup table
   short        gdResPref;         //preferred resolution of GDITable
   SProcHndl    gdSearchProc;      //search proc list head
   CProcHndl    gdCompProc;        //complement proc list
   short        gdFlags;           //grafDevice flags word
   PixMapHandle gdPMap;            //describing pixmap
   long         gdRefCon;          //reference value
   Handle       gdNextGD;          //GDHandle handle of next GDevice
   Rect         gdRect;            //device's bounds in global coordinates
   long         gdMode;            //device's current mode
   short        gdCCBytes;         //depth of expanded cursor data
   short        gdCCDepth;         //depth of expanded cursor data
   Handle       gdCCXData;         //handle to cursor's expanded data
   Handle       gdCCXMask;         //handle to cursor's expanded mask
   long         gdReserved;        //future use. MUST BE 0
}GDevice, *GDPtr, **GDHandle;
```

Most of the stuff in the above GDevice structure you can ignore: GWorlds provide a nice, much more comprehensible interface to them. However, there are a few fields of interest:

- `gdType`—A device may be either Direct, Indexed, or Fixed; `gdType` tells you what a particular device is: 0 is Indexed, 1 is Fixed, and 2 is Direct.
- `gdPMap`—This is a handle to the device's pixmap; the pixmap's fields indicate, among other things, the pixel depth, and also whether the device is Direct or Indexed.

- `gdRect`—The bounds of the device in global coordinates are given by `gdRect`.
- `gdFlags`—These describe several interesting attributes of a GDevice. You can use `TestDeviceAttribute()` to test for any particular characteristic. Bit assignments follow.

```
enum {
   gdDevType = 0        //0 = monochrome, 1 = color
   roundedDevice  = 5, //1 if device has been "rounded" into the GrayRgn
   hasAuxMenuBar  = 6, //1 if device has an aux menu bar on it
   burstDevice  = 7,
   ext32Device  = 8,
   ramInit    = 10,     //1 if initialized from 'scrn' resource
   mainScreen   = 11,   //1 if main screen
   allInit    = 12,     //1 if all devices initialized
   screenDevice = 13,   //1 if screen device [not used]
   noDriver     = 14,   //1 if no driver for this GDevice
   screenActive = 15    //1 if in use
};
enum {
   clutType     = 0,    //0 if look-up table
   fixedType = 1,       //1 if fixed table
   directType   = 2     //2 if direct values
};
```

There are, of course, a few core functions that are worth covering:

```
GDHandle GetGDevice(void)
```

This obtains a handle to the current GDevice. It's useful for determining attributes of the current output device. Note that because a window can span more than one physical device, this does not return the device that describes a port. A handle to the current device is always kept in the `TheGDevice` global variable. (`GetGDevice()` and its partner `SetGDevice()` are analogous to `GetPort()` and `SetPort()` and can be used the same way.)

```
void SetGDevice(GDHandle gd)
```

This sets the `gd` to be the current device.

```
GDHandle GetMaxDevice(const Rect *globalRect)
```

Pass in a rectangle in global coordinates and this will return the GDevice handle of the deepest device the rectangle intersects.

```
GDHandle GetMainDevice(void)
```

This returns a handle pointing to the Main GDevice (you were expecting something different?).

```
GDHandle GetDeviceList(void)
```

This returns a handle to the first GDevice in the public devices list.

```
GDHandle GetNextDevice(GDHandle curDevice)
```

This returns a handle to the next device in the public GDevice list.

```
Boolean TestDeviceAttribute(GDHandle gdh, short attribute)
```

This allows you to test if any single attribute is TRUE (see the listing of bit assignments previously for `gdFlags`). You might be walking through the GDevice list and want to test each device to see if it is a screen device and if it is active—if both of those attributes are TRUE, you may decide to draw on that device.

```
void SetDeviceAttribute(GDHandle gdh, short attribute, Boolean value)
```

Given here for completeness, `SetDeviceAttribute()` is the reciprocal of `TestDeviceAttribute()`.

```
short HasDepth(GDHandle gd, short depth, short whichFlags, short flags)
```

This determines if a device supports a particular pixel depth. In `whichFlags`, you specify which of the `gdFlags` in the device you want checked, and in `flags` you specify the value to check for. The function will return 0 if the device does not support the mode you require; if the function does not return 0, the device supports the depth mode you've specified. Always test a device before setting its depth.


```
OSErr SetDepth(GDHandle gd, short depth, short whichFlags, short flags)
```

Before setting a device's depth, ask the user's permission! (For instance, ask, "Would you like to change to 8-bit color for speedy graphics, or remain at 16?") And, of course, present the option to switch back just before you quit. No matter what, when the system is rebooted, the original depth is restored. Currently, for `whichFlags` only the `gdDevType` flag has meaning. For instance, if you wanted to change a device to grayscale, you would specify a `whichFlags` value of 0x0001 (see the listing of bit assignments for `gdFlags`) and a `flags` value of 0x0000. This returns noErr (0) if depth were changed.

As stated, most of the time you can forget the more esoteric GDevice stuff, but be aware that a precise level of control is there if you need it.

For an example of a program which uses a modicum of GDevice stuff, check out the Amalgam program on the CD-ROM (in the 02 - Basic Game Graphics folder). It does several different things:

- Finds deepest screen available. Since there may be more than one screen with the same depth, this program finds the last, deepest one in the GDevice list.
- Finds the screen with the largest area. Again, since there may be more than one in the list with similar resolutions, this program will find the last, largest one it found in the GDevice list.
- Allows you to take over an entire window. If you have more than one screen available, whichever one contains a larger portion of the dialog will be taken over.
- Hides and unhides the menu bar.
- Fades a GDevice to black and back again.

- Demonstrates intelligent window zooming. Windows are zoomed out to whichever screen the majority of their real estate is on. Access to this function is via the usual zoom-box; since we do not provide for window resizing, you may reset the size of the window by Option-clicking on the zoom box.

There are a few things that should be noted about this program and its implementation of some of the features:

- Clicking in the close box of the dialog quits the program.
- Although the program lets you hide the menu bar at any time, real programs would only do so if they were taking over the entire main screen. To see what this is like, Minimax the dialog so it fills the main screen, then hide the menu bar. Now, click where the menu bar was and it will reappear. Then, click elsewhere in the dialog and it will resume hiding—this is what real games do.
- Notice that the menu bar is restored when you click out to the Finder or any other program; when you resume Amalgam, the menu bar goes back to whatever state you put it in.
- Fading demonstrated here fades an entire screen. This is the most robust method of fading, as it avoids some problems presented by the 68K AV Macs. Other methods of fading are to slowly modify the device's gamma and to do a color table animation fade—but both of these methods may fail on the AVs. Thanks, Apple.

Game Graphics Programming

What we've covered so far is pretty much the basic Macintosh graphics stuff presented with an eye toward games. There are several tricky areas that games always have to deal with, and we'll cover those here.

Multiple Monitor Tactics

Our demo code demonstrates a few no-nos when it comes to multiple monitor setups, and the most glaring offense is the assumption that the main screen is an Indexed, 256-color screen. It is common to find people whose main screen is an Indexed, 256-color *grayscale* screen, and whose second screen is a gargantuan color monitor that a drive-in movie theater would envy. You shouldn't get tripped up by this, and avoiding this particular gaffe is easy.

One method, and probably the best, is to simply ask the users where they want the game to be played (this question is, of course, moot if there's only one screen available, and you did check for that, right?). Pop up a movable modal dialog (like the Copy progress-bar dialog used by the Finder) displaying a sentence such as, "Drag me to the monitor on which you want to play this awesome game." After handling the drag event, use a routine such as the `GetMaxIntersectDevice()` provided in the Utils.c file to determine which GDevice the users

want to use. Check that GDevice against the kinds of devices your game can be played on. If it's a go, go. If not, gently inform the users that the game must be played on a monitor with such-and-so characteristics, and ask them to try again with a different monitor. If you're super-cool, you can even walk through the GDevice list and determine which monitor would be optimal and suggest that one to begin with.

Screen depth is another issue that should be dealt with gracefully. Most applications can run fine under a variety of screen configurations, but games, being such graphic beasties, often *require* a certain environment. Imagine that your game must run on an 8-bit Indexed device, but the users have only one monitor and it's set to 16-bit Direct. If it's a choice between being ugly and compliant, or beautiful and not-so-compatible, game programmers will always choose the latter (...nah, it's *way* too easy!).

This can be done with grace and charm. Okay, it can be done so there aren't angry crowds of villagers with torches surrounding your home. First of all, your game *will run* on any kind of monitor—all that pesky GDevice stuff is there for a reason—so you don't *require* a certain depth.

What you really want is to look good, and lucky for you that's what the users want, too. So, give them the option to run fast and lovely, or slow and compatibly (slow, because QuickDraw is going to have to go through mathematical contortions to display game graphics on nonoptimal screens).

If the users choose to change the bit depth of the monitor (you can make this change via `SetDepth()`), your application must be graceful enough to ask if they want to switch back to their original depth when they quit the game.

Fading Strategies

Fading to and from frames or stages in a game is an elegant and stylish way of transitioning—it can also avoid a nasty flash in color context if you're changing palettes. Here's how to do it properly:

1. Fade your chosen GDevice down to black.
2. Paint the entire screen flat black. Use the last entry in the palette to accomplish this—that entry will also be black in the palette you stuff in, so there's no need to worry about flashing.
3. Stuff in your new palette. Since the screen's black, there won't be any telltale flash.
4. Here is the tricky part: Fade to black again in one giant step. Now, the screen's still black, but the correct palette is set up for fading in.
5. Render your graphic on the screen. Since all entries are black, nothing will show up.
6. Fade in the GDevice, and your graphic will appear.

Having a second monitor for debugging fades is a must. Imagine fading to black and then jumping into your debugger—it's trying to tell you something important, but it's pitch black! Beg, borrow, or steal a second monitor for this stuff.

The shareware game Apeiron, included on the CD-ROM in the Games folder, fades to *white* instead of black, and it looks really cool. White works just as well as black; you just use the first entry in your palette rather than the last entry.

Hiding the Menu Bar Suavely

As mentioned previously in the description of the Amalgam program, you shouldn't go hiding the menu bar willy-nilly. There is one well-defined situation in which hiding the menu bar is okey-dokey:

- If you need to take over the whole screen, and it happens to be the main screen, then *let the users decide* to hide the menu bar or not.

Note the italics and let's repeat in unison: *let the users decide.* As with any radical change to the warm and GUI interface we love, we should at least tell the users what's going on and give them an option to do it or not.

One sneaky exception is in splash screens: You can hide the menu bar all you want, as long as you promise to give it back when the users click or the sign-in screen appears.

Actually, hiding the menu bar is something that, done well, can really enhance the players' immersion in a game. Doing it right, however, requires a little thought and reasoning. The Amalgam program shows how to hide the menu bar, but when to hide it is up to you.

Your Friend, `CopyBits()`

We've been saving the best for last. `CopyBits()` is where your program will spend a *lot* of time, and here we'll show you how to get the most out of it.

`CopyBits()` is one of the single most used Toolbox functions on the Macintosh. And for good reason: It's fast, flexible, and compatible. Because of its prominence, incredible amounts of time and money have gone into making sure that `CopyBits()` excels at its job.

And it does. But there several good ways to make it faster and a whole slew of ways to make it creep along slower than a snail going uphill on Jupiter.

First, let's get this out of the way:

```
void CopyBits(const BitMap *srcBits, const BitMap *dstBits, const Rect *srcRect, ➥
const Rect *dstRect, short mode, RgnHandle maskRgn)
```

Both `srcBits` and `dstBits` are addresses of bitmap or pixmap structures; `srcRect` is a pointer to the bounds of the size at the location of the source image and is expressed in terms of the `srcBits portRect`; `dstRect`, likewise, is the address of a rectangle expressing the destination

location and size as expressed in the destination port's coordinate system; `mode` is one of several mathematical transfer modes; `maskRgn` may be a region defining a clipping shape, or may be NULL indicating no special clipping region is supplied.

You are likely already familiar with `CopyBits()` by now, and if you aren't, you will be *more* than familiar with it by the time you get through chapter 3, "Advanced Graphics." A few observations:

- If `CopyBits()` is copying to the screen, the image may not only be clipped explicitly by the `maskRgn` parameter, it may also be clipped at the least by the `portRect` and `visRgn` of the port, as well as the user-definable `clipRgn` for the port. On the other hand, most of that time-consuming clipping goes away when blitting back and forth between off-screens.
- Any type of colorizing is mathematical molasses for `CopyBits()`. Avoid it by always making sure your `fgColor` is black and your `bkColor` is white, and by ensuring that the color tables and `ctSeeds` of Indexed GWorlds match (the infamous `ctSeed` Slam, described later).
- Copying lots of little rects (on the order of 100) incurs an overhead penalty that is noticeable; either try combining rectangles (as shown in the next chapter) or, if all else fails, try the tricky and dangerous (and not likely to be compatible in the future) direct-to-screen drawing.
- Blitting between pixmaps of different depths is also guaranteed to be achingly slow.
- Pixmaps that are not aligned chew up time and are easily avoided.
- Scaling images is also a time-eater.

If you can avoid these ugly things and make sure you're doing just the simplest blits you can, you will enjoy very efficient blitting, since the vast majority of time is spent squirting bits from one memory location to another. Because 90 percent or more of your blits should fall into this category, eking out a little more speed is often an exercise in frustration. Really, `CopyBits()` is that good. `CopyBits()` does special case a lot of things, but that overhead is negligible compared to the time spent actually moving bits around.

Still, there are times when speed is all that matters (hmm... when *isn't* speed a factor in game development?) and there are certainly things that can be done to minimize extra work on `CopyBits()`' part. Here, we will address the most useful techniques for speeding up our old friend:

- When blitting to and between Indexed GDevices, make sure the same color table is used for all. Even if the color tables are identical, QuickDraw won't know it unless you ensure the `ctSeeds` are the same for both: `( *( ( *( srcPixMap ) )->pmTable ) )->ctSeed = ( *( ( *( ( *aGDevice )->gdPMap ) )->pmTable ) )->ctSeed;` This is the `ctSeed` Slam, and if differing `ctSeeds` are slowing you down when they shouldn't, this will cure the problem pronto.

Caution

One giant exception to using the `ctSeed` Slam is when the screen is a Fixed device (currently, the only Fixed screens exist on grayscale PowerBooks—they have a fixed grayscale ramp as their hardware color table). Recall that the `gdType` of a GDevice will contain a 0 for Indexed, 1 for Fixed, and 2 for Direct. Never *do the `ctSeed` Slam trick if the screen is a Fixed device. Best strategy: Create a set of art using the same luminance values and in the same order as the color table on the Fixed device; then, when you call `CopyBits()`, QuickDraw will note that your off-screen's `ctSeed` is different, compare the two color tables, note that they are identical, and then copy quickly (that is, without translation). If you use the same luminances but the colors are in a different order, QuickDraw will have to translate each pixel as it copies, which is very slow (but, luckily, guaranteed to work). Worst case: You've done the `ctSeed` Slam and your art doesn't exactly match the hardware colors; QuickDraw does* no *color translation and your art looks horrible. Lesson: At startup, determine if screen is Fixed; if so, use appropriate art and avoid the `ctSeed` Slam. End of sermon.*

- If you must clip, try to clip to simple rectangular regions; `CopyBits()` has been fine-tuned for blitting to those (think about all the rectangular blits in the Finder).
- Avoid dithering, in life as well as in blitting.
- Blitting a squat, wide thing is quicker than blitting a tall, skinny thing of the same area, because there are fewer rows of pixels involved and there are fewer address calculations necessary. (Consider a dirty rects routine: In order to blisteringly blit blocks, you might write the routine so that it really likes to see two dirty objects that are near each other horizontally and make one dirty rect out of them when similar blocks of equal size and *vertical* distance from each other would not be combined.)
- Pixel alignment can be very important to speeding up `CopyBits()`. Simply put, copying operations will be fastest when the source and destination addresses are aligned on 16-bit or 32-bit boundaries. The best thing you can do is make sure your windows are longword-aligned (start at a global pixel coordinate divisible by four) and have a width that is a multiple of four. `NewGWorld()` will set up an equivalent off-screen that is also aligned. `UpdateGWorld()` will also attempt to realign pixmaps.
- Stay away from `CopyBits()`' fanciful cousins, `CopyMask()` and `CopyDeepMask()`. Most of the operations you might want to perform with `CopyMask()` will be better served by various skip-draw methods introduced in the next chapter, and nobody can think of a compelling reason to use `CopyDeepMask()` in a time-critical, game-oriented operation.

Chapter 3
Advanced Graphics

Most games involve some form of animation: figures running, ships moving, monsters drooling, and so on. However, creating animation that is fast and smooth is tricky.

If you are near your computer, open the Book Chapters folder, then the 03 - Advanced Graphics folder on the CD-ROM. Within that, open the folder Example 7. Run the application there called Demo68K or DemoPPC (whichever matches your machine). You will see ten colored disks bouncing quickly over a bluish background. Press Command-Q to exit.

In this chapter, we're going to start with a simple animation program. Then we will gradually refine it into the program you just ran.

Where Did I Get that Art?

The number disks are actually characters from Zapf Dingbats. They were typed into Photoshop 3.0, beveled, and colored.

The background was generated in a texture maker called Alien Skin.

All the graphics were imported into DeBabelizer to determine the best color set.

The Theory

How are we going to present an animated scene to the players?

First, let's define some terms:

- *Background*—The image behind the animation. For the purposes of this discussion, it is not animated. Animated and scrolling backgrounds are beyond the scope of this chapter.
- *Sprite*—An image drawn on top of the background. Sprites are the items in an animated scene that are moving. They can be an arbitrary shape.

In this chapter, we will focus on using an off-screen buffer. With an off-screen buffer (colloquially known as an "off-screen"), we can prepare the image we want and then quickly copy that image onto the screen. If we do this fast enough, players will perceive a smooth animation.

There are three steps involved for each frame:

1. Restore the background of the off-screen to make sure that the off-screen is ready to have new images drawn on it.
2. Draw the sprites.
3. Copy the off-screen to the screen. This shows the player this frame of animation.

As we refine the example in this chapter, we will work on making all three of these steps as fast as possible.

The Practice

Theory is one thing, implementation another. We start with a simple, straightforward implementation and then enhance it.

Example 1: A Very Simple Scene

Let's start with a simple implementation of the ideas we're talking about: restoring the background, drawing the sprites, and updating the screen. In this case, we use the Toolbox routine `EraseRect()` to restore the entire background to a simple white. The sprites are drawn with another Toolbox routine, `DrawPicture()`. Finally, for simplicity, the entire off-screen is copied to the screen.

The code, demo, resources, and project files are in the Example 1 folder, which is in the 03 - Advanced Graphics folder on the CD-ROM. Open it up and run either Demo68K or DemoPPC, depending on your machine. The demo responds to a few controls, as shown in table 3.1.

Table 3.1
Demo Controls

Action	Meaning
Click and hold mouse button	While the button is down, run the demo without polling for events.
Press a number key	Toggle that disk's visible state.
Press C	Turn clipping mode on or off.
Press Command-Q	Quit.

Try each of the options in the demo, especially using the clipping mode and making disks appear and disappear.

What you should see is ten colored, numbered disks bouncing slowly around on your screen. We'll look at speeding it up soon, but let's look at the code for this example.

The code is organized into three modules: `main`, `demo`, and `sprite`. Each of these has a .c and .h file. There is a global.h which has values that are used throughout the project. Also, the data for the project is stored in the resource file Demo.res.

global.h

Let's start by looking at global.h:

```
//
// File: global.h
//
// This file contains defines of a global nature.
//
// 2/18/95 -- Created by Mick
//

// shield
#ifndef _global_h_
#define _global_h_

// global defines

// useful constants
#define kNil                0L          // the value of an invalid pointer
#define kTrue               1           // the value of truth
#define kFalse              0           // the value of untruth

#define kWindowInFront   ( WindowPtr )-1L           // the value to pass to ➥
NewWindow() to place the window in the front

// resource IDs

// color tables ('clut')
#define kAppColorTableResID     128         // the resource ID of the 'clut' ➥
resource that defines the colors for the program

// windows ('WIND')
#define kMainWindowResID        128         // the resource ID of the 'WIND' ➥
resource for the main window

// PICTs ('PICT')
#define kNumberOne     201     // the resource ID of the picture of the 1 disk
#define kNumberTwo     202     // the resource ID of the picture of the 2 disk
#define kNumberThree   203     // the resource ID of the picture of the 3 disk
#define kNumberFour    204     // the resource ID of the picture of the 4 disk
#define kNumberFive    205     // the resource ID of the picture of the 5 disk
#define kNumberSix     206     // the resource ID of the picture of the 6 disk
#define kNumberSeven   207     // the resource ID of the picture of the 7 disk
#define kNumberEight   208     // the resource ID of the picture of the 8 disk
```

```
#define kNumberNine    209     // the resource ID of the picture of the 9 disk
#define kNumberTen     210     // the resource ID of the picture of the 10 disk

// end of shield
#endif // #ifndef _global_h_
```

The file starts with a shield to prevent it from being included more than once. Actually, in CodeWarrior you can create shields by placing

```
#pragma once
```

at the top of a header file. However, this `pragma` is not supported in all the compilers out there, so if your code is going to be used for multiple compilers, use the shield method.

The next section consists of defines for values that we need almost everywhere. `kNil` is a value for invalid pointers. `kTrue` and `kFalse` are values of Boolean variables and expressions. `kWindowInFront` is a value used by the Toolbox routines `NewWindow()`, `NewCWindow()`, `GetWindow()`, and `GetCWindow()`. It places the new window in front of all the other windows.

The last section of global.h is a list of resource ID numbers for resources that we load during the program. These are divided by types: color tables ('clut'), windows ('WIND'), and pictures ('PICT').

Finally, the shield is closed.

Demo.res

Demo.res contains all the data for the application. Open it in ResEdit or Resorcerer, and let's look at its resources.

- 'clut' #128—This contains a table of 256 colors that is used to set the on- and off-screen colors for the demo.
- 'WIND' #128—This is a borderless window that covers an entire 13" screen.
- 'PICT' #201-#210—Each of these contains the graphic for one of the colored discs in the demo.

main

Now that we've looked at the global values and the data, let's follow the execution of the demo. We start at the beginning: the `main` module. Here is `main()`:

```
//
// main()
//
//   This is where it all begins.
//
```

```
void main( void )
{
  PaletteHandle appPalette;       // the palette that we will make the ➥
application palette

  // start up the profiler
#if __profile__
  if ( ProfilerInit( collectDetailed, bestTimeBase, 100, 10 ) != noErr )
    {
      Debugger();
    }
#endif

  // fire up the Toolboxes
  initMacToolboxes();

  // get the color table and make it the default palette
  gAppColorTable = GetCTable( kAppColorTableResID );
  appPalette = NewPalette( 256, gAppColorTable, pmExplicit + pmTolerant, 0 );
  SetPalette( ( WindowPtr )-1L, appPalette, kFalse );

  // hide the menu bar
  hideMenuBar();

  // create and show the window
  gMainWindow = GetNewCWindow( kMainWindowResID, ( Ptr )kNil, kWindowInFront );
  ShowWindow( gMainWindow );
  SetPort( gMainWindow );

  // give the demo code a chance to set up
  startupDemo();

  // handle events until the user signals a quit
  doEventLoop();

  // give the demo code a chance to shut down
  shutdownDemo();

  // hide and delete the window
  HideWindow( gMainWindow );
  DisposeWindow( gMainWindow );

  // show the menu bar again
  showMenuBar();

  // shut down the profiler
#if __profile__
  ProfilerDump( "\pDemo.prof" );
  ProfilerTerm();
#endif
}
```

Overall, `main` falls into three parts: startup, event loop, and shutdown. In startup, `main` initializes, loads, and allocates all the things needed for the program to operate. The event loop runs the demo, responds to user input, and waits for the user to signal a quit. The shutdown part cleans up before the program ends.

`main` starts with some code surrounded by `#if` and `#endif`. This code is included only if the Generate Profiler Calls option is checked in the preferences. Otherwise, it's left out. This code initializes the profiler.

What Is a Profiler?

A profiler measures where the processor spends its time. This data can tell us what parts of a program are taking the most time to execute. The CodeWarrior profiler works by adding a call at the beginning and end of every function that tells the profiling code which function has been called. This data is collected while the program runs and is dumped to a file when the program ends. We'll be using the profiler to tell us where to focus our efforts in improving this code.

The next line of code calls `initMacToolboxes()`. This is another function in main.c. Let's take a short detour to look at it:

```
//
// initMacToolboxes()
//
//   Initialize all the Toolboxes that we will need.
//

void initMacToolboxes( void )
{
   // initialize all the Toolboxes
   InitGraf( &( qd.thePort ) );
   InitFonts();
   InitWindows();
   InitMenus();
   TEInit();
   InitDialogs( ( ProcPtr )kNil );
   InitCursor();

   // make sure that we have the entire heap
   MaxApplZone();

   // allocate some extra master pointers
   MoreMasters();
   MoreMasters();
   MoreMasters();
   MoreMasters();

   // set the random seed
   qd.randSeed = TickCount();
}
```

This code calls all the Toolbox initialization routines in the proper order. It then grabs all the space allocated to this program for the main heap. Next, it allocates additional master pointers so we won't run out of handles. Finally, the random number seed is set to the number of Ticks since machine startup. This last point is important because the velocities and initial positions of the ten disks are randomly determined. If the random seed isn't set to something that is different every time the program runs, the disks will always start with the same position and velocity.

The next block of code in `main` loads the color table from the resource into a global, `gAppColorTable`. It then uses the color table to create a palette with the same colors. The call to `SetPalette()` makes this palette the default palette for the application.

What's all this for? Why do we need both a color table and a palette? Both a color table and a palette provide a list of colors. However, a palette is a list of colors for a window and includes extra information to tell the operating system how to allocate colors on the screen. The screen has hardware for only one set of colors that all the windows on that screen must share. Off-screen drawing environments do not need to share, so they need only the list of colors, the color table.

We want control of all of the colors on the screen and we want our off-screen environments to have the same colors as our on-screen environment. Therefore, it make sense to derive one from the other. Choosing between loading the color table and deriving the palette or loading the palette and deriving the color table depends on several factors. The advantage of loading the palette and creating a color table from it is that you can have different usage values for parts of the color set. If you have a palette at ID number 0, it will automatically become the default palette. On the other hand, most tools that build color lists save them as color tables, so creating a palette from a color table streamlines working with these tools. In general, if you are going to use only one color set, loading the palette may make more sense. On the other hand, if you wish to change colors during your program, loading the color table may be better.

The next line in `main` calls `hideMenuBar()`. Let's detour again and look at it:

```
//
// hideMenuBar()
//
// Hides the menu bar and adds it to the area that can be drawn
//

void hideMenuBar( void )
{
   GDHandle mainScreen;                  // the information on the main screen
   Rect mainScreenRect;                  // the rect that bounds the menu bar
   RgnHandle mainScreenRgn;              // the region of the menu bar

   // record the menu bar height
   sMenuBarHeight = GetMBarHeight();
```

```
    // set the menu bar to no height
    LMSetMBarHeight( 0 );

    // save the original gray regions (so we can restore it later)
    sOriginalGrayRgn = NewRgn();
    CopyRgn( GetGrayRgn(), sOriginalGrayRgn );

    // make sure that the entire main screen is ours to play with (no menu bars or ➥
corners)
    mainScreenRgn = NewRgn();
    mainScreen = GetMainDevice();
    mainScreenRect = ( ( *mainScreen )->gdRect );
    RectRgn( mainScreenRgn, &mainScreenRect );
    UnionRgn( sOriginalGrayRgn, mainScreenRgn, GetGrayRgn() );

    // draw the desktop
    PaintOne( ( WindowRef )kNil, sOriginalGrayRgn );
}
```

In order to hide the menu bar, we have to change two things: the menu bar height and the gray region. The menu bar height tells the system how high to draw the menu bar. The gray region defines the size and shape of the desktop—the area in which windows can be drawn. To hide the menu bar we need to do two things. First, we have to save the current menu bar height and gray region so we can restore them later. Second, we must set the menu bar height to 0 and add the menu area to the gray region.

That's exactly what this routine does. It starts by saving the old menu bar height and then sets it to 0. Next it copies the gray region and adds the menu bar to the original gray region. Finally, it forces a redraw of everything.

Let's return to `main`. In order to display images to the screen, we need a window. The next set of code does just that. It builds a window based on a 'WIND' resource, shows it, and makes it the current port. This window will have the default palette that we created earlier. When the code shows the window, the screen will adjust to match the palette.

The next line of code calls `startupDemo()`. This gives the demo module a chance to load data, allocate space, and do other things to get ready. We'll go into detail on that routine later—when we cover the `demo` module.

After the call to `startupDemo()`, `main` calls `doEventLoop()`. All of the startup is complete and the program is ready to go. This routine runs the program until the user asks to quit. We will delve into `doEventLoop()` right after we finish with `main`.

After the user has signaled that she wants to quit, we have to prepare to end the program. The next line gives the `demo` module an opportunity to do just that with a call to `shutdownDemo()`. Like `startupDemo()`, we'll examine that when we look at the `demo` module.

Now that the demo is finished, we should get rid of the window. The next two lines hide the window and dispose of the memory it used. Actually, when a program ends, all of the windows it created are hidden and disposed of, so these calls are not necessary.

Before quitting, the menu bar must be restored—that won't be done automatically. To restore it, `main` calls `showMenuBar()`:

```
//
// showMenuBar()
//
// Shows the menu bar and removes it from the drawable area.
//

void showMenuBar( void )
{
   // set the menu bar to its normal height
   LMSetMBarHeight( sMenuBarHeight );

   // restore the original gray region
   SectRgn( sOriginalGrayRgn, GetGrayRgn(), GetGrayRgn() );

   // draw the menu bar
   DrawMenuBar();
}
```


This routine restores the menu bar height and gray region to the values that `hideMenuBar()` stored before it removed the menu bar. After restoring those values, it redraws the menu bar to make the change appear.

The last thing that `main` does is shut down the profiler—if we are using it. The call to `ProfileDump()` saves the data the profiler collected to a file called Demo.prof. `ProfilerTerm()` terminates the profiler.

Earlier we skipped over `doEventLoop()`. Let's examine it now:

```
//
// doEventLoop()
//
// Get and process events until the user signals a quit.
//

void doEventLoop( void )
{
   unsigned char quitNow;    // a flag that notes if the user has signaled a quit
   EventRecord theEvent;     // a place to put an event

   quitNow = kFalse;
   while( !quitNow )
      {
```

```
        if( WaitNextEvent( everyEvent, &theEvent, 0L, ( RgnHandle )kNil ) == kTrue)
           {
              switch( theEvent.what )
                 {
                    case mouseDown:
                       // give the demo uninteruppted time to draw
                       while( StillDown() )
                          {
                             doDemoFrame();
                          }
                       break;
                    case keyDown:
                   // check to see if the user has hit Command-Q (to signal a quit)
                       if ( ( theEvent.modifiers & cmdKey ) && ( ( theEvent.message & ➥
charCodeMask ) == 'q' ) )
                          {
                             quitNow = kTrue;
                          }
                       else
                          {
                             // send the key to the demo code
                             demoKey( ( unsigned char )( theEvent.message & ➥
charCodeMask ) );
                          }
                       break;
                    case updateEvt:
                       BeginUpdate( gMainWindow );
                       EndUpdate( gMainWindow );
                    default:
                       break;
                 }
           }
        else
           {
              // give the demo code some time to run
              doDemoFrame();
           }
     }
}
```

`doEventLoop()` does two things: It processes user input and it runs the demo. User input comes in the form of events—key pressed, mouse clicked, and so on. `doEventLoop()` processes these events and, when there are no events, runs the demo.

The routine starts with a loop. It sets up the exit variable—`quitNow` set to false. Then it starts a loop to wait for the exit variable to become true. The loop consists of getting events and processing them. In the beginning of the loop, it asks for the next event. If it gets one, it handles the event depending on the event's type.

If there is a mouse-down event, `doDemoFrame()` is called repeatedly until the user lets up on the mouse. `doDemoFrame()` is the routine in the demo module that moves the demo ahead one "frame." Think of frames as single images in an animation, like individual images in a movie. Like `startupDemo()` and `shutdownDemo()`, we'll look at `demoFrame()` after we finish with the `main` module. The loop around this call to `demoFrame()` allows the user to see how fast the demo would go without the event-polling mechanism. It's not very noticeable on this demo, but as we speed up the demo, it will be.

The next type of event it checks for is a key-down. If the user has hit Command-Q, the exit flag is set. This will cause the event-polling loop to end. If the user did not press Command-Q, the key is passed to `demoKey()`. This allows the demo code to respond to keystrokes. We'll look at `demoKey()` when we look at the rest of `demo`.

The last type the code handles is an update event. Actually, this code just throws that event away. A real program would handle update events by updating the areas affected. In this case, the program simply clears the update area so that more update events are not generated.

`doEventLoop()` is a minimal implementation of an event loop. A real program would handle more event types—suspend, resume, high level, and so on—and would handle these events more completely. This is a demo focusing on graphics, so this area of the code is somewhat minimal.

That's all of the `main` module. It will not change much in upcoming revisions, so we won't need to cover it again. Now on to the `demo` module.

demo

The `demo` module controls the demo itself. It handles both simulating and drawing the moving disks as well as updating the screen. The code for the `demo` module is in demo.c and demo.h. We've already seen calls to four functions in `demo`: `startupDemo()`, `shutdownDemo()`, `doDemoFrame()`, and `demoKey()`.

Before we look at them, however, we need to look at a structure definition:

```
typedef struct
{
   tSpriteInfo *fSpriteInfo;        // the information on the sprite for this object
   unsigned char fIsVisible;        // should we draw this object
   signed long fXPos;               // the x position of this object (16:16 fixed)
   signed long fYPos;               // the y position of this object (16:16 fixed)
   signed long fDeltaX;             // the x coord of the velocity of this object
   signed long fDeltaY;             // the y coord of the velocity of this object
} tObjectInfo;
```

This structure defines information about one of the bouncing disks. The first field is a pointer to a sprite information structure that contains the size and graphic data for a sprite. We'll look at the exact definition of the `tSpriteInfo` structure in the `sprite` module.

The next field is a flag that notes if the disk is visible or not. `demo` uses this flag to determine if it should draw the disk. The user can set or clear this flag by pressing the number keys.

The next two fields are the x and y positions of the disk. These numbers are fixed point numbers. A fixed point number is a way to represent a noninteger number. Technically, the high 16 bits of the long are the integer and the low 16 bits are the fraction. The easy way to think of fixed point numbers is to visualize them as numbers that have been multiplied by 65,536. For example, 1.5 would be 98,304. This will shift the number over 16 bits so that the integral and fractional parts are in the proper places. The x and y positions are fixed numbers so that the velocities can be less than one.

Speaking of velocity, the next two fields are the x and y velocities. These values are added to the position of the disk every frame. Like the position, these values are fixed point numbers.

`demo` maintains one of these structures for each disk in an array called `sObjects`. It updates the disks every frame and uses them to draw the scene.

Moving to the code, we start with `startupDemo()`:

```
//
// startupDemo()
//
// Create and load all the buffers and data needed for the demo.
//

void startupDemo( void )
{
  unsigned short indexCounter;          // a counter to scan the object array

  // create the off-screen GWorld
  NewGWorld( &gOffscreenBuffer, 8, &( gMainWindow->portRect ), gAppColorTable, ➥
( GDHandle )kNil, keepLocal );

  // get the pixel map and rect
  gOffscreenPixels = GetGWorldPixMap( gOffscreenBuffer );
  gOffscreenRect = gMainWindow->portRect;

  // load the sprite
  for( indexCounter = 0; indexCounter < kMaxObjects; indexCounter++ )
    {
      // load the sprites
      sObjects[ indexCounter ].fSpriteInfo = loadSprite( kNumberOne + ➥
indexCounter );

      // give it a random location and velocity
      sObjects[ indexCounter ].fXPos = ( Random() & 0x00FF ) << 16;
      sObjects[ indexCounter ].fYPos = ( Random() & 0x00FF ) << 16;
```

```
        sObjects[ indexCounter ].fDeltaX = Random() << 4;
        sObjects[ indexCounter ].fDeltaY = Random() << 4;

        // initially, it is visible
        sObjects[ indexCounter ].fIsVisible = kTrue;
      }

   // determine the clip rect
   sClipRect = gOffscreenRect;
   InsetRect( &sClipRect, kClipRectInset, kClipRectInset );

   // set the initial flags
   sClippingFlag = kFalse;
}
```

`startupDemo()` gets everything ready for the demo. It starts by creating the off-screen buffer. This buffer is the same size as the window created in `main` and uses the color table loaded in `main`. `startupDemo()` then extracts and stores the handle to the pixel map and the bounds rectangle of the off-screen.

Next, `startupDemo()` initializes each of the disks. The sprites are loaded and random positions and velocities are set. The visible flag is set TRUE so all the disks are visible when the program starts.

Finally, some values for the clipping mode are determined. The clipping rectangle is calculated and stored. The clipping flag is set to FALSE—the program starts out of clipping mode.

When the user presses a key, that key is passed on to `demoKey()`. Let's look at what `demoKey()` does with those keystrokes.

```
//
// demoKey()
//
// Handle a key stroke -- allows the user to set options while the demo runs.
//

void demoKey( unsigned char inKey )
{
   // depending on the key pressed
   switch( inKey )
      {
         case 'c':    // turn clipping on and off
            sClippingFlag = !sClippingFlag;
            break;
         case '1':    // turn this number on and off
            sObjects[ 0 ].fIsVisible = !sObjects[ 0 ].fIsVisible;
            break;
         case '2':    // turn this number on and off
            sObjects[ 1 ].fIsVisible = !sObjects[ 1 ].fIsVisible;
            break;
         case '3':    // turn this number on and off
```

```
            sObjects[ 2 ].fIsVisible = !sObjects[ 2 ].fIsVisible;
            break;
         case '4':    // turn this number on and off
            sObjects[ 3 ].fIsVisible = !sObjects[ 3 ].fIsVisible;
            break;
         case '5':    // turn this number on and off
            sObjects[ 4 ].fIsVisible = !sObjects[ 4 ].fIsVisible;
            break;
         case '6':    // turn this number on and off
            sObjects[ 5 ].fIsVisible = !sObjects[ 5 ].fIsVisible;
            break;
         case '7':    // turn this number on and off
            sObjects[ 6 ].fIsVisible = !sObjects[ 6 ].fIsVisible;
            break;
         case '8':    // turn this number on and off
            sObjects[ 7 ].fIsVisible = !sObjects[ 7 ].fIsVisible;
            break;
         case '9':    // turn this number on and off
            sObjects[ 8 ].fIsVisible = !sObjects[ 8 ].fIsVisible;
            break;
         case '0':    // turn this number on and off
            sObjects[ 9 ].fIsVisible = !sObjects[ 9 ].fIsVisible;
            break;
         default:     // unknown key
            SysBeep( 0 );
            break;
      }
}
```

This code is pretty simple. Depending on what key is pressed, the code does one of three things. If the C key is pressed, clipping is toggled. If one of the number keys is pressed, that disk's visible flag is toggled. Finally, if any other key is pressed, a beep is played to indicate an unknown key.

The real action happens in `doDemoFrame()`:

```
//
// doDemoFrame()
//
// Move the demo ahead one frame -- move the sprites and draw.
//

void doDemoFrame( void )
{
   CGrafPtr oldPort;           // the graf port that is in place when we are called
   GDHandle oldDevice;         // the GDevice that is in place when we are called
   signed short indexCounter; // a counter to scan all the objects
   Rect clipFrame;             // the rect of the frame to the clipping area
```

```
    Point drawPoint;            // where to draw the sprite

    // save the current port and GDevice
    GetGWorld( &oldPort, &oldDevice );

    // set the off-screen buffer as current (and lock the pixel map)
    SetGWorld( gOffscreenBuffer, ( GDHandle )kNil );
    LockPixels( gOffscreenPixels );

    // restore the background
    restoreBackground();

    // move the objects
    moveObjects();

    // set up to draw the shapes (and handle the clipping)
    if ( sClippingFlag )
      {
        clipFrame = sClipRect;
        InsetRect( &clipFrame, -1, -1 );
        FrameRect( &clipFrame );
        startSpriteDraw( &sClipRect );
      }
    else
      {
        startSpriteDraw( &gOffscreenRect );
      }

    // draw the sprites from back to front
    for( indexCounter = kMaxObjects - 1; indexCounter >= 0; indexCounter-- )
      {
        if ( sObjects[ indexCounter ].fIsVisible )
          {
            drawPoint.h = ( sObjects[ indexCounter ].fXPos ) >> 16;
            drawPoint.v = ( sObjects[ indexCounter ].fYPos ) >> 16;
            drawSprite( sObjects[ indexCounter ].fSpriteInfo, drawPoint );
          }
      }

    // shut down the shape draw
    endSpriteDraw();

    // copy the buffer to the screen
    blitToScreen();

    // restore the current port and GDevice
    UnlockPixels( gOffscreenPixels );
    SetGWorld( oldPort, oldDevice );
}
```

`doDemo()` frame coordinates the creation and display of a single frame in the animation. More than any other routine, this is the implementation of the off-screen drawing theory we discussed early in this chapter.

`doDemoFrame()` starts by saving the current port and device so it can restore them before it exits. It then prepares to draw to the off-screen buffer by locking the pixel map. In a GWorld, the pixel map is stored in a handle and can be moved by the memory manager. So before drawing to a GWorld, you need to lock the handle, and you need to unlock the handle when you're finished drawing so the memory manager can store the data most efficiently. The system calls `LockPixels()` and `UnlockPixels()` do just that.

Next, it removes the image from the last frame by `restoreBackground()`. In this version of the demo, restore background is very simple:

```
//
// restoreBackground()
//
// This routine redraws the background over changed areas.
//

void restoreBackground( void )
{
   // erase the entire off-screen
   EraseRect( &gOffscreenRect );
}
```

`restoreBackground()` simply calls `EraseRect()` to erase the entire off-screen to white. Though quite simple, `restoreBackground()` may waste some time. One of the improvements we make to the demo will happen here.

Back in `doDemoFrame()`, `moveObjects()` is called to simulate the movement of each of the disks.

```
//
// moveObjects()
//
// Move all the objects.
//

void moveObjects( void )
{
   unsigned short indexCounter;       // a counter to scan all the objects

   // consider each object
   for( indexCounter = 0; indexCounter < kMaxObjects; indexCounter++ )
      {
         // apply the velocity
         sObjects[ indexCounter ].fXPos += sObjects[ indexCounter ].fDeltaX;
         sObjects[ indexCounter ].fYPos += sObjects[ indexCounter ].fDeltaY;
```

```
        // if the object has hit an edge, bounce it
        if( sObjects[ indexCounter ].fXPos < ( gOffscreenRect.left << 16 ) )
            {
                sObjects[ indexCounter ].fDeltaX *= -1;
                sObjects[ indexCounter ].fXPos += ( gOffscreenRect.left << 16 ) - ➥
sObjects[ indexCounter ].fXPos;
            }
        if( sObjects[ indexCounter ].fYPos < ( gOffscreenRect.top << 16 ) )
            {
                sObjects[ indexCounter ].fDeltaY *= -1;
                sObjects[ indexCounter ].fYPos += ( gOffscreenRect.top << 16 ) - ➥
sObjects[ indexCounter ].fYPos;
            }
        if( sObjects[ indexCounter ].fXPos +
                ( ( ( sObjects[ indexCounter ].fSpriteInfo )->fSpriteRect.right -
                ( sObjects[ indexCounter ].fSpriteInfo )->fSpriteRect.left ) << 16 ) >
                ( gOffscreenRect.right << 16 ) )
            {
                sObjects[ indexCounter ].fDeltaX *= -1;
                sObjects[ indexCounter ].fXPos += ( ( gOffscreenRect.right -
                        ( ( sObjects[ indexCounter ].fSpriteInfo )->fSpriteRect.right -
                        ( sObjects[ indexCounter ].fSpriteInfo )->fSpriteRect.left ) ) << ➥
16 ) -
                        sObjects[ indexCounter ].fXPos;
            }
        if( sObjects[ indexCounter ].fYPos +
                ( ( ( sObjects[ indexCounter ].fSpriteInfo )->fSpriteRect.bottom -
                ( sObjects[ indexCounter ].fSpriteInfo )->fSpriteRect.top ) << 16 ) >
                ( gOffscreenRect.bottom << 16 ) )
            {
                sObjects[ indexCounter ].fDeltaY *= -1;
                sObjects[ indexCounter ].fYPos += ( ( gOffscreenRect.bottom -
                        ( ( sObjects[ indexCounter ].fSpriteInfo )->fSpriteRect.bottom -
                        ( sObjects[ indexCounter ].fSpriteInfo )->fSpriteRect.top ) ) << ➥
16 ) -
                        sObjects[ indexCounter ].fYPos;
            }
    }
}
```

While all this looks quite complex, it's actually fairly simple. The complexity comes from all the structures that need to be accessed. The loop at the beginning scans through the disks. For each disk, the new position is calculated by adding the velocity to the current position. The next four `if` statements make sure that the disk has not hit the left, top, right, or bottom edge. If it has, the corresponding component of the velocity is reversed and its position is adjusted by how far beyond the edge it moved. In simple terms, it bounces.

Let's go back to `doDemoFrame()`. Now that the objects have moved, `doDemoFrame()` can get ready to draw them. How it gets ready depends on if the demo is in clipping mode. If it is, the clipping frame is drawn and `startSpriteDraw()` is called with the clipping rectangle. If it's not in clipping mode, `startSpriteDraw()` is called with the off-screen's rectangle. `startSpriteDraw()` gives the `sprite` module a chance to prepare before sprites are drawn. The rectangle parameter sets the clipping rectangle. We'll see more of `startSpiteDraw()` in the `sprite` module.

Now that everything is ready, it's time to draw. Drawing in a consistent order is important so that things pass in front and behind each other consistently. `doDrawFrame()` draws the disks so that the lower-numbered ones are in front. That means that they have to be drawn last. The `for` loop scans the object array backwards, drawing each disk by first determining the integer position of the disk and then calling `drawSprite()`. We'll look at `drawSprite()` later.

Now that the sprites are drawn, `endSpriteDraw()` is called to give the `sprite` module a chance to clean up if it needs to. When we look at `startSpiteDraw()` and `drawSprite()` later, we'll look at `endSpriteDraw()` as well.

The off-screen contains the image, but the screen does not. The next line calls `blitToScreen()` to transfer the off-screen image.

```
//
// blitToScreen()
//
// This routine updates the screen from the off-screen buffer.
//

void blitToScreen( void )
{
   CGrafPtr oldPort;        // the graf port that is in place when we are called
   GDHandle oldDevice;      // the GDevice that is in place when we are called

   // save the current port and GDevice
   GetGWorld( &oldPort, &oldDevice );

   // set the drawing environment to the screen
   SetGWorld( ( CWindowPtr )gMainWindow, GetMainDevice() );

   // make sure that the fore and back colors are correct to prevent colorize mode
   ForeColor( blackColor );
   BackColor( whiteColor );

   // Copy the screen's color table seed into the source pixmap.
   // This will minimize CopyBits()' setup time.
   ( *( ( *gOffscreenPixels )->pmTable ) )->ctSeed = ( *( ( *( ( *( GetGDevice() ) ➥
) ->gdPMap ) )->pmTable ) )->ctSeed;

   // copy the buffer to the screen
   CopyBits( ( BitMap * )( *gOffscreenPixels ), &( gMainWindow->portBits ),
        &gOffscreenRect, &( gMainWindow->portRect ),
```

```
        srcCopy, ( RgnHandle )kNil );

    // restore the current port and GDevice
    SetGWorld( oldPort, oldDevice );
}
```

`blitToScreen()` starts by making sure that `CopyBits()` will run as fast as possible. First, it sets the port and device to the on-screen window. Then, it sets the foreground and background colors to black and white. This prevents `CopyBits()` from entering colorize mode, in which it replaces black pixels with the current foreground color.

Next, `blitToScreen()` copies the color table seed from the screen's color table to the off-screen's color table. Why? One of the things that `CopyBits()` does is translate colors from one color set to another. We have made sure that the on-screen and the off-screen color sets match. `CopyBits()`, however, does not know this. Therefore, before it moves the data, it checks the two tables to see if they match. If they do, it just moves the data. If not, it translates. This comparison is slow, but there is a shortcut. Think of color table seeds as serial numbers—each color table gets a unique seed. So, `CopyBits()` first compares the color table seed values. If they match, it assumes the tables are the same and skips the rest of the test.

At last, it's time to show the user the image. The `CopyBits()` call moves the entire image from the off-screen to the window. Like the `restoreBackground()` call, moving the entire image may be wasteful. We'll examine this later.

After the `CopyBits()` call, the original port and device are restored and control returns to `doDemoFrame()`.

`doDemoFrame()` is basically finished. It restores the port and device that were in place when it was called and exits.

The last routine in `demo` is `shutdownDemo()`.

```
//
// shutdownDemo()
//
// Release all the memory used by the demo.
//

void shutdownDemo( void )
{
    unsigned short indexCounter;          // a counter to scan the object array

    // dump the sprites
    for( indexCounter = 0; indexCounter < kMaxObjects; indexCounter++ )
        {
            disposeSprite( sObjects[ indexCounter ].fSpriteInfo );
        }

    // dump the off-screen GWorld
    DisposeGWorld( gOffscreenBuffer );
}
```

This routine frees all the memory allocated in `startupDemo()`. First, it calls `disposeSprite()` for each object. Then it disposes of the off-screen buffer.

Well, there is the `demo` module. There's only one more to go: `sprite`.

sprite

The `sprite` module controls the loading, disposing, and drawing of sprites. The `sprite` module for this example is actually quite simple. The sprites are being drawn by the operating system call `DrawPicture()`, so we don't have that much work to do. However, most of the improvements in the upcoming examples focus on sprite drawing, so we'll be working with this code a great deal. The code for the sprite module is in sprite.c and sprite.h.

In the `tObjectInfo` structure from demo.c, there is a field that points to a `tSpriteInfo` structure. Earlier we glossed over what was in that field. Here it is in full detail:

```
typedef struct
{
   Rect fSpriteRect;           // the rect that defines the bounds of the sprite
   PicHandle fSpritePict;      // the sprite data itself
} tSpriteInfo;
```

There are only two fields in the structure: the rectangle of the sprite and the picture that contains the sprite data.

Why not have one structure for both the sprite and the object? There are two reasons. First, there is not always a one-to-one relation between objects and sprites. For example, a single object may have many frames of animation, or a single sprite may be used by several objects, or both. The object information structure contains the information about the specific object: where it is, where it's going, what sprite it's using, and so on. The sprite information structure contains information about that sprite only. The other reason not to combine the two structures is modularity. The objects are handled in one module of the program, `demo`. Sprites are handled in a different module, `sprite`. By keeping the structures separate, we can modify `sprite` or `demo` without changing the other. In fact, we are going to be doing a lot of this.

The first routine we look at is `loadSprite()`:

```
//
// loadSprite()
//
', Loads/allocates all the data for a sprite.
//

tSpriteInfo *loadSprite( signed short inSpriteResID )
{
   tSpriteInfo *newSprite;       // the new sprite data

   // create the sprite info record
   newSprite = ( tSpriteInfo * )NewPtr( sizeof( tSpriteInfo ) );
```

```
    // load the PICT
    newSprite->fSpritePict = GetPicture( inSpriteResID );
    if ( newSprite->fSpritePict == ( PicHandle )kNil )
        {
            // if it did not load, drop into the debugger -- real programs would have ➥
error checking
            Debugger();
        }

    // copy its bounds rect
    newSprite->fSpriteRect = ( *( newSprite->fSpritePict ) )->picFrame;

    // return the new sprite!
    return newSprite;
}
```

`loadSprite()` is pretty simple. First, it allocates the sprite information structure. Next, it loads the sprite's picture and stores its handle in the sprite structure. Then, it determines the bounding rectangle for the picture and stores that in the structure as well. Finally, it returns a pointer to the structure.

What we allocate, we should dispose of. Here is `disposeSprite()`:

```
//
// disposeSprite()
//
// Disposes of/releases all the memory used in a sprite
//

void disposeSprite( tSpriteInfo *inSpriteInfo )
{
    // dump the picture
    ReleaseResource( ( Handle )( inSpriteInfo->fSpritePict ) );

    // free the structure
    DisposePtr( ( Ptr )inSpriteInfo );
}
```

`loadSprite()` loads one resource and allocates one pointer. `disposeSprite()` releases that resource and disposes of that pointer.

Before sprites are drawn, some preparation must be done. After sprites have been drawn, some cleanup may be necessary. To provide opportunities for this, there are two functions: `startSpriteDraw()` and `endSpriteDraw()`. To use them, you call `startSpriteDraw()` before drawing sprites. Then you draw the sprites. After all the sprites have been drawn, call `endSpriteDraw()` to clean up. Between start and end, you should only draw sprites. Think of these two calls as being something like `BeginUpdate()` and `EndUpdate()`: You call `BeginUpdate()` to prepare for an update. You perform the update. You call `EndUpdate()`.

Here is `startSpriteDraw()`:

```
//
// startSpriteDraw()
//
// Prepare the sprite draw. Assumes that the port is set to the destination port.
//

void startSpriteDraw( Rect *inClipRect )
{
   // save the current port's clip region
   sOldClipRgn = NewRgn();
   GetClip( sOldClipRgn );

   // set the clip region to be the passed in rect
   ClipRect( inClipRect );
}
```

We are using `DrawPicture()` to draw the sprite, so any clipping will have to be done with the off-screen's clip region. `startSpriteDraw()` saves the current clip region and sets it to the clipping rectangle that is passed in.

`endSpriteDraw()` reverses the process:

```
//
// endSpriteDraw()
//
// End the sprite draw sequence.
//

void endSpriteDraw( void )
{
   // restore the old clip region
   SetClip( sOldClipRgn );

   // dump the region
   DisposeRgn( sOldClipRgn );
}
```

`endSpriteDraw()` restores the original clip region and dumps the copy.

We have seen how sprites are loaded and disposed of and how to prepare for and clean up after drawing sprites. All that's left is drawing them. That is done in `drawSprite()`:

```
//
// drawSprite()
//
// Draw the sprite in the port.
//
```

```
void drawSprite( tSpriteInfo *inSpriteInfo, Point inWhere )
{
   Rect destRect;    // where we want to draw the sprite

   // calculate the destination rect
   destRect = inSpriteInfo->fSpriteRect;
   OffsetRect( &destRect, inWhere.h, inWhere.v );

   // draw the sprite
   DrawPicture( inSpriteInfo->fSpritePict, &destRect );
}
```

First, `drawSprite()` calculates the rectangle where the sprite will be drawn. Then it tells `DrawPicture()` to draw the sprite. This is as simple as sprite drawing gets. Upcoming examples will revise this routine heavily.

Analysis

There you have it—a simple animated scene. It works, it's simple, and it's smooth. However, it is slow.

But where is it slow? To find that out, we use the profiler. Recompile with the Generate Profiler Calls option set. Run the program for a while, then quit. The profiler data is saved in a file called Demo.prof. Double-click on it to run the profiler. Select the summary in the View menu. Look at the percentage value next to the Only column (see figure 3.1).

Demo.prof

Method: Detailed Timebase: PowerPC Saved at: 10:35:49 PM 3/6/95 Overhead: 231.077

Function Name	Count	Only	%	+Children	%	Average	Maximum	Minimum
drawSprite	5040	28224.679	56.2	28224.679	56.2	5.600	12.937	4.617
bltToScreen	504	11393.834	22.7	11393.834	22.7	22.607	23.455	21.843
restoreBackground	504	5584.107	11.1	5584.107	11.1	11.080	44.230	10.277
doEventLoop	1	4005.928	8.0	49904.788	99.4	4005.928	4005.928	4005.928
doDemoFrame	504	357.340	0.7	45898.860	91.4	0.709	1.392	0.553
startSpriteDraw	504	314.062	0.6	314.062	0.6	0.623	1.468	0.246
hideMenuBar	1	105.078	0.2	105.078	0.2	105.078	105.078	105.078
loadSprite	10	64.782	0.1	64.782	0.1	6.478	7.393	5.913
startupDemo	1	50.277	0.1	115.059	0.2	50.277	50.277	50.277
initMacToolboxes	1	45.718	0.1	45.718	0.1	45.718	45.718	45.718
endSpriteDraw	504	17.947	0.0	17.947	0.0	0.036	0.210	0.000
showMenuBar	1	16.146	0.0	16.146	0.0	16.146	16.146	16.146
disposeSprite	10	16.094	0.0	16.094	0.0	1.609	1.935	1.475
moveObjects	504	6.890	0.0	6.890	0.0	0.014	0.879	0.000
shutdownDemo	1	0.659	0.0	16.753	0.0	0.659	0.659	0.659

Figure 3.1 *Profiler results.*

These are the results that I got from this code on a Power Macintosh 8100/80. Your results may be somewhat different, especially for 68K systems. However, these results show one interesting fact: 56.2 percent of the program's time is spent in `drawSprite()`. If we want to speed up the demo, we must improve `drawSprite()`.

On the other hand, using `DrawPicture()` for sprite drawing does have some advantages:

- It's easy to program.
- The data is stored in a convenient format—'PICT' resources.

`DrawPicture()` sprites would be very useful in areas in which high speed is not necessary. For example, buttons in a panel do not need to run very fast; `DrawPicture()` would be ideal here.

Example 2: `CopyMask()`

Calling `DrawPicture()` every time we draw the sprite is clearly too slow. Maybe we could draw the picture once, then just copy it quickly each time we need it. Is there a Toolbox function that will do that for us? Actually, there is one routine that is almost ideal: `CopyMask()`.

Before we discuss `CopyMask()`, we need to discuss what a mask is. When a sprite is drawn, the data must tell the routine which pixels to draw and which ones to skip. Otherwise, the sprite would fill the entire rectangle and nothing behind it would show through. A mask is one of the ways to note if a pixel is to be drawn (see figure 3.2).



Figure 3.2 *A shape and mask.*

Look at the two images in figure 3.2. Every pixel in the disk that we want to draw (on the left) has a corresponding black pixel in the mask (on the right). A routine that uses a mask to draw checks each pixel in the mask. If the mask is set, it copies the corresponding pixel. If it's not set, the pixel is not drawn. Actually, a mask does not have to be an image; it can, for example, be a region.

What about the pictures in the last example? They didn't have any masks, did they? Actually, they did. When the disks were copied with the lasso tool common to many paint programs (in this case Studio/32), a mask region was built into the picture. When the pictures were drawn, that mask region was used to draw the disk correctly.

There are a few ways to transfer data based on a mask:

- Use the mask region parameter of `CopyBits()`. The mask can be built from an image with `BitMapToRegion()`.
- Create the image as a color icon resource ('cicn') and use `PlotCIcon()`. The color icon format contains mask data.
- Use `CopyMask()`.

We're going to examine just one of these options: `CopyMask()`. It provides the functionality we are looking for without the bitmap-to-mask conversion issue that `CopyBits()` causes. And we avoid the color conversion problems that `PlotCIcon()` causes.

In order to implement `CopyMask()`, we need to modify a few files: Demo.res, global.h, sprite.c, and sprite.h. There are also some minor changes to demo.c—we'll mention them as well.

Demo.res

If we are going to use masks, we need some masks to match the pictures. The masks are stored as black-and-white pictures. Their ID numbers range from 211 to 220.

global.h

The additional picture IDs have been added to global.h:

```
#define kMaskOne    211  // the resource ID# of the mask for the 1 disk picture
#define kMaskTwo    212  // the resource ID# of the mask for the 2 disk picture
#define kMaskThree  213  // the resource ID# of the mask for the 3 disk picture
#define kMaskFour   214  // the resource ID# of the mask for the 4 disk picture
#define kMaskFive   215  // the resource ID# of the mask for the 5 disk picture
#define kMaskSix    216  // the resource ID# of the mask for the 6 disk picture
#define kMaskSeven  217  // the resource ID# of the mask for the 7 disk picture
#define kMaskEight  218  // the resource ID# of the mask for the 8 disk picture
#define kMaskNine   219  // the resource ID# of the mask for the 9 disk picture
#define kMaskTen    220  // the resource ID# of the mask for the 10 disk picture
```


`sprite`

The bulk of the modifications are in the `sprite` module, which has been almost completely rewritten.

Let's start with the sprite information structure:

```
typedef struct
{
   Rect fSpriteRect;          // the rect that defines the bounds of the sprite
   GWorldPtr fSpriteWorld;    // the GWorld for the sprite data
   PixMapHandle fSpritePix;   // the pixmap of the sprite data
   GrafPtr fMaskPort;         // the port of the mask data
} tSpriteInfo;
```

The sprite rectangle is still the same. However, the sprite picture has been replaced with two fields: `fSpriteWorld` and `fSpritePix`. `fSpriteWorld` is a pointer to the GWorld with which the sprite was drawn into a buffer. `fSpritePix` is `fSpriteWorld`'s buffer. The pixel image of the sprite is in the buffer. The mask is stored in a black-and-white port pointed to by `fMaskPort`.

Now let's see how we set up one of these structures. Here is the new version of `loadSprite()`:

```
//
// loadSprite()
//
// Loads/allocates all the data for a sprite.
//

tSpriteInfo *loadSprite( signed short inSpriteResID, signed short inMaskResID )
{
  tSpriteInfo *newSprite;  // the new sprite data
  PicHandle spritePict;    // the sprite/mask picture
  CGrafPtr oldCPort;       // the graf port that is in place when we are called
  GrafPtr oldPort;         // the b&w port that we are saving
  GDHandle oldDevice;      // the GDevice that is in place when we are called

  // create the sprite info record
  newSprite = ( tSpriteInfo * )NewPtr( sizeof( tSpriteInfo ) );

  // load the PICT
  spritePict = GetPicture( inSpriteResID );
  if ( spritePict == ( PicHandle )kNil )
    {
      // if it did not load, drop into the debugger -- real programs would have ➥
error checking
      Debugger();
    }

  // copy its bounds rect
  newSprite->fSpriteRect = ( *spritePict )->picFrame;

  // create a GWorld for it
  NewGWorld( &newSprite->fSpriteWorld, 8, &( newSprite->fSpriteRect ),
      gAppColorTable, ( GDHandle )kNil, keepLocal );

  // extract the pixmap handle
  newSprite->fSpritePix = GetGWorldPixMap( newSprite->fSpriteWorld );

  // save the current port and GDevice
  GetGWorld( &oldCPort, &oldDevice );

  // set the off-screen buffer as current (and lock the pixel map)
  SetGWorld( newSprite->fSpriteWorld, ( GDHandle )kNil );
  LockPixels( newSprite->fSpritePix );

  // draw the picture
  EraseRect( &( newSprite->fSpriteRect ) );
  DrawPicture( spritePict, &( newSprite->fSpriteRect ) );
```

```
    // restore the current port and GDevice
    UnlockPixels( newSprite->fSpritePix );
    SetGWorld( oldCPort, oldDevice );

    // dump this PICT
    ReleaseResource( ( Handle )spritePict );

    // load the mask PICT
    spritePict = GetPicture( inMaskResID );

    // allocate a new port
    newSprite->fMaskPort = ( GrafPtr )NewPtr( sizeof( GrafPort ) );
    OpenPort( newSprite->fMaskPort );

    // fiddle with its port bits
    newSprite->fMaskPort->portBits.bounds = newSprite->fSpriteRect;
    newSprite->fMaskPort->portBits.rowBytes = ( ( newSprite->fSpriteRect.right -
newSprite->fSpriteRect.left ) + 15 ) / 8;
    newSprite->fMaskPort->portBits.baseAddr =
        ( QDPtr )NewPtr( newSprite->fMaskPort->portBits.rowBytes * ( newSprite
->fSpriteRect.bottom - newSprite->fSpriteRect.top ) );

    // save the port
    GetPort( &oldPort );

    // set the new port
    SetPort( newSprite->fMaskPort );

    // draw the picture
    EraseRect( &( newSprite->fSpriteRect ) );
    DrawPicture( spritePict, &( newSprite->fSpriteRect ) );

    // restore the old port
    SetPort( oldPort );

    // dump this PICT
    ReleaseResource( ( Handle )spritePict );

    // return the new sprite!
    return newSprite;
}
```

This sure is a lot longer than the `loadSprite()` from the last example! This routine needs to do three things:

- Allocate the sprite structure and fill in its fields.
- Create a color buffer with the sprite image in it.
- Create a black-and-white buffer with the sprite mask in it.

First on the list: Allocate the sprite structure and fill in its fields. `loadSprite()` starts by allocating space for the sprite structure. It then loads the sprite picture and extracts the bounding rectangle from that.

Next, the code creates an off-screen world to hold the sprite. Both the GWorld and the `PixMapHandle` from the off-screen world are stored in the structure. The sprite picture is drawn into the buffer. Then, the picture resource is freed.

Now all that remains is the buffer with the black-and-white mask. First, the mask picture is loaded. Next, a buffer to draw it must be created. Unfortunately, we need a bitmap, not a pixmap, for the mask. The GWorld stuff will not help us here. So the code allocates and initializes a black-and-white port and a bitmap. It then draws the mask picture into the bitmap. Finally, it dumps the picture and returns the sprite information structure.

One final note about `loadSprite()`. It now takes an additional parameter: `inMaskResID`. `startupDemo()` is modified to send that parameter as well. The modifications are slight, so we will not go into them here.

Fortunately, `disposeSprite()` has not grown as much a `loadSprite()`:

```
//
// disposeSprite()
//
// Disposes of/releases all the memory used in a sprite
//

void disposeSprite( tSpriteInfo *inSpriteInfo )
{
   // dump the GWorld
   DisposeGWorld( inSpriteInfo->fSpriteWorld );

   // dump the memory from the bitmap
   DisposePtr( ( Ptr )( inSpriteInfo->fMaskPort->portBits.baseAddr ) );

   // dump the port
   DisposePtr( ( Ptr )( inSpriteInfo->fMaskPort ) );

   // free the structure
   DisposePtr( ( Ptr )inSpriteInfo );
}
```

There are a few more disposals: the GWorld, the bitmap, and the black-and-white port. However, the spirit of this action remains: Dispose of everything allocated or loaded in `loadSprite()`.

There is only one change to `startSpriteDraw()`:

```
//
// startSpriteDraw()
//
// Prepare the sprite draw. Assumes that the port is set to the destination port.
//

void startSpriteDraw( Rect *inClipRect, PixMapHandle inDestPixMap )
{
   // save the current port's clip region
   sOldClipRgn = NewRgn();
   GetClip( sOldClipRgn );

   // set the clip region to be the passed in rect
   ClipRect( inClipRect );

   // save the pixmap info (so we can use it)
   sDestPixMap = inDestPixMap;
}
```

`CopyMask()` needs to have the destination pixel map handle passed to it. So, that information is now passed into `startSpriteDraw()` so the routine can save it. The code in `demo` that calls `startSpriteDraw()` has also been modified to send this parameter, but like the changes to the caller of `loadSprite()`, they do not warrant examination.

Other than handling this new parameter, `startSpriteDraw()` is exactly the same as it was before. `endSpriteDraw()` is also exactly the same as it was, so we'll skip it.

Let's finish up by looking at `drawSprite()`:

```
//
// drawSprite()
//
// Draw the sprite in the port.
//

void drawSprite( tSpriteInfo *inSpriteInfo, Point inWhere )
{
   Rect destRect;    // where we want to draw the sprite

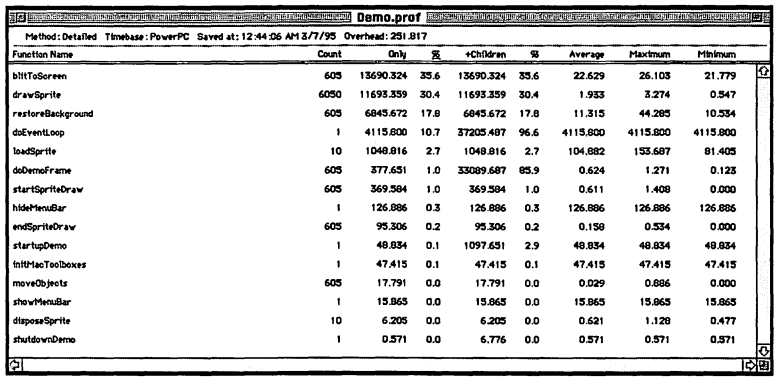
   // calculate the destination rect
   destRect = inSpriteInfo->fSpriteRect;
   OffsetRect( &destRect, inWhere.h, inWhere.v );

   // draw the sprite
   CopyMask( ( BitMap * )( *( inSpriteInfo->fSpritePix ) ), &( inSpriteInfo ➥
->fMaskPort->portBits ),
         ( BitMap * )( *sDestPixMap ), &( inSpriteInfo->fSpriteRect ), ➥
&( inSpriteInfo->fSpriteRect ), &destRect );
}
```

Surprisingly, `drawSprite()` has only one change: The `DrawPicture()` has been replaced with the call to `CopyMask()`. The call to `CopyMask()` may look complex, but the parameters are fairly obvious: the source buffer, the mask buffer, the destination buffer, the source rectangle, the mask rectangle, and the destination rectangle.

Analysis

If you have not already, run the demo. It is noticeably faster. Let's profile it (see figure 3.3).



Demo.prof

Method: Detailed Timebase: PowerPC Saved at: 12:44:06 AM 3/7/95 Overhead: 251.817

Function Name	Count	Only	%	+Children	%	Average	Maximum	Minimum
blitToScreen	605	13690.324	35.6	13690.324	35.6	22.629	26.103	21.779
drawSprite	6050	11693.359	30.4	11693.359	30.4	1.933	3.274	0.547
restoreBackground	605	6845.672	17.8	6845.672	17.8	11.315	44.285	10.534
doEventLoop	1	4115.600	10.7	37205.487	96.6	4115.600	4115.600	4115.600
loadSprite	10	1048.816	2.7	1048.816	2.7	104.882	153.687	81.405
doDemoFrame	605	377.651	1.0	33089.687	85.9	0.624	1.271	0.123
startSpriteDraw	605	369.584	1.0	369.584	1.0	0.611	1.408	0.000
hideMenuBar	1	126.886	0.3	126.886	0.3	126.886	126.886	126.886
endSpriteDraw	605	95.306	0.2	95.306	0.2	0.158	0.534	0.000
startupDemo	1	48.834	0.1	1097.651	2.9	48.834	48.834	48.834
initMacToolboxes	1	47.415	0.1	47.415	0.1	47.415	47.415	47.415
moveObjects	605	17.791	0.0	17.791	0.0	0.029	0.686	0.000
showMenuBar	1	15.865	0.0	15.865	0.0	15.865	15.865	15.865
disposeSprite	10	6.205	0.0	6.205	0.0	0.621	1.128	0.477
shutdownDemo	1	0.571	0.0	6.776	0.0	0.571	0.571	0.571

Figure 3.3 *Profiler results.*

The percentage of time that `drawSprite()` takes has dropped from 56.2 percent to 30.4 percent—a big change. In fact, it is no longer the champion time user: `blitToScreen()` takes 35.6 percent.

Still, `CopyMask()` is not perfect. It requires that we keep two sets of data for each sprite. And, while it is faster than `DrawPicture()`, it's no speed demon.

Example 3: A Clear Color Sprite Draw

Why do we need a mask? The mask tells us which pixels to draw and which pixels to leave alone. However, keeping two sets of graphics around—sprite data and mask data—can be a real pain.

Is there a way we can tell which pixels to draw and which to skip without using a separate image? In fact, there is. In 8-bit color, we have 256 colors. If we can spare one of those colors, we can get rid of the mask data. We take a color index, whatever it is, and consider it a marker for pixels that are not part of the shape. We'll call that marker the "clear color." Now when we draw, we check to see if the pixel is that clear color. If it is, we don't draw the pixel. If the pixel's not clear, we do draw.

That sounds pretty good. Is there a Toolbox routine to do this? No, we're going to have to draw the sprite ourselves.

Let's look at the changes we have to make. Like the last example, almost all the changes are limited to `sprite`. There are a few small modifications to global.h and Demo.res.

Demo.res

Actually, the only change to Demo.res is the removal of the mask pictures. We no longer need them.

global.h

With the mask pictures removed, the #defines for the mask resource IDs go away as well.

`sprite`

As usual, let's start with the sprite information structure:

```
typedef struct
{
   Rect fSpriteRect;           // the rect that defines the bounds of the sprite
   GWorldPtr fSpriteWorld;     // the GWorld for the sprite data
   PixMapHandle fSpritePix;    // the pixmap of the sprite data
} tSpriteInfo;
```


The only change here is the removal of the mask field; it's no longer needed.

With the removal of the mask, `loadSprite()` is simplified as well:

```
//
// loadSprite()
//
// Loads/allocates all the data for a sprite.
//

tSpriteInfo *loadSprite( signed short inSpriteResID )
{
   tSpriteInfo *newSprite;  // the new sprite data
   PicHandle spritePict;    // the sprite/mask picture
   CGrafPtr oldCPort;       // the graf port that is in place when we are called
   GDHandle oldDevice;      // the GDevice that is in place when we are called

   // create the sprite info record
   newSprite = ( tSpriteInfo * )NewPtr( sizeof( tSpriteInfo ) );

   // load the PICT
   spritePict = GetPicture( inSpriteResID );
   if ( spritePict == ( PicHandle )kNil )
      {
         // if it did not load, drop into the debugger -- real programs would have ➥
error checking
```

```
        Debugger();
      }

    // copy its bounds rect
    newSprite->fSpriteRect = ( *spritePict )->picFrame;

    // create a GWorld for it
    NewGWorld( &newSprite->fSpriteWorld, 8, &( newSprite->fSpriteRect ),
         gAppColorTable, ( GDHandle )kNil, keepLocal );

    // extract the pixmap handle
    newSprite->fSpritePix = GetGWorldPixMap( newSprite->fSpriteWorld );

    // save the current port and GDevice
    GetGWorld( &oldCPort, &oldDevice );

    // set the off-screen buffer as current (and lock the pixel map)
    SetGWorld( newSprite->fSpriteWorld, ( GDHandle )kNil );
    LockPixels( newSprite->fSpritePix );

    // draw the picture
    EraseRect( &( newSprite->fSpriteRect ) );
    DrawPicture( spritePict, &( newSprite->fSpriteRect ) );

    // restore the current port and GDevice
    UnlockPixels( newSprite->fSpritePix );
    SetGWorld( oldCPort, oldDevice );

    // dump the PICT
    ReleaseResource( ( Handle )spritePict );

    // return the new sprite!
    return newSprite;
}
```

If you look, this code is identical to the code from the `CopyMask()` example, except that the code to create the black-and-white buffer and draw the mask is gone.

Not surprisingly, `disposeSprite()` is shorter as well:

```
//
// disposeSprite()
//
// Disposes of/releases all the memory used in a sprite
//

void disposeSprite( tSpriteInfo *inSpriteInfo )
{
```

```
    // dump the GWorld
    DisposeGWorld( inSpriteInfo->fSpriteWorld );

    // free the structure
    DisposePtr( ( Ptr )inSpriteInfo );
}
```

The code to dispose of the mask buffer and port is gone. The rest is unchanged.

`startSpriteDraw()` has changed, but not much:

```
//
// startSpriteDraw()
//
// Prepare the sprite draw. Assumes that the port is set to the destination port.
//

void startSpriteDraw( Rect *inClipRect, PixMapHandle inDestPixMap )
{
    // set the clip region to be the passed in rect
    sClipRect = *inClipRect;

    // save the pixmap info (so we can use it)
    sDestPixMap = inDestPixMap;

    // extract the needed information from the pixmap
    sBaseAddr = ( unsigned char * )GetPixBaseAddr( sDestPixMap );
    sRowBytes = ( *sDestPixMap )->rowBytes & 0x3FFF;
}
```


Since we are not using a Toolbox routine to draw the sprite, setting the clip rect of the destination port will not do our clipping for us. We will need to clip so that we do not write outside our graphic buffer and corrupt memory. So, the clip rect is saved so `drawSprite()` can use it. The destination pixel map, its base address, and its row bytes are saved as well.

What are the row bytes and base address? To understand these, we have to understand how images are stored in memory. An image is stored in a block of memory. The starting address of this block is the base address of the image. The image is stored pixel by pixel, left to right, top to bottom—just like we read text on a page. The row bytes defines the number of bytes in each row. With these two values we can calculate a specific pixel's address from its coordinates.

Why is the row bytes anded with 0x3FFF? QuickDraw uses the upper 2 bits of the row bytes field to store flags. If we do not strip those flag bits out, we can get drastically wrong row byte values.

Let's move on to `endSpriteDraw()`:

```
//
// endSpriteDraw()
//
// End the sprite draw sequence.
//

void endSpriteDraw( void )
{
}
```

There is no longer a need to restore the old clip region; `startSpriteDraw()` does not change it. We'll keep `endSpriteDraw()` around, because it may be useful later.

Now for the meat of this example—`drawSprite()`:

```
//
// drawSprite()
//
// Draw the sprite in the port.
//

void drawSprite( tSpriteInfo *inSpriteInfo, Point inWhere )
{
   Rect destRect;    // where we want to draw the sprite

   // calculate the destination rect
   destRect = inSpriteInfo->fSpriteRect;
   OffsetRect( &destRect, inWhere.h, inWhere.v );

   // determine if the sprite needs to be drawn at all
   if( destRect.top >= sClipRect.bottom ¦¦ destRect.bottom <= sClipRect.top ¦¦
        destRect.left >= sClipRect.right ¦¦ destRect.right <= sClipRect.left )
      {
         // no need to draw, goodbye
         return;
      }

   // determine if the sprite will be clipped
   if ( destRect.top < sClipRect.top ¦¦ destRect.bottom > sClipRect.bottom ¦¦
         destRect.left < sClipRect.left ¦¦ destRect.right > sClipRect.right )
      {
         // handle the clipped case
         renderSpriteClipped( inSpriteInfo, &destRect );
      }
   else
      {
         // handle the unclipped case
         renderSpriteUnclipped( inSpriteInfo, &destRect );
      }
}
```

This version of `drawSprite()` does three things. First, it calculates the destination rectangle of the sprite. Second, it determines if any of the sprite will be drawn. If not, `drawSprite()` returns immediately. Finally, it determines if the sprite needs to be drawn clipped or not. If the sprite does need to be drawn clipped, `drawSprite()` calls `renderSpriteClipped()`. If the sprite can be drawn without clipping, it calls `renderSpriteUnclipped()`. The unclipped version is faster than the clipped version, so calling the unclipped version whenever we can gives us the fastest overall sprite draw.

The unclipped case is simpler, so we'll examine it first.

```
//
// renderSpriteUnclipped()
//
// Draw the sprite, no clipping needed
//

void renderSpriteUnclipped( tSpriteInfo *inSpriteInfo, Rect *inDestRect )
{
   unsigned char *srcPtr;          // our current position in the shape
   unsigned char *destPtr;         // our current position in the destination
   unsigned long srcRowBytes;      // the row bytes of the source data
   unsigned char *srcBaseAddr;     // the base address of the source data
   unsigned char *srcRowStart;     // the start of the current row in the source
   unsigned char *destRowStart;    // the start of the current row in the destination
   unsigned long xCounter;         // a counter to scan the row
   unsigned long yCounter;         // a counter to scan the column

   // lock down the pixmap for the sprite
   LockPixels( inSpriteInfo->fSpritePix );

   // extract the needed information from the pixmap
   srcBaseAddr = ( unsigned char * )GetPixBaseAddr( inSpriteInfo->fSpritePix );
   srcRowBytes = ( *( inSpriteInfo->fSpritePix ) )->rowBytes & 0x3FFF;

   // determine the starting pointers
   srcRowStart = srcBaseAddr;
   destRowStart = sBaseAddr + inDestRect->top * sRowBytes + inDestRect->left;

   // loop through the data
   for( yCounter = inSpriteInfo->fSpriteRect.bottom; yCounter > 0; yCounter-- )
      {
      // set the scanning pointers
      srcPtr = srcRowStart;
      destPtr = destRowStart;

      // draw this row
      for( xCounter = inSpriteInfo->fSpriteRect.right; xCounter > 0; xCounter-- )
         {
            if ( *srcPtr != kClearColor )
               {
                  *destPtr = *srcPtr;
               }
```

```
            destPtr++;
            srcPtr++;
          }

        // move to the next row
        srcRowStart += srcRowBytes;
        destRowStart += sRowBytes;
      }

  // unlock the pixels
  UnlockPixels( inSpriteInfo->fSpritePix );
}
```

This routine breaks down into three sections: setup, loop, and shutdown. In setup, the code starts by locking the pixel map of the source data. It then extracts the base address and row bytes from that pixel map. The next block of code calculates the address of the upper left corner of the sprite in both the source and destination buffers.

The loop scans the sprite, from left to right, top to bottom. At each pixel in the sprite, it compares the sprite's pixel value with the clear color value. If the pixel is not clear, it copies the value into the destination buffer.

The only thing that the shutdown section does is unlock the sprite's pixel map buffer.

That was the easy version. Here is the harder, clipped case—`renderShapeClipped()`:

```
//
// renderSpriteClipped()
//
// Draw the sprite, deal with clipping.
//

void renderSpriteClipped( tSpriteInfo *inSpriteInfo, Rect *inDestRect )
{
  Rect srcRect;     // the rect that defines the clipped shape (source coordinates)
  Rect destRect;    // the rect that defines the clipped shape (destination coords)
  unsigned char *srcPtr;        // our current position in the shape
  unsigned char *destPtr;       // our current position in the destination
  unsigned long srcRowBytes;    // the row bytes of the source data
  unsigned char *srcBaseAddr;   // the base address of the source data
  unsigned char *srcRowStart;   // the start of the current row in the source
  unsigned char *destRowStart;  // the start of the current row in the destination
  unsigned long xCounter;       // a counter to scan the row
  unsigned long yCounter;       // a counter to scan the column

  // create a clipped rect in the coordinates of the sprite
  srcRect.left = inDestRect->left < sClipRect.left ? sClipRect.left - inDestRect ➥
->left : 0;
```

```
  srcRect.right = inDestRect->right > sClipRect.right ? sClipRect.right - ➥
inDestRect->left : inDestRect->right - inDestRect->left;
  srcRect.top = inDestRect->top < sClipRect.top ? sClipRect.top - inDestRect->top ➥
:0;
  srcRect.bottom = inDestRect->bottom > sClipRect.bottom ? sClipRect.bottom - ➥
inDestRect->top : inDestRect->bottom - inDestRect->top;

  // create the same rect in the coordinates of the destination
  destRect.left = inDestRect->left + srcRect.left;
  destRect.right = inDestRect->left + srcRect.right;
  destRect.top = inDestRect->top + srcRect.top;
  destRect.bottom = inDestRect->top + srcRect.bottom;

  // lock down the pixmap for the sprite
  LockPixels( inSpriteInfo->fSpritePix );

  // extract the needed information from the pixmap
  srcBaseAddr = ( unsigned char * )GetPixBaseAddr( inSpriteInfo->fSpritePix );
  srcRowBytes = ( *( inSpriteInfo->fSpritePix ) )->rowBytes & 0x3FFF;

  // determine the starting pointers
  srcRowStart = srcBaseAddr + srcRect.top * srcRowBytes + srcRect.left;
  destRowStart = sBaseAddr + destRect.top * sRowBytes + destRect.left;

  // loop through the data
  for( yCounter = srcRect.bottom - srcRect.top; yCounter > 0; yCounter-- )
    {
      // set the scanning pointers
      srcPtr = srcRowStart;
      destPtr = destRowStart;

      // draw this row
      for( xCounter = srcRect.right - srcRect.left; xCounter > 0; xCounter-- )
        {
          if ( *srcPtr != kClearColor )
            {
              *destPtr = *srcPtr;
            }
          destPtr++;
          srcPtr++;
        }

      // move to the next row
      srcRowStart += srcRowBytes;
      destRowStart += sRowBytes;
    }

  // unlock the pixels
  UnlockPixels( inSpriteInfo->fSpritePix );
}
```

This routine also breaks down into the same three parts. In fact, the real difference between these two routines, `renderShapeClipped()` and `renderShapeUnclipped()`, is the setup code.

This code relies on the fact that the intersection of two rectangles is another rectangle. The code intersects the clip rectangle and the sprite rectangle to determine the rectangle that the shape should be drawn in and sets the values of the loop so that it draws only that rectangle.

That is what the first two sections of code are doing: calculating the intersection rectangle. Actually, we're calculating two versions of the intersection rectangle: one relative to the destination and the other relative to the sprite. These rectangles are then used to determine the starting values for the source and destination pointers and as the limits for the loops to draw the data.

Analysis

Let's take a look at how well all of this performs. Visually, it looks about as fast as the `CopyMask()` version did. Let's see what the profiler says (see figure 3.4).

Demo.prof

Method: Detailed Timebase: PowerPC Saved at: 3:04:11 AM 3/7/95 Overhead: 485.136

Function Name	Count	Only	%	+Children	%	Average	Maximum	Minimum
doEventLoop	1	6683.912	12.5	52631.334	98.5	6683.912	6683.912	6683.912
doDemoFrame	1126	644.770	1.2	45947.422	86.0	0.573	1.402	0.195
blitToScreen	1126	25099.955	47.0	25099.955	47.0	22.291	23.612	21.184
restoreBackground	1126	12641.283	23.7	12641.283	23.7	11.227	44.288	10.383
drawSprite	11260	129.577	0.2	7237.797	13.5	0.012	0.723	0.000
renderSpriteUnclipped	11260	7114.351	13.3	7114.351	13.3	0.632	1.676	0.296
startupDemo	1	50.394	0.1	627.914	1.2	50.394	50.394	50.394
loadSprite	10	577.521	1.1	577.521	1.1	57.752	101.528	48.242
startSpriteDraw	1126	291.352	0.5	291.352	0.5	0.259	1.016	0.000
hideMenuBar	1	114.830	0.2	114.830	0.2	114.830	114.830	114.830
initMacToolboxes	1	48.165	0.1	48.165	0.1	48.165	48.165	48.165
moveObjects	1126	25.963	0.0	25.963	0.0	0.023	0.808	0.000
showMenuBar	1	15.819	0.0	15.819	0.0	15.819	15.819	15.819
shutdownDemo	1	0.499	0.0	5.480	0.0	0.499	0.499	0.499
disposeSprite	10	4.981	0.0	4.981	0.0	0.498	0.743	0.439
endSpriteDraw	1126	0.172	0.0	0.172	0.0	0.000	0.172	0.000

Figure 3.4 *Profiler results.*

This time the number we are looking for is in the +Children and % columns. This tells the total time for a routine, including all the routines it calls. In this case, we see that `drawSprite()` took 13.5 percent of the time. Compare that with 30.4 percent in the `CopyMask()` demo.

This method is one of my favorites for a couple of reasons:

- It's easy to add and remove art from the project.
- It's fast enough for many purposes.

However, for those times when you need as much speed as possible, there are ways to draw sprites even faster.

Example 4: Encoded Sprites

How can we improve on `drawSprite()` in example 3? What about example 3 is slow? The key is how example 3 transfers data from the sprite. It moved the data 1 byte at a time. However, the 68K Macintoshes can move 32 bits—4 bytes—at a time. Even better, the Power Macintoshes can move 64 bits—8 bytes—at a time. And each of these can move that data in the same time that it takes to move 1 byte! (Actually, there are some exceptions, but this statement is mostly true.)

The only problem is that we need to make sure we draw only the pixels we want. In the last example, we checked each pixel to make sure that we wanted to draw it. That won't work if we want to move the data in large chunks.

What if we considered a sprite as a series of instructions instead of as a pixel image? We would need a draw pixels instruction that told us how many pixels to draw and with what data to draw them. Also, we'd want a skip pixels instruction to skip clear data.

What operations does the sprite drawer do? It draws some pixels and skips others. What if we encoded the sprite as a series of instructions? What instructions would we need? We would need instructions to draw pixels and skip pixels. In addition, we would need an instruction to end the sprite. Finally, we would need instructions telling us to start the next row. Table 3.2 shows a summary.

Table 3.2
Sprite Draw Operations

Operation	Count	Data
draw pixels	number of pixels	pixel values
skip pixels	number of pixels	n/a
start new row	n/a	n/a
end shape	n/a	n/a

Let's assume that we have a sprite in this format. What advantages would a sprite-drawing routine have? First, it would not have to check every pixel to determine whether to draw or skip it. Second, it could move the draw pixels data in large chunks because it would know how many pixels to draw without checking each one. Overall, this could mean a big improvement.

Of course, there is a down side. The sprite needs to be encoded into this format. We will need to create a tool to translate the sprite from the picture format to this one. We'll get to that after we create the sprite drawer.

Before we get into the code, let's tighten up the data format. We are going to need the bounding rectangle for each sprite, so we'll store that at the beginning. Just after the rectangle, we'll store the sprite instructions. Table 3.3 shows the sprite instructions:

Table 3.3
Encoding Format

Op Code: 1 Byte	Count: 3 Bytes	Data: Variable—Padded to 4 Byte Boundary
0x00: end shape	unused—set to 0	n/a
0x01: start new row	number of byte to next start instruction	n/a
0x02: draw pixels	number of pixels	the pixels values
0x03: skip pixels	number to skip	n/a

Both PowerPC and 68K Macintoshes move data quickest when it is "properly aligned." For bytes, anywhere is properly aligned. For words (2 byte pairs), any even address is properly aligned. Finally, for longs (4 bytes), an address evenly divisible by four is properly aligned. This data format takes alignment into account. The memory manager always aligns an allocated block to a 4 byte boundary. If we keep our data sizes in even multiples of four, we can load the data quickly. Every instruction has a count attached, even if it is used. This means that we can load the op code and the count as a long—a 4 byte value. The data for the draw pixels operation is padded to a 4 byte boundary. That keeps upcoming data properly aligned.

We're going to need a tool to convert the sprite images from pictures to this encoded format. However, before we look at that, let's look at the drawing code. Like the previous examples, there are modifications to global.h, Demo.res, and the bulk in the `sprite` module.

global.h

There is only one modification to global.h. We define a new resource type:

```
#define kSpriteResType   'Sprt' // the res type of a sprite resource
```

We keep the definitions for the IDs of the sprites—`kNumberOne` through `kNumberTen`—but they apply to the new type.

Demo.res

We remove the 'PICT' versions of the sprites and replace them with 'Sprt' resources—the new encoded format.

sprite

As usual, we'll start with the `loadSprite()` function:

```
//
// loadSprite()
//
// Loads/allocates all the data for a sprite.
//

tSpriteInfo *loadSprite( signed short inSpriteResID )
{
   tSpriteInfo *newSprite;        // the new sprite data

   // create the sprite info record
   newSprite = ( tSpriteInfo * )NewPtr( sizeof( tSpriteInfo ) );

   // load the sprite
   newSprite->fSpriteData = GetResource( kSpriteResType, inSpriteResID );
   HNoPurge( newSprite->fSpriteData );
   if ( newSprite->fSpriteData == ( Handle )kNil )
      {
         // if it did not load, drop into the debugger -- real programs would have ➥
error checking
         Debugger();
      }

   // copy its bounds rect
   newSprite->fSpriteRect = **( ( Rect ** )( newSprite->fSpriteData ) );

   // return the new sprite!
   return newSprite;
}
```

This is simpler than the last version of `loadSprite()`; we don't have to draw the picture into a buffer. This code allocates the sprite information structure, loads the sprites, extracts the bounding rect, and returns the initialized structure.

Next, let's examine `disposeSprite()`:

```
//
// disposeSprite()
//
// Disposes of/releases all the memory used in a sprite
//

void disposeSprite( tSpriteInfo *inSpriteInfo )
{
   // dump the sprite resource
   ReleaseResource( inSpriteInfo->fSpriteData );
```

```
    // free the structure
    DisposePtr( ( Ptr )inSpriteInfo );
}
```

This hardly needs comment: Dump sprite, dump the information structure.

`startSpriteDraw()`, `endSprite()`, and `drawSprite()` are unchanged. However, `drawSpiteUnclipped()` and `drawSpriteClipped()` have been completely rewritten. `drawSpiteUnclipped()` is much simpler than `drawSpriteClipped()`, so we'll start with the unclipped case:

```
//
// renderSpriteUnclipped()
//
// Draw the sprite, no clipping needed
//

void renderSpriteUnclipped( tSpriteInfo *inSpriteInfo, Rect *inDestRect )
{
    unsigned char *rowStart;        // the pointer to the start of this row
    unsigned char *srcPtr;          // the current position in the sprite data
    unsigned char *destPtr;         // the current position in the destination pixmap
    unsigned long miscCounter;      // a counter for various purposes
    unsigned long tokenOp;          // the op code from the token
    unsigned long tokenData;        // the data from the token
    unsigned char exitFlag;         // should we exit from the loop?

    // determine characteristics about the pixmap
    rowStart = sBaseAddr + inDestRect->top * sRowBytes + inDestRect->left;

    // move to the right place in the shape (just past the size rect)
    srcPtr = ( unsigned char * )( ( *( inSpriteInfo->fSpriteData ) ) + sizeof( Rect ➥
) );

    // loop until we are done
    exitFlag = kFalse;
    while( !exitFlag )
        {
            // get a token
            tokenOp = ( *( ( unsigned long * )srcPtr ) ) >> 24;
            tokenData = ( *( ( unsigned long * )srcPtr ) ) & 0x00ffffff;
            srcPtr += sizeof( unsigned long );

            // depending on the token
            switch( tokenOp )
                {
                    case kDrawPixelsToken:
                        miscCounter = tokenData;

                        // move data in the biggest chunks we can find
```

```
#ifdef powerc
                // move in doubles while we can
                while( miscCounter >= sizeof( double ) )
                  {
                    *( ( double * )destPtr ) = *( ( double * )srcPtr );
                    destPtr += sizeof( double );
                    srcPtr += sizeof( double );
                    miscCounter -= sizeof( double );
                  }

                // move a long if we can
                if ( miscCounter >= sizeof( unsigned long ) )
                  {
                    *( ( unsigned long * )destPtr ) = *( ( unsigned long  * ) ➥
srcPtr );
                    destPtr += sizeof( unsigned long  );
                    srcPtr += sizeof( unsigned long  );
                    miscCounter -= sizeof( unsigned long  );
                  }
#else  // 68K
                // move in longs while we can
                while( miscCounter >= sizeof( unsigned long ) )
                  {
                    *( ( unsigned long * )destPtr ) = *( ( unsigned long * )srcPtr );
                    destPtr += sizeof( unsigned long );
                    srcPtr += sizeof( unsigned long );
                    miscCounter -= sizeof( unsigned long );
                  }
#endif

                // move a short if we can
                if ( miscCounter >= sizeof( unsigned short ) )
                  {
                    *( ( unsigned short * )destPtr ) = *( ( unsigned short  * ) ➥
srcPtr );
                    destPtr += sizeof( unsigned short  );
                    srcPtr += sizeof( unsigned short  );
                    miscCounter -= sizeof( unsigned short  );
                  }

                // move a char if we can
                if ( miscCounter >= sizeof( unsigned char ) )
                  {
                    *( ( unsigned char  * )destPtr ) = *( ( unsigned char  * ) ➥
srcPtr );
                    destPtr += sizeof( unsigned char  );
                    srcPtr += sizeof( unsigned char  );
                    miscCounter -= sizeof( unsigned char );
                  }

                // adjust for the padding
                srcPtr += ( ( tokenData & 3L ) == 0 ) ? 0 : ( 4 - ( tokenData & 3L ➥
) );
```

```
                break;

            case kSkipPixelsToken:
                destPtr += tokenData;
                break;

            case kLineStartToken:
                // set up the destination pointer
                destPtr = rowStart;
                rowStart += sRowBytes;
                break;

            case kEndShapeToken:
                // signal a loop exit
                exitFlag = kTrue;
                break;

            default:
                // we should never get here
                Debugger();
                break;
        }
    }
}
```

As long as this code is, it's very straightforward. It starts by determining where we will be putting data, `rowStart`, and where we will get it, `srcPtr`. Then the code enters a loop. Each time through the loop, the code gets an instruction and breaks it up into the 1 byte operation code and the 3 byte count. It then switches based on the operation code.

If the code is draw pixels, the code copies the number of bytes indicated in the count to the destination. There are two sets of code surrounded by `#ifdef powerc`. The first code is for PowerPC. The second set of code is for the 68K. What's the difference? Although the PowerPC's integer registers are 32 bits wide (4 bytes), it's floating point registers are 64 bits wide (8 bytes). The bus is also 64 bits wide, so the PowerPC can load 8 bytes into a float in one bus cycle. Therefore, the code transfers as many 8 byte chunks as it can, then finishes up with 4, 2, and 1 byte chunks. The 68K's registers and bus are 32 bits wide, so the code moves as many 4 byte chunks and finishes up with 2 and 1 byte chunks. After all the pixels have been drawn, the code skips the bytes padding to a 4 byte boundary (if any).

To handle skip pixels, the code just adds the number of pixels to skip to the destination pointer. A line start is simple as well. The code makes a copy of the current line start to use as a destination pointer. It then calculates the next line start. The end shape instruction is the easiest to handle: All it does is set the loop exit flag. Finally, if the token is unknown, the code drops into the debugger—something is seriously wrong with the data.

There is one additional thing to note about `drawSprite()`: the order of the case statements. A switch is implemented as a series of `if-elses`. Therefore, when speed is very important, you should place your cases in descending order of frequency. That's what this code does: Draws and skips are very common, line starts are less common, and there is only one end sprite for each sprite.

That was the easy version; the clipped case is more complex:

```
//
// renderSpriteClipped()
//
// Draw the spite, deal with clipping.
//

void renderSpriteClipped( tSpriteInfo *inSpriteInfo, Rect *inDestRect )
{
   Rect clipRect;                // the rect that defines the clipped shape
   unsigned char *rowStart;      // the pointer to the start of this row
   unsigned char *srcPtr;        // the current position in the sprite data
   unsigned char *destPtr;       // the current position in the destination pixmap
   unsigned long miscCounter;    // a counter for various purposes
   unsigned long extraCounter;   // a counter for right clippling purposes (how ➥
much extra was there?)
   unsigned long tokenOp;        // the op code from the token
   unsigned long tokenData;      // the data from the token
   unsigned char exitFlag;       // should we exit from the loop?
   unsigned long yCount;         // how many lines down in the shape are we?
   unsigned long xCount;         // where are we in this line?

   // create a clipped rect in the coordinates of the sprite
   clipRect.left = inDestRect->left < sClipRect.left ? sClipRect.left - inDestRect ➥
->left : 0;
   clipRect.right = inDestRect->right > sClipRect.right ? sClipRect.right - ➥
inDestRect->left : inDestRect->right - inDestRect->left;
   clipRect.top = inDestRect->top < sClipRect.top ? sClipRect.top - inDestRect ➥
->top : 0;
   clipRect.bottom = inDestRect->bottom > sClipRect.bottom ? sClipRect.bottom - ➥
inDestRect->top : inDestRect->bottom - inDestRect->top;

   // set up the counters
   yCount = 0;

   // determine characteristics about the pixmap
   rowStart = sBaseAddr + inDestRect->top * sRowBytes + inDestRect->left;

   // move to the right place in the shape (just past the size rect)
   srcPtr = ( unsigned char * )( ( *( inSpriteInfo->fSpriteData ) ) + sizeof( Rect ➥
) );
```

```
   // loop until we are done
   exitFlag = kFalse;
   while( !exitFlag )
      {
         // get a token
         tokenOp = ( *( ( unsigned long * )srcPtr ) ) >> 24;
         tokenData = ( *( ( unsigned long * )srcPtr ) ) & 0x00ffffff;
         srcPtr += sizeof( unsigned long );

         // depending on the token
         switch( tokenOp )
            {
               case kDrawPixelsToken:
                  miscCounter = tokenData;
                  extraCounter = 0;

                  // if we need to, clip to the left
                  if( xCount < clipRect.left )
                     {
                        // if this run does not appear at all, don't draw it
                        if ( miscCounter < clipRect.left - xCount )
                           {
                              destPtr += miscCounter;
                              srcPtr += miscCounter;
                              srcPtr += ( ( tokenData & 3L ) == 0 ) ? 0 : ( 4 - ➥
( tokenData & 3L ) );
                              xCount += miscCounter;
                              break;
                           }
                        else
                           {
                              // if it does, skip to where we can draw
                              miscCounter -= clipRect.left - xCount;
                              destPtr += clipRect.left - xCount;
                              srcPtr += clipRect.left - xCount;
                              xCount += clipRect.left - xCount;
                           }
                     }

                  // if we need to, clip to the right
                  if ( xCount + miscCounter > clipRect.right )
                     {
                        // if this run does not appear at all, skip it
                        if ( xCount > clipRect.right )
                           {
                              destPtr += miscCounter;
                              srcPtr += miscCounter;
                              srcPtr += ( ( tokenData & 3L ) == 0 ) ? 0 : ( 4 - ➥
( tokenData & 3L ) );
```

```
                    xCount += miscCounter;
                    break;
                  }
                else
                  {
                    // if it does, set up to draw what we can
                    extraCounter = miscCounter;
                    miscCounter -= ( xCount + miscCounter ) - clipRect.right;
                    extraCounter -= miscCounter;
                  }
              }

            // adjust xCount for the run
            xCount += miscCounter;

            // move data in the biggest chunks we can find
#ifdef powerc
            // move in doubles while we can
            while( miscCounter >= sizeof( double ) )
              {
                *( ( double * )destPtr ) = *( ( double * )srcPtr );
                destPtr += sizeof( double );
                srcPtr += sizeof( double );
                miscCounter -= sizeof( double );
              }

            // move a long if we can
            if ( miscCounter >= sizeof( unsigned long ) )
              {
                *( ( unsigned long * )destPtr ) = *( ( unsigned long  * ) ➥
srcPtr );
                destPtr += sizeof( unsigned long  );
                srcPtr += sizeof( unsigned long  );
                miscCounter -= sizeof( unsigned long  );
              }
#else
            // move in longs while we can
            while( miscCounter >= sizeof( unsigned long ) )
              {
                *( ( unsigned long * )destPtr ) = *( ( unsigned long * )srcPtr );
                destPtr += sizeof( unsigned long );
                srcPtr += sizeof( unsigned long );
                miscCounter -= sizeof( unsigned long );
              }
#endif

            // move a short if we can
            if ( miscCounter >= sizeof( unsigned short ) )
              {
```

```
                    *( ( unsigned short * )destPtr ) = *( ( unsigned short  * ) ➥
srcPtr );
                    destPtr += sizeof( unsigned short  );
                    srcPtr += sizeof( unsigned short  );
                    miscCounter -= sizeof( unsigned short  );
                  }

                // move a char if we can
                if ( miscCounter >= sizeof( unsigned char ) )
                  {
                    *( ( unsigned char  * )destPtr ) = *( ( unsigned char  * ) ➥
srcPtr );
                    destPtr += sizeof( unsigned char  );
                    srcPtr += sizeof( unsigned char  );
                    miscCounter -= sizeof( unsigned char );
                  }

                // adjust for right clipping
                destPtr += extraCounter;
                srcPtr += extraCounter;
                xCount += extraCounter;

                // adjust for the padding
                srcPtr += ( ( tokenData & 3L ) == 0 ) ? 0 : ( 4 - ( tokenData & 3L ➥
) );
                break;

              case kSkipPixelsToken:
                destPtr += tokenData;
                xCount += tokenData;
                break;

              case kLineStartToken:
                // if this line is above the clip rect, skip to the next line
                if( yCount < clipRect.top )
                  {
                    srcPtr += tokenData;
                  }

                // set up the destination pointer
                destPtr = rowStart;
                rowStart += sRowBytes;

                // move the yCounter
                yCount++;

                // reset the xCounter
                xCount = 0;
```

```
                // if we have hit the bottom clip, exit the loop
                if ( yCount > clipRect.bottom )
                {
                    exitFlag = kTrue;
                }
                break;

            case kEndShapeToken:
                // signal a loop exit
                exitFlag = kTrue;
                break;

            default:
                // we should never get here
                Debugger();
                break;
        }
    }
}
```

This code is an extension of the unclipped code. We'll focus on the differences. There are four sides to clip to and each is handled somewhat independently.

We'll start with the top clip. Remember from the definition that the line start instruction has a count that tells how many bytes are in the current line. This is there so the code can skip from line to line quickly. The code to handle a line start checks to see if the line is above the clipping rectangle. If it is, the source pointer is moved to the next line start, skipping the line.

If you look further down in the line start code, you'll see the bottom clip. That last `if` checks if the code has hit the bottom of the clip rectangle. If it has, the exit flag is set and the sprite draw ends.

The left and right clippings are handled in the draw pixels section. There are three cases: The pixel run should not be drawn at all, the entire pixel run should be drawn, or part of the run should be drawn. First, the code checks to see if the run starts before the left side of the clipping rectangle. If it does, the code checks to see if it ends before the left side as well. If so, the run is skipped. If not, the starting location and run length are adjusted to start at the left clipping edge. The code then checks the right side to see if the run extends past the right edge of the clipping rectangle. If it does, the code checks to see if the run starts after the right edge. If so, it skips the pixel run. If not, it shortens the run so that it will end at the right clipping edge. Then the run is drawn. Afterwards, the source pointer is adjusted, if needed, to the real end of the run.

Encoding Tool

Before we move on and analyze this sprite drawer, let's look at the tool to encode the sprites. This little tool works by taking all the pictures in its resource fork and converting them to 'Sprt' resources, which it writes into its fork. Admittedly, there is a lot of room for improvement, but this gets the basic job done.

The `encoder` is based on the same code that we've been using for all of the examples. In fact, the resource file and global.h are the same ones used in example 3. There are only two modules: `main` and `encoder`. `main` is a reduced version of the `main` we are used to. `encoder` is completely new.

main

Let's start by quickly looking at `main()`:

```
//
// main()
//   This is where it all begins.
//

void main( void )
{
   // fire up the Toolboxes
   initMacToolboxes();

   // get the color table
   gAppColorTable = GetCTable( kAppColorTableResID );

   // give the encoding code a chance to set up
   startupEncoder();

   // handle events until the user signals a quit
   doEncode();

   // give the encoding a chance to shut down
   shutdownEncoder();
}
```

The code starts with a call to `initMacToolboxes()`. We are going to skip over `initMacToolboxes()` because it is the same one from example 1. `main` then gets the color table. Since we are not going to display the graphics on the screen, there is no code to create a palette. The next three calls initialize, execute, and shut down the `encoder` module.

encoder

With a `main` module that small, all the magic must occur in `encoder`. But before we get into the gritty code, let's look at `startupEncoder()`:

```
//
// startupEncoder()
//
// Create the buffer for the encode routine.
//

void startupEncoder( void )
{
   printf( "Allocating offscreen\n" );

   // create the off-screen GWorld
   NewGWorld( &gOffscreenBuffer, 8, &gOffscreenRect, gAppColorTable, ( GDHandle ➥
)kNil, keepLocal );

   // get the pixel map and rect
   gOffscreenPixels = GetGWorldPixMap( gOffscreenBuffer );
}
```

This is pretty simple, but there are a couple of points worth noting. First, note the `printf` at the top of the function. This project uses CodeWarrior's SIOUX library, which provides support for the standard C and C++ library functions that expect a terminal (how primitive!). The terminal automatically opens the first time a function sends output to or asks for input from the terminal. After the rest of the program exits, the terminal remains until the user quits. I am not sure how the internals of SIOUX work; they sound pretty ugly. However, SIOUX works great for what we want here: a way to note progress.

After printing the text, `startupEncoder()` creates an off-screen and gets its pixmap.

Let's also sneak a look at `shutdownEncoder()`:

```
//
// shutdownEncoder()
//
// Release all the memory used by the encoder.
//

void shutdownEncoder( void )
{
   printf( "Disposing offscreen\n" );

   // dump the off-screen GWorld
   DisposeGWorld( gOffscreenBuffer );
}
```

Just as you may have guessed, all `shutdownEncoder()` does is dump the off-screen GWorld (and print a line to the terminal).

The job of encoding the data is divided between two functions: `doEncode()` and `encodeRect()`. `doEncode()` performs the high-level functions of loading, drawing, and saving the data. `encodeRect()` actually builds the encoded data. We'll start with `doEncode()`:

```
//
// doEncode()
//
// Encode all the 'PICT's from the resource fork.
//

void doEncode( void )
{
  CGrafPtr oldPort;              // the port in place when we started
  GDHandle oldGDevice;           // the GDevice in place when we started
  unsigned short numResources;   // the number of a type or resource (used for both ➥
'PICT's and 'Sprt's)
  unsigned short resCounter;     // a counter to scan resources
  Handle spriteResource;         // a handle to a sprite resource
  PicHandle pictResource;        // a handle to a 'PICT' resource
  signed short resID;            // the ID# of the resource that we are examining
  ResType resType;               // the type of resource that we are examining ➥
(needed for GetResInfo())
  Str255 resName;                // the name of the resource that we are examining

  // save the current port and device
  GetGWorld( &oldPort, &oldGDevice );

  // set the off-screen
  LockPixels( gOffscreenPixels );
  SetGWorld( gOffscreenBuffer, ( GDHandle )kNil );

  // remove all the sprite resources
  printf( "Removing old sprites - " );

  // determine the number of sprite resources
  numResources = Count1Resources( kSpriteResType );

  // if there are any,
  if ( numResources != 0 )
    {
      // get each one,
      for( resCounter = 1; resCounter <= numResources; resCounter++ )
        {
          // and delete it
          spriteResource = Get1IndResource( kSpriteResType, resCounter );
          RmveResource( spriteResource );
          DisposeHandle( spriteResource );
        }
    }
```

```
// let the user know
printf( "Done\n" );

// determine the number of 'PICT' resources
numResources = Count1Resources( 'PICT' );

// if there are not any,
if ( numResources == 0 )
  {
    // tell the user
    printf( "There are no picts to encode!\n" );
  }
else
  {
    // get each one,
    for( resCounter = 1; resCounter <= numResources; resCounter++ )
      {
        // load the 'PICT'
        pictResource = ( PicHandle )Get1IndResource( 'PICT', resCounter );
        HLock( ( Handle )pictResource );

        // determine its ID and name
        GetResInfo( ( Handle )pictResource, &resID, &resType, resName );

        // let the user know what is going on
        printf( "\tProcessing PICT #%d - ", resID );

        // erase the background
        EraseRect( &( ( *pictResource )->picFrame ) );

        // draw the picture
        DrawPicture( pictResource, &( ( *pictResource )->picFrame ) );

        // encode the data
        spriteResource = encodeRect( &( ( *pictResource )->picFrame ) );

        // add the new resource to the file
        AddResource( spriteResource, kSpriteResType, resID, resName );
        WriteResource( spriteResource );
        ReleaseResource( spriteResource );

        // release the 'PICT'
        ReleaseResource( ( Handle )pictResource );

        // let the user know
        printf( "Done\n" );
      }
  }
```

```
    // restore the previous port and device
    SetGWorld( oldPort, oldGDevice );

    // unlock the pixels
    UnlockPixels( gOffscreenPixels );
}
```

The code is long, but simple. `doEncode()` starts by setting the off-screen GWorld. If you add a resource to a file that already has a resource with the same type and ID, you'll create an error in the file. Therefore, before starting to encode and save sprites, the code removes all the old sprites. Next, it starts looping through all the pictures. For each picture, the code loads it, draws it, and passes it to `encodeRect()`. `encodeRect()` returns the encoded sprite, which is written as a resource of type 'Sprt,' and the ID of the picture from which it came. After processing all the pictures, the code restores the original port and exits.

The last and most complex routine is `encodeRect()`:

```
//
// encodeRect()
//
// Encode the data in the given rect into a sprite.
//

Handle encodeRect( Rect *inRect )
{
    Handle shapeHandle;           // the shape that we create in the function
    unsigned short shapeHeight;   // the height of the shape
    unsigned short shapeWidth;    // the width of the shape
    unsigned char *destPtr;       // the current position in the shape
    unsigned char *srcPtr;        // the current position in the source graphic data
    unsigned char *baseAddr;      // the base address of the source pixmap
    unsigned long rowBytes;       // the row bytes of the source pixmap
    unsigned char *rowStart;      // the start of the current row in the pixmap
    unsigned long yCounter;       // a counter to scan the shape vertically
    unsigned long xCounter;       // a counter to scan the shape horizontally
    unsigned char drawRunFlag;    // are we in a draw pixels run?
    unsigned char skipRunFlag;    // are we in a skip pixels run?
    unsigned char *lineStartPtr;  // where is the line start token for this line
    unsigned char *runTokenPtr;   // where is the token for the current run
    unsigned long runCounter;     // how long is the current run?

    // determine the width and height of the shape (we use these values a lot)
    shapeHeight = inRect->bottom - inRect->top;
    shapeWidth = inRect->right - inRect->left;

    // create a handle big enough for the worst case encoding
    // ( 8 bytes/pixel + 4 bytes/row + 4 bytes/shape (end token) + 8 bytes/shape ➥
(rect) )
```

```
shapeHandle = NewHandle( 8 * shapeHeight * shapeWidth + 4 * shapeHeight + 4 + 8 );

// make sure that we got the buffer
if( shapeHandle == ( Handle )kNil )
   {
      printf( "Error - Could not allocate buffer to build shape\n");
      return shapeHandle;
   }

// lock the handle and get the pointer
HLock( shapeHandle );
destPtr = ( unsigned char * )( *shapeHandle );

// store the shape rect
*( ( Rect * )destPtr ) = *inRect;
destPtr += sizeof( Rect );

// get the location of the source data
baseAddr = ( unsigned char * )GetPixBaseAddr( gOffscreenPixels );
rowBytes = ( *gOffscreenPixels )->rowBytes & 0x3fff;
rowStart = baseAddr + rowBytes * inRect->top + inRect->left;

// scan the shape row by row
for( yCounter = 0; yCounter < shapeHeight; yCounter++ )
   {
      // store the location of this line start
      lineStartPtr = destPtr;
      destPtr += sizeof( unsigned long );

      // at the beginning of each row we are not in any run
      drawRunFlag = kFalse;
      skipRunFlag = kFalse;

      // move to the start of the row
      srcPtr = rowStart;

      // scan each row of the shape
      for( xCounter = 0; xCounter < shapeWidth; xCounter++ )
         {
            // is this pixel clear?
            if ( *srcPtr == kClearColorIndex )
               {
                  // are we in a draw run?
                  if ( drawRunFlag )
                     {
                        // end the draw run
                        drawRunFlag = kFalse;

                        // create the draw token
```

```
                        *( ( unsigned long * )runTokenPtr ) = ( kDrawPixelsToken ➥
<< 24 ) + runCounter;

                        // pad to a mulitple of four
                        *( ( unsigned long * )destPtr ) = 0L;
                        destPtr += ( ( runCounter & 3L ) == 0 ) ? 0 : ( 4 - ➥
( runCounter & 3L ) );
                     }

                  // are we in a skip run
                  if ( skipRunFlag )
                     {
                        // continue it
                        runCounter++;
                     }
                  else
                     {
                        // start one
                        skipRunFlag = kTrue;
                        runCounter = 1;
                     }
               }
            else
               {
                  // are we in a skip run
                  if ( skipRunFlag )
                     {
                        // end the skip run
                        skipRunFlag = kFalse;

                        // create the skip token
                        *( ( unsigned long * )destPtr ) = ( kSkipPixelsToken << 24 ) ➥
+ runCounter;
                        destPtr += sizeof( unsigned long );
                     }

                  // are we in a draw run
                  if ( drawRunFlag )
                     {
                        // continue it
                        runCounter++;

                        // copy the pixel
                        *destPtr = *srcPtr;
                        destPtr++;
                     }
                  else
                     {
                        // start one
                        drawRunFlag = kTrue;
                        runCounter = 1;
```

```
                    // save the location of the token (so we can fill it in later)
                    runTokenPtr = destPtr;
                    destPtr += sizeof( unsigned long );

                    // copy the pixel
                    *destPtr = *srcPtr;
                    destPtr++;
                }
            }

            // move to the next byte
            srcPtr++;
        }

        // are we in a draw run
        if( drawRunFlag )
            {
            // end the draw run
            drawRunFlag = kFalse;

            // create the draw token
            *( ( unsigned long * )runTokenPtr ) = ( kDrawPixelsToken << 24 ) + ➥
runCounter;

            // pad to a multiple of four
            *( ( unsigned long * )destPtr ) = 0L;
            destPtr += ( ( runCounter & 3L ) == 0 ) ? 0 : ( 4 - ( runCounter & 3L ➥
) );
        }

        // create the line start token
        *( ( unsigned long * )lineStartPtr ) = ( kLineStartToken << 24 ) + ( ➥
destPtr - ( lineStartPtr + 4 ) );

        // move the row start to the next row
        rowStart += rowBytes;
    }

    // create the end of shape token
    *( ( unsigned long * )destPtr ) = kEndShapeToken << 24;
    destPtr += sizeof( unsigned long );

    // Unlock the handle
    HUnlock( shapeHandle );

    // Resize the handle to match the real size of the shape
    SetHandleSize( shapeHandle, destPtr - ( unsigned char * )( *shapeHandle ) );

    // return the handle
    return shapeHandle;
}
```

Before we get into specifics, let's think about what we need to do here. We have to encode the shape into a series of pixels to draw and pixels to skip. We're going to assume that pixels that match the clear color, white, were not drawn by the picture and should be clear. Remember, think of the clear color as a marker for which pixels are not part of the shape. Therefore, whatever color is being used as a clear color cannot appear in the shape itself. If you want white in your sprite, you will need to change the clear color and paint the background with it before you draw the picture.

Let's talk about the code. The code starts by allocating a buffer in which to store the sprite. The formula to determine the size assumes the worst case: alternating runs of 1 byte clear and draw runs. Most sprites will be much smaller. The code stores the bounding rectangle for the sprite. Next, the code prepares to scan the image pixel by pixel, in a left-to-right, top-to-bottom order.

Each time it starts a new line, the code leaves space for the line token and remembers where it is. It cannot actually create the line token until the line has been encoded—it needs the size of the line first. The code then scans the line. There are four possible conditions at each pixel and a different action for each (see table 3.4).

Table 3.4
Encoder Options

Last Pixel	This Pixel	Action
clear	clear	continue skip run
clear	not clear	end skip run, start draw run
not clear	clear	end draw run, start skip run
not clear	not clear	continue draw run

Implementing table 3.4 is what the large `if-else` statements do in the middle of the code. One other thing worth noting is that a line always ends with the last draw run. There is no need to skip after that.

After all of the pixels have been scanned, the end shape token is added to the end. The handle is resized to match the real size of the sprite. Finally, the handle is returned.

Analysis

As usual, let's run the profiler on this. Figure 3.5 shows the results.

Demo.prof

Method: Detailed Timebase: PowerPC Saved at: 12:15:24 AM 3/10/95 Overhead: 211.842

Function Name	Count	Only	%	+Children	%	Average	Maximum	Minimum
doEventLoop	1	5229.150	18.6	27811.904	98.7	5229.150	5229.150	5229.150
doDemoFrame	627	325.299	1.2	22582.754	80.2	0.519	1.551	0.190
blitToScreen	627	13809.875	49.0	13809.875	49.0	22.025	22.066	21.191
restoreBackground	627	6678.438	23.7	6678.438	23.7	10.651	44.373	9.843
drawSprite	6270	61.099	0.2	1592.226	5.7	0.010	0.587	0.000
renderSpriteUnclipped	6270	1531.779	5.4	1531.779	5.4	0.244	1.291	0.000
startSpriteDraw	627	170.257	0.6	170.257	0.6	0.272	1.077	0.089
startupDemo	1	49.790	0.2	146.233	0.5	49.790	49.790	49.790
hideMenuBar	1	132.745	0.5	132.745	0.5	132.745	132.745	132.745
loadSprite	10	96.443	0.3	96.443	0.3	9.644	40.987	5.257
initMacToolboxes	1	45.764	0.2	45.764	0.2	45.764	45.764	45.764
shutdownDemo	1	0.622	0.0	16.449	0.1	0.622	0.622	0.622
disposeSprite	10	15.826	0.1	15.826	0.1	1.583	2.012	1.463
showMenuBar	1	15.187	0.1	15.187	0.1	15.187	15.187	15.187
moveObjects	627	5.846	0.0	5.846	0.0	0.009	0.416	0.000
endSpriteDraw	627	0.161	0.0	0.161	0.0	0.000	0.161	0.000

Figure 3.5 *Profiler results.*

Here we see that `drawSprite()` and all the routines that it calls total only 5.7 percent of the time the application uses. This is down from 13.5 percent from the last example and way down from where we started—56.2 percent. What does this mean? Drawing sprites is no longer the slow part of this program. The new champion time-user is `blitToScreen()` at 49 percent.

In addition to being fast, the encoded draw has some other pluses. The first is ease of use: Just load the sprite and draw—all the data sits in the resource. Another plus is the room for expansion. We used four operation codes; there is room for another 252 codes. You could add codes to filter colors, darken, lighten, and so on. These pluses are somewhat balanced by the need to encode the data. You can't just drop a picture into the game; you have to encode first. Despite this, the encoded sprite draw is my favorite because of its speed and expandability.

Now that we have improved sprites so much, we are going to focus our attention elsewhere: `blitToScreen()`.

Example 5: Dirty Rectangle

Lets take a look at the current version of `blitToScreen()`:

```
//
// blitToScreen()
//
// This routine updates the screen from the off-screen buffer.
//

void blitToScreen( void )
{
   CGrafPtr oldPort;        // the graf port that is in place when we are called
   GDHandle oldDevice;      // the GDevice that is in place when we are called
```

```
    // save the current port and GDevice
    GetGWorld( &oldPort, &oldDevice );

    // set the drawing environment to the screen
    SetGWorld( ( CWindowPtr )gMainWindow, GetMainDevice() );

    // make sure that the fore and back colors are correct to prevent colorize mode
    ForeColor( blackColor );
    BackColor( whiteColor );

    // Copy the screen's color table seed into the source pixmap.
    // This will minimize CopyBits' setup time.
    ( *( ( *gOffscreenPixels )->pmTable ) )->ctSeed = ( *( ( *( ( *( GetGDevice() ) ➥
) )->gdPMap ) )->pmTable ) )->ctSeed;

    // copy the buffer to the screen
    CopyBits( ( BitMap * )( *gOffscreenPixels ), &( gMainWindow->portBits ),
         &gOffscreenRect, &( gMainWindow->portRect ),
         srcCopy, ( RgnHandle )kNil );

    // restore the current port and GDevice
    SetGWorld( oldPort, oldDevice );
}
```

What about this one routine is taking almost half of the demo's time? The answer to that is the call to `CopyBits()`. That call is moving the contents of the entire off-screen to the screen. At 640 x 480 pixels, each of which is a byte, that is 300K every frame. Even on a fast machine, that is a lot of memory to move.

What does that move do for us? Well, it erases objects' old positions and draws them in their new ones. Now for the key question: Can we get the same effect without moving all of the memory?

Let's try this approach: Start with an empty rectangle, which we'll call the dirty rectangle. Each time we move an object, let's add the rectangle where the object was, and the rectangle of where the object is now, to the dirty rectangle. After we move all of the objects, the bounds of the dirty rectangle will include everything that needs to be copied to the screen. Then we can copy the dirty rectangle instead of the entire screen, and hopefully save some time.

In fact, we can be even more sophisticated. Not only do we not have to copy the entire buffer to the screen, but we do not have to draw the entire off-screen. If nothing has moved in an area, nothing should have changed. Therefore, we can clip our drawing to this rectangle as well.

We're going to add a new module to the demo: `update`. `update` keeps track of what needs to be updated to the screen. Whenever we move something, we tell `update` where it was and where it is now. When we want to update the screen, we ask `update` for the rectangle to copy. To help see how this works, table 3.5 shows some new controls for `demo`.

Table 3.5
New Demo Commands

Key	Meaning
Shift + a number	Freeze/unfreeze the corresponding disk.
U	Turn show update rects on/off.
S	Turn step mode on/off. In step mode, pressing space will move one frame at a time.

One control has been removed: clip mode. We need to use clipping for the drawing process now, so we cannot spare it for clip mode.

If you want to see exactly what we are talking about here, run the demo and press U. What you'll see is the rectangle we are using to draw and copy with. Try freezing various numbers and see how it affects the update rectangle.

`update`

The `update` module has three entry points:

- `clearUpdate()`—Resets the update rectangle to empty.
- `addRectToUpdate()`—Adds a rectangle to the update.
- `getUpdateRect()`—Gives us the accumulated update rectangle.

Let's start with `clearUpdate()`:

```
//
// clearUpdate()
//
// Resets the update to none.
//

void clearUpdate( void )
{
   // note that there is no dirty rect
   sIsDirty = kFalse;
}
```

All clear update does is note that there is no update rectangle. Pretty simple.

Next is `addRectToUpdate()`:

```
//
// addRectToUpdate()
//
// Adds a rect to the update area.
//

void addRectToUpdate( Rect *inUpdateRect )
{
   // if there is an update rect already
   if( sIsDirty )
      {
         // add this one to it
         sDirtyRect.top = sDirtyRect.top < inUpdateRect->top ? sDirtyRect.top : ➥
inUpdateRect->top;
         sDirtyRect.left = sDirtyRect.left < inUpdateRect->left ? sDirtyRect.left : ➥
inUpdateRect->left;
         sDirtyRect.bottom = sDirtyRect.bottom > inUpdateRect->bottom ? ➥
sDirtyRect.bottom : inUpdateRect->bottom;
         sDirtyRect.right = sDirtyRect.right > inUpdateRect->right ? ➥
sDirtyRect.right : inUpdateRect->right;

         // adjust the rect to 32-bit bounds
         adjustRect( &sDirtyRect );
      }
   else
      {
         // make this the dirty rect
         sDirtyRect = *inUpdateRect;

         // note that there is a dirty rect
         sIsDirty = kTrue;

         // adjust the rect to 32-bit bounds
         adjustRect( &sDirtyRect );
      }
}
```

This routine either extends the current update rectangle around the input one or, if there is no update rectangle, makes the input rectangle the current update rectangle.

What is the call to `adjustRect()`? Remember that `CopyBits()` works fastest when it is copying from 32-bit bounds. `adjustRect()` moves the bounds of a rectangle to the nearest 32-bit bounds. Here it is:

```
//
// adjustRect()
//
// Move the left and right sides to 32-bit bounds. This will speed up CopyBits().
//

void adjustRect( Rect *ioRect )
{
   // expand the rect so its horizontal sides are on 32-bit bounds
   ioRect->left = ioRect->left & 0xfffC;
   ioRect->right = ( ioRect->right + 3 ) & 0xfffC;
}
```

Finally, we need to be able to get the current update rect so we can draw and copy with it. Here is `getUpdateRect()`:

```
//
// getUpdateRect()
//
// Get the dirty rect. If there is one the return is TRUE, otherwise FALSE.
//

unsigned char getUpdateRect( Rect *outUpdateRect )
{
   // if there is an update rect, send it out
   if ( sIsDirty )
      {
         *outUpdateRect = sDirtyRect;
      }

   // return the dirty flag
   return sIsDirty;
}
```

The one thing worth noting here is the Boolean return value. If `getUpdateRect()` returns FALSE, there is no update rectangle.

`demo`

Adding the update rectangle to `demo` requires some changes. First of all, we need to add a field to the object structure:

```
   unsigned char fWasVisible;    // was this object visible last time?
```

Now that we need to note where we draw and where we erase, we need to know if we drew an object in the last frame. If we did not, adding the update rectangle for the erase would be wasteful.

`demoKey()` has changes to add the new commands, but the code is simple enough that we can skip over it. Also, `blitToScreen()` now takes a rectangle that defines what to copy. Again, the code is simple enough to skip.

Let's look at where we keep track of the changes:

```
//
// ()
//
// Move all the objects.
//

void moveObjects( void )
{
   unsigned short indexCounter;     // a counter to scan all the objects
   Rect objectRect;                 // the rectangle that bounds the object

   // clear the update area
   clearUpdate();

   // consider each object
   for( indexCounter = 0; indexCounter < kMaxObjects; indexCounter++ )
      {
         // if this object has no velocity, we can ignore it
         if ( sObjects[ indexCounter ].fDeltaX == 0 && sObjects[ indexCounter ] ➥
.fDeltaY == 0 )
            {
               continue;
            }

         // if the object were visible last time
         if ( sObjects[ indexCounter ].fWasVisible )
            {
               // determine the object's rect
               objectRect = ( sObjects[ indexCounter ].fSpriteInfo )->fSpriteRect;
               objectRect.left += sObjects[ indexCounter ].fIntXPos;
               objectRect.top += sObjects[ indexCounter ].fIntYPos;
               objectRect.right += sObjects[ indexCounter ].fIntXPos;
               objectRect.bottom += sObjects[ indexCounter ].fIntYPos;

               // add the old rect to the update (so it will be erased)
               addRectToUpdate( &objectRect );
            }

         // apply the velocity
         sObjects[ indexCounter ].fXPos += sObjects[ indexCounter ].fDeltaX;
         sObjects[ indexCounter ].fYPos += sObjects[ indexCounter ].fDeltaY;

         // if the object has hit an edge, bounce it
         if( sObjects[ indexCounter ].fXPos < ( gOffscreenRect.left << 16 ) )
            {
```

```
            sObjects[ indexCounter ].fDeltaX *= -1;
            sObjects[ indexCounter ].fXPos += ( gOffscreenRect.left << 16 ) - ➥
sObjects[ indexCounter ].fXPos;
          }
        if( sObjects[ indexCounter ].fYPos < ( gOffscreenRect.top << 16 ) )
          {
            sObjects[ indexCounter ].fDeltaY *= -1;
            sObjects[ indexCounter ].fYPos += ( gOffscreenRect.top << 16 ) - ➥
sObjects[ indexCounter ].fYPos;
          }
        if( sObjects[ indexCounter ].fXPos +
            ( ( ( sObjects[ indexCounter ].fSpriteInfo )->fSpriteRect.right -
            ( sObjects[ indexCounter ].fSpriteInfo )->fSpriteRect.left ) << 16 ) >
            ( gOffscreenRect.right << 16 ) )
          {
            sObjects[ indexCounter ].fDeltaX *= -1;
            sObjects[ indexCounter ].fXPos += ( ( gOffscreenRect.right -
                ( ( sObjects[ indexCounter ].fSpriteInfo )->fSpriteRect.right -
                ( sObjects[ indexCounter ].fSpriteInfo )->fSpriteRect.left ) ) << ➥
16 ) -
                sObjects[ indexCounter ].fXPos;
          }
        if( sObjects[ indexCounter ].fYPos +
            ( ( ( sObjects[ indexCounter ].fSpriteInfo )->fSpriteRect.bottom -
            ( sObjects[ indexCounter ].fSpriteInfo )->fSpriteRect.top ) << 16 ) >
            ( gOffscreenRect.bottom << 16 ) )
          {
            sObjects[ indexCounter ].fDeltaY *= -1;
            sObjects[ indexCounter ].fYPos += ( ( gOffscreenRect.bottom -
                ( ( sObjects[ indexCounter ].fSpriteInfo )->fSpriteRect.bottom -
                ( sObjects[ indexCounter ].fSpriteInfo )->fSpriteRect.top ) ) << ➥
16 ) -
                sObjects[ indexCounter ].fYPos;
          }

        // if the object is visible
        if( sObjects[ indexCounter ].fIsVisible )
          {
            // evaluate the fixed numbers to new integer positions
            sObjects[ indexCounter ].fIntXPos = ( sObjects[ indexCounter ].fXPos ) ➥
>> 16;
            sObjects[ indexCounter ].fIntYPos = ( sObjects[ indexCounter ].fYPos ) ➥
>> 16;

            // determine the object's rect
            objectRect = ( sObjects[ indexCounter ].fSpriteInfo )->fSpriteRect;
            objectRect.left += sObjects[ indexCounter ].fIntXPos;
            objectRect.top += sObjects[ indexCounter ].fIntYPos;
            objectRect.right += sObjects[ indexCounter ].fIntXPos;
            objectRect.bottom += sObjects[ indexCounter ].fIntYPos;
```



```
            // add the new rect to the update (so it will be drawn)
            addRectToUpdate( &objectRect );
          }

        // record the this frame's visibility
        sObjects[ indexCounter ].fWasVisible = sObjects[ indexCounter ].fIsVisible;
      }
}
```

The code starts by clearing the update; we are starting a new frame here. It then scans each object. For each object, it checks to see if the object has moved. If not, we can skip it. Otherwise, the code checks to see if the object was visible last frame. If so, then the current location is added to the update so that the old position will be erased. Next, the new position of the object is calculated. If the object is visible this frame, the new position is added to the update so that the object will be drawn. Finally, this frame's visible state is stored, so next frame we can tell if the object is visible this time.

So much for the recording side of the update scheme. How about the drawing side? Here is `doDemoFrame()`:

```
//
// doDemoFrame()
//
// Move the demo ahead one frame -- move the sprites and draw.
//

void doDemoFrame( void )
{
  CGrafPtr oldPort;          // the graf port that is in place when we are called
  GDHandle oldDevice;        // the GDevice that is in place when we are called
  signed short indexCounter; // a counter to scan all the objects
  Rect updateRect;           // the rect that needs to be updated
  Point drawPoint;           // where to draw the sprite

  // if we are in step mode, return
  if ( sStepMode )
    {
      return;
    }

  // save the current port and GDevice
  GetGWorld( &oldPort, &oldDevice );

  // set the off-screen buffer as current (and lock the pixel map)
  SetGWorld( gOffscreenBuffer, ( GDHandle )kNil );
  LockPixels( gOffscreenPixels );

  // restore the background
  restoreBackground();
```

```
    // move the objects
    moveObjects();

    // determine the update rect
    if ( getUpdateRect( &updateRect ) )
      {
        // there is a rect to update

        // if the user wants it, draw the update rect
        if ( sShowUpdateAreaFlag )
          {
            FrameRect( &updateRect );
          }

        // set up to draw the sprites
        startSpriteDraw( &updateRect, gOffscreenPixels );

        // draw the sprites from back to front
        for( indexCounter = kMaxObjects - 1; indexCounter >= 0; indexCounter-- )
          {
            if ( sObjects[ indexCounter ].fIsVisible )
              {
                drawPoint.h = ( sObjects[ indexCounter ].fIntXPos ) ;
                drawPoint.v = ( sObjects[ indexCounter ].fIntYPos );
                drawSprite( sObjects[ indexCounter ].fSpriteInfo, drawPoint );
              }
          }

        // shut down the shape draw
        endSpriteDraw();

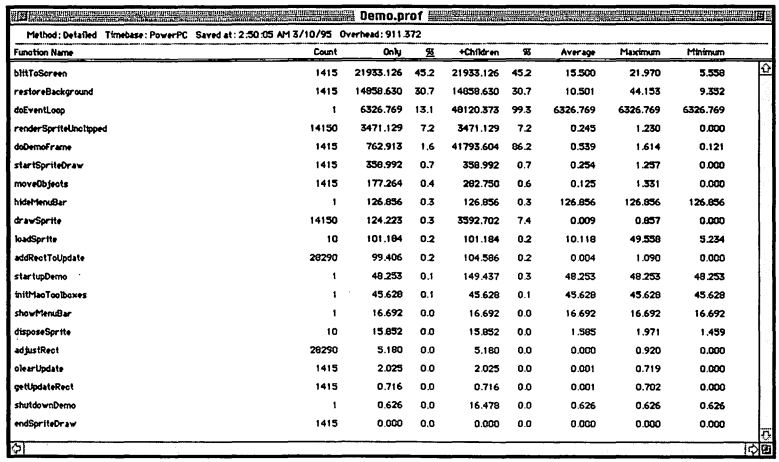
        if ( sShowUpdateAreaFlag )
          {
            // copy the buffer to the screen
            blitToScreen( &gOffscreenRect );
          }
        else
          {
            // copy the buffer to the screen
            blitToScreen( &updateRect );
          }
      }

    // restore the current port and GDevice
    UnlockPixels( gOffscreenPixels );
    SetGWorld( oldPort, oldDevice );
}
```

The key change to `doDemoFrame()` starts with the call to `getUpdate()`. If there is no update, all the draw code gets skipped. But if there is an update, the sprites are drawn clipped to the update. This ensures that objects pass in front of and behind each other without disturbing the image outside the update rectangle. Finally, the update rectangle area in the off-screen is copied to the screen.

Analysis

It's time for another profile (see figure 3.6).



Demo.prof

Method: Detailed Timebase: PowerPC Saved at: 2:50:05 AM 3/10/95 Overhead: 911.372

Function Name	Count	Only	%	+Children	%	Average	Maximum	Minimum
blitToScreen	1415	21933.126	45.2	21933.126	45.2	15.500	21.970	5.358
restoreBackground	1415	14858.630	30.7	14858.630	30.7	10.501	44.153	9.332
doEventLoop	1	6326.769	13.1	48120.373	99.3	6326.769	6326.769	6326.769
renderSpriteUnclipped	14150	3471.129	7.2	3471.129	7.2	0.245	1.230	0.000
doDemoFrame	1415	762.913	1.6	41793.604	86.2	0.539	1.614	0.121
startSpriteDraw	1415	358.992	0.7	358.992	0.7	0.254	1.257	0.000
moveObjects	1415	177.264	0.4	282.750	0.6	0.125	1.331	0.000
hideMenuBar	1	126.856	0.3	126.856	0.3	126.856	126.856	126.856
drawSprite	14150	124.223	0.3	3392.702	7.4	0.009	0.857	0.000
loadSprite	10	101.184	0.2	101.184	0.2	10.118	49.538	5.234
addRectToUpdate	28290	99.406	0.2	104.586	0.2	0.004	1.090	0.000
startupDemo	1	48.253	0.1	149.437	0.3	48.253	48.253	48.253
initMacToolboxes	1	45.628	0.1	45.628	0.1	45.628	45.628	45.628
showMenuBar	1	16.692	0.0	16.692	0.0	16.692	16.692	16.692
disposeSprite	10	15.852	0.0	15.852	0.0	1.585	1.971	1.459
adjustRect	28290	5.180	0.0	5.180	0.0	0.000	0.920	0.000
clearUpdate	1415	2.025	0.0	2.025	0.0	0.001	0.719	0.000
getUpdateRect	1415	0.716	0.0	0.716	0.0	0.001	0.702	0.000
shutdownDemo	1	0.626	0.0	16.478	0.0	0.626	0.626	0.626
endSpriteDraw	1415	0.000	0.0	0.000	0.0	0.000	0.000	0.000

Figure 3.6 *Profiler results.*

The profiler now shows `blitToScreen()` taking 45.2 percent. Not much of an improvement. Why? Take a good look at the demo with the show update mode on. Even though we are not copying the entire screen, we are usually copying most of it. We need to find a way to reduce that area even more.

Example 6: An Update Rectangle List

Rather than keep only one rectangle, let's keep a list of them. Every time we add a rect to the update, it will be added to a list. When we get around to drawing, we can update just those rectangles. This should give us better performance, because we will be moving less memory.

There are, of course, a few problems. First, what about overlapping rectangles? We don't want to draw the same area twice; that would defeat the whole purpose. Second, there is a setup cost to `CopyBits()`. Each time you use it, it takes some time, regardless of how much or little data you are using to prepare the copy. So if you were updating a thousand tiny rectangles, the setup time for `CopyBits()` would slow you down substantially. Finally, if you have a maximum number of rectangles, how do you handle list overflows?

In this demo, we implement a rectangle list as a static array. When a new rectangle is added, this is how it's handled:

1. Adjust the new rectangle to 32-bit bounds.
2. Check to see if the new rectangle touches any of the rectangles in the list.
3. If it does, join the rectangles. Remove the joined rectangle from the list and call the add rectangle routine with the joined rectangle.
4. If it doesn't touch and there is room for another rectangle, add it to the list.
5. If it doesn't touch and there isn't room for another rectangle, join the new rectangle to whichever rectangle in the list will have the least increase in area after the joining.

This method keeps a list of fixed size without overflowing and it also handles overlap.

Here is `addRectToUpdate()` with this method:

```
//
// addRectToUpdate()
//
// Adds a rect to the update area.
//

void addRectToUpdate( Rect *inUpdateRect )
{
   Rect newRect;                  // the rect we are adding
   unsigned long indexCounter;    // a counter to scan the rect array
   unsigned char xTouch;          // do the rects intersect horizontally
   unsigned char yTouch;          // do the rects intersect vertically
   unsigned long bestIndex;       // the index of our best candidate rect
   unsigned long bestArea;        // the area of our best candidate rect
   unsigned long scratchArea;     // a temp area for calculations
   Rect scratchRect;              // a temp rect for calculations
   Rect bestRect;                 // the best rect so far

   // adjust the rect to 32-bit bounds
   newRect = *inUpdateRect;
   adjustRect( &newRect );

   // scan the list and see if the rect touches any of the rects there
   for( indexCounter = 0; indexCounter < sUpdateRectCount; indexCounter++ )
      {
         // do the rects touch?

         // check horizontally
         if ( newRect.left < sUpdateRects[ indexCounter ].left )
            {
               if ( newRect.right >= sUpdateRects[ indexCounter ].left )
                  {
                     xTouch = kTrue;
                  }
```

```
            else
               {
                  xTouch = kFalse;
               }
         }
      else
         {
            if ( sUpdateRects[ indexCounter ].right >= newRect.left )
               {
                  xTouch = kTrue;
               }
            else
               {
                  xTouch = kFalse;
               }
         }

      // check vertically
      if ( newRect.top < sUpdateRects[ indexCounter ].top )
         {
            if ( newRect.bottom >= sUpdateRects[ indexCounter ].top )
               {
                  yTouch = kTrue;
               }
            else
               {
                  yTouch = kFalse;
               }
         }
      else
         {
            if ( sUpdateRects[ indexCounter ].bottom >= newRect.top )
               {
                  yTouch = kTrue;
               }
            else
               {
                  yTouch = kFalse;
               }
         }

      // if the rects do touch,
      if ( yTouch && xTouch )
         {
            // join the rects into new rect
            newRect.left = ( newRect.left < sUpdateRects[ indexCounter ].left ) ? ➥
newRect.left : sUpdateRects[ indexCounter ].left;
            newRect.top = ( newRect.top < sUpdateRects[ indexCounter ].top ) ? ➥
newRect.top : sUpdateRects[ indexCounter ].top;
```

```
            newRect.right = ( newRect.right > sUpdateRects[ indexCounter ].right ) ➥
? newRect.right : sUpdateRects[ indexCounter ].right;
            newRect.bottom = ( newRect.bottom > sUpdateRects[ indexCounter ] ➥
.bottom ) ? newRect.bottom : sUpdateRects[ indexCounter ].bottom;

            // remove the rect from the list
            sUpdateRectCount -= 1;
            for( ; indexCounter < sUpdateRectCount; indexCounter++ )
               {
                  // shift the next rects down
                  sUpdateRects[ indexCounter ] = sUpdateRects[ indexCounter + 1 ];
               }

            // call this routine with the new, joined rect
            addRectToUpdate( &newRect );

            // we are done
            return;
         }
      }

   // if there is room, add this rect to the list
   if ( sUpdateRectCount < sNumUpdateRects )
      {
         sUpdateRects[ sUpdateRectCount ] = newRect;
         sUpdateRectCount++;
      }
   else
      {
         // otherwise find the rect in which the union is the smallest
         bestIndex = 0;
         bestArea = 0xFFFFFFFF;        // a very big area
         for( indexCounter = 0; indexCounter < sUpdateRectCount; indexCounter++ )
            {
               // create the unified rect
               scratchRect.left = ( newRect.left < sUpdateRects[ indexCounter ].left ➥
) ? newRect.left : sUpdateRects[ indexCounter ].left;
               scratchRect.top = ( newRect.top < sUpdateRects[ indexCounter ].top ) ? ➥
newRect.top : sUpdateRects[ indexCounter ].top;
               scratchRect.right = ( newRect.right > sUpdateRects[ indexCounter ] ➥
.right ) ? newRect.right : sUpdateRects[ indexCounter ].right;
               scratchRect.bottom = ( newRect.bottom > sUpdateRects[ indexCounter ] ➥
.bottom ) ? newRect.bottom : sUpdateRects[ indexCounter ].bottom;

               // determine how much this will grow the rect
               scratchArea = ( scratchRect.right - scratchRect.left ) * ( ➥
scratchRect.bottom - scratchRect.top ) -
                     ( sUpdateRects[ indexCounter ].right - sUpdateRects[ indexCounter ➥
] .left ) *
```

```
                ( sUpdateRects[ indexCounter ].bottom - sUpdateRects ➥
[ indexCounter ].top );

            // if this area is smaller than the best so far, keep it
            if ( scratchArea < bestArea )
               {
                  bestArea = scratchArea;
                  bestIndex = indexCounter;
                  bestRect = scratchRect;
               }
         }

      // save our best candidate
      sUpdateRects[ bestIndex ] = bestRect;
   }
}
```

The code easily falls into the three sections of the method: First, check for touching rectangles; second, add a new rectangle if there is no touching rectangle and if there is room; and third, if there is no touching rectangle and no room for a new rectangle, join to the best available rectangle. There are a couple key points, however. When we check for overlap, we're looking for rectangles that touch as well as intersect. That's why the code does not use `SectRect()`; it doesn't detect touches, only intersections. Also, this routine is recursive. When two rectangles intersect, the routine is called with the new, joined rectangle. This should not be a problem, because the recursion depth is limited to the number of rectangles in the list.

So, how do you use the rectangle list? Actually, we use it just like we used the single update rect, except we repeat the process for each rectangle. Here is `doDemoFrame()`:

```
//
// doDemoFrame()
//
// Move the demo ahead one frame -- move the sprites and draw.
//

void doDemoFrame( void )
{
   CGrafPtr oldPort;           // the graf port that is in place when we are called
   GDHandle oldDevice;         // the GDevice that is in place when we are called
   signed long indexCounter;   // a counter to scan all the objects
   signed long rectCounter;    // a counter to scan all the rects
   Rect updateRect;            // the rect that needs to be updated
   Point drawPoint;            // where to draw the sprite

   // if we are in step mode, return
   if ( sStepMode )
      {
         return;
      }
```

```
// save the current port and GDevice
GetGWorld( &oldPort, &oldDevice );

// set the off-screen buffer as current (and lock the pixel map)
SetGWorld( gOffscreenBuffer, ( GDHandle )kNil );
LockPixels( gOffscreenPixels );

// restore the background
restoreBackground();

// move the objects
moveObjects();

// draw the sprites in each of the update rects
for( rectCounter = 0; rectCounter < getUpdateRectCount(); rectCounter++ )
  {
    // get one of the update rects
    getUpdateRect( rectCounter, &updateRect );

    // if the user wants it, draw the update rect
    if ( sShowUpdateAreaFlag )
      {
        FrameRect( &updateRect );
      }

    // set up to draw the sprites
    startSpriteDraw( &updateRect, gOffscreenPixels );

    // draw the sprites from back to front
    for( indexCounter = kMaxObjects - 1; indexCounter >= 0; indexCounter-- )
      {
        if ( sObjects[ indexCounter ].fIsVisible )
          {
            drawPoint.h = ( sObjects[ indexCounter ].fIntXPos ) ;
            drawPoint.v = ( sObjects[ indexCounter ].fIntYPos );
            drawSprite( sObjects[ indexCounter ].fSpriteInfo, drawPoint );
          }
      }

    // shut down the shape draw
    endSpriteDraw();
  }

if ( sShowUpdateAreaFlag )
  {
    // copy the buffer to the screen
    blitToScreen( &gOffscreenRect );
  }
else
  {
```

```
        // copy each of the update rects to the screen
        for( rectCounter = 0; rectCounter < getUpdateRectCount(); rectCounter++ )
          {
            // get one of the update rects
            getUpdateRect( rectCounter, &updateRect );

            // copy the buffer to the screen
            blitToScreen( &updateRect );
          }
      }

    // restore the current port and GDevice
    UnlockPixels( gOffscreenPixels );
    SetGWorld( oldPort, oldDevice );
}
```

Here you can see the update process. After moving all the objects, the code draws, clipped to each rectangle. Remember that the clip case of "don't draw at all" is very fast, so repeatedly drawing clipped to small rectangles should not be a problem. Later on, the screen is updated by copying the data in each of the rectangles.

Analysis

Run the demo and activate the show update mode. See how little area is bounded by the rectangles? There are two new controls for this demo (see table 3.6).

Table 3.6
New Demo Commands

Action	Meaning
+	Increase the maximum rectangle list size (up to ten).
-	Decrease the maximum rectangle list size (down to one).

Try raising and lowering the number of rectangles. The lesson here is the more rectangles, the better. Remember, however, that there is a point at which the number of rectangles can cause problems due to `CopyBits()`' setup time.

Let's see how much this improved things (see figure 3.7).

Demo.prof

Method: Detailed Timebase: PowerPC Saved at: 3:57:23 AM 3/10/95 Overhead: 2651.494

Function Name	Count	Only	%	+Children	%	Average	Maximum	Minimum
restoreBackground	1571	17243.758	41.8	17243.758	41.8	10.976	44.677	9.960
blitToScreen	12032	9736.813	23.6	9736.813	23.6	0.809	5.967	0.168
doEventLoop	1	6971.384	16.9	40932.049	99.2	6971.384	6971.384	6971.384
renderSpriteUnclipped	15710	3792.903	9.2	3792.903	9.2	0.241	1.136	0.000
doDemoFrame	1571	1656.256	4.0	33960.665	82.3	1.054	2.151	0.560
startSpriteDraw	12032	772.729	1.9	772.729	1.9	0.064	1.306	0.000
addRectToUpdate	50788	383.299	0.9	396.877	1.0	0.008	1.178	0.000
moveObjects	1571	185.336	0.4	577.934	1.4	0.118	1.246	0.000
drawSprite	120320	158.792	0.4	3948.901	9.6	0.001	1.004	0.000
hideMenuBar	1	123.761	0.3	123.761	0.3	123.761	123.761	123.761
initMacToolboxes	1	73.246	0.2	73.246	0.2	73.246	73.246	73.246
loadSprite	10	54.389	0.1	54.389	0.1	5.439	6.757	3.674
startupDemo	1	49.883	0.1	104.272	0.3	49.883	49.883	49.883
showMenuBar	1	16.016	0.0	16.016	0.0	16.016	16.016	16.016
disposeSprite	10	15.972	0.0	15.972	0.0	1.597	1.879	1.446
adjustRect	50788	13.647	0.0	13.647	0.0	0.000	0.904	0.000
getUpdateRect	24064	10.586	0.0	10.586	0.0	0.000	0.655	0.000
getUpdateRectCount	27206	3.718	0.0	3.718	0.0	0.000	0.475	0.000
endSpriteDraw	12032	2.347	0.0	2.347	0.0	0.000	0.775	0.000
shutdownDemo	1	0.639	0.0	16.611	0.0	0.639	0.639	0.639
clearUpdate	1571	0.482	0.0	0.482	0.0	0.000	0.482	0.000

Figure 3.7 *Profiler results.*

Good news! `blitToScreen()` dropped from 45 percent to 24 percent! This really makes a difference in how fast things can go.

Example 7: The Background

That last demo is really nice, except there is no background. The call to `restoreBackground()` just calls `EraseRect()` and clears the off-screen to white. How can we add a background?

This simplest way to handle a background is to have another off-screen buffer with the background in it. Whenever you need to erase an object, you simply copy from the background into the main off-screen. That is the approach we're going to take here.

We have added a new picture, `#128`, to Demo.res. It contains the background we'll use. In addition, we have moved `restoreBackground()` into its own module: `background`. We have also given it a parameter: the rectangle to restore.

There are three functions to background: `startupBackground()`, `shutdownBackground()`, and `restoreBackground()`. Here is `startupBackground()`:

```
//
// startupBackground()
//
// Allocate and load any data needed for the background.
//
```

```
void startupBackground( void )
{
   PicHandle backgroundPict;          // the background in PICT form
   CGrafPtr oldPort;                  // the current port
   GDHandle oldGDevice;               // the current GDevice

   // load the PICT
   backgroundPict = GetPicture( kBackgroundPictResID );

   // create the GWorld
   NewGWorld( &sBackgroundWorld, 8, &( ( *backgroundPict )->picFrame ),
        gAppColorTable, ( GDHandle )kNil, keepLocal );

   // get the pixmap
   sBackgroundPixels = GetGWorldPixMap( sBackgroundWorld );

   // lock the pixels
   LockPixels( sBackgroundPixels );

   // save the old port and device
   GetGWorld( &oldPort, &oldGDevice );

   // set the new GWorld
   SetGWorld( sBackgroundWorld, ( GDHandle )kNil );

   // draw the picture
   DrawPicture( backgroundPict, &( ( *backgroundPict )->picFrame ) );

   // dump the picture
   ReleaseResource( ( Handle )backgroundPict );

   // restore the old port and device
   SetGWorld( oldPort, oldGDevice );

   // lock the pixels
   UnlockPixels( sBackgroundPixels );
}
```

First, `startupBackground()` creates an off-screen. Next, it loads and draws the background picture into the off-screen. Just before exiting, `startupBackground()` disposes of the PICT data—it is no longer needed.

`shutdownBackground()` is very simple:

```
//
// shutdownBackground()
//
// Free any memory used by the background.
//
```

```
void shutdownBackground( void )
{
   // dump the GWorld
   DisposeGWorld( sBackgroundWorld );
}
```

All `shutdownBackground()` does is dispose of the off-screen that `startupBackground()` allocated. Both `startupBackground()` and `shutdownBackground()` are called by their demo counterparts (`startupDemo()` and `shutdownDemo()`).

The real work occurs in `restoreBackground()`:

```
//
// restoreBackground()
//
// Draw the background in the given rect.
//

void restoreBackground( Rect *inRect )
{
   // lock the pixels
   LockPixels( sBackgroundPixels );

   // make sure that the fore and back colors are correct to prevent colorize mode
   ForeColor( blackColor );
   BackColor( whiteColor );

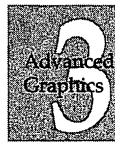
   // Copy the screen's color table seed into the source pixmap.
   // This will minimize CopyBits()' setup time.
   ( *( ( *sBackgroundPixels )->pmTable ) )->ctSeed = ( *( ( *gOffscreenPixels ) ➥
->pmTable ) )->ctSeed;

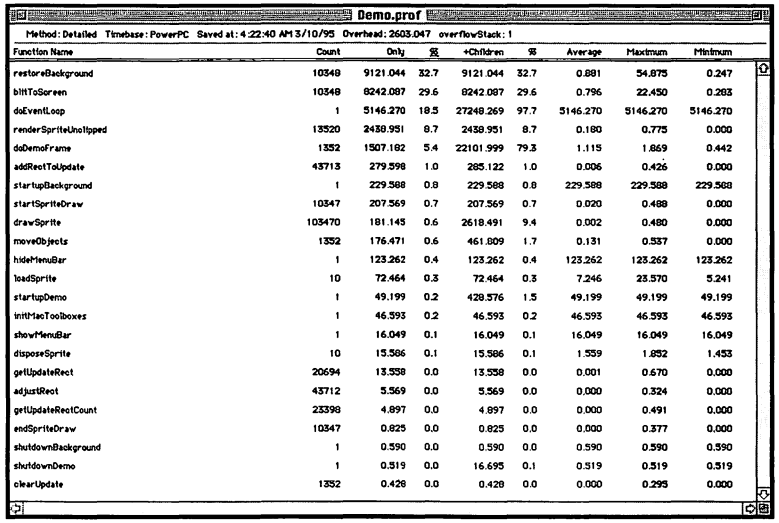
   // copy the buffer to the screen
   CopyBits( ( BitMap * )( *sBackgroundPixels ), ( BitMap * )( *gOffscreenPixels ),
        inRect, inRect, srcCopy, ( RgnHandle )kNil );

   // lock the pixels
   UnlockPixels( sBackgroundPixels );
}
```

`restoreBackground()` is actually a modified copy of `blitToScreen()`. It differs only in its source and destination buffers (background to main off-screen verses main off-screen to screen).

There is also one small but important change to `doDemoFrame()`: The background is restored only in the update area. This makes a real difference in profiler: `restoreBackground()` drops from 42 percent to 33 percent (see figure 3.8).





Demo.prof

Method: Detailed Timebase: PowerPC Saved at: 4:22:40 AM 3/10/95 Overhead: 2603.047 overflowStack: 1

Function Name	Count	Only	%	+Children	%	Average	Maximum	Minimum
restoreBackground	10348	9121.044	32.7	9121.044	32.7	0.881	54.875	0.247
bltToScreen	10348	8242.087	29.6	8242.087	29.6	0.796	22.450	0.283
doEventLoop	1	5146.270	18.5	27248.269	97.7	5146.270	5146.270	5146.270
renderSpriteUnclipped	13520	2438.951	8.7	2438.951	8.7	0.180	0.775	0.000
doDemoFrame	1352	1507.182	5.4	22101.999	79.3	1.115	1.869	0.442
addRectToUpdate	43713	279.598	1.0	285.122	1.0	0.006	0.426	0.000
startupBackground	1	229.588	0.8	229.588	0.8	229.588	229.588	229.588
startSpriteDraw	10347	207.569	0.7	207.569	0.7	0.020	0.488	0.000
drawSprite	103470	181.145	0.6	2618.491	9.4	0.002	0.480	0.000
moveObjects	1352	176.471	0.6	461.809	1.7	0.131	0.537	0.000
hideMenuBar	1	123.262	0.4	123.262	0.4	123.262	123.262	123.262
loadSprite	10	72.464	0.3	72.464	0.3	7.246	23.570	5.241
startupDemo	1	49.199	0.2	428.576	1.5	49.199	49.199	49.199
initMacToolboxes	1	46.593	0.2	46.593	0.2	46.593	46.593	46.593
showMenuBar	1	16.049	0.1	16.049	0.1	16.049	16.049	16.049
disposeSprite	10	15.586	0.1	15.586	0.1	1.559	1.852	1.453
getUpdateRect	20694	13.558	0.0	13.558	0.0	0.001	0.670	0.000
adjustRect	43712	5.569	0.0	5.569	0.0	0.000	0.324	0.000
getUpdateRectCount	23398	4.897	0.0	4.897	0.0	0.000	0.491	0.000
endSpriteDraw	10347	0.825	0.0	0.825	0.0	0.000	0.377	0.000
shutdownBackground	1	0.590	0.0	0.590	0.0	0.590	0.590	0.590
shutdownDemo	1	0.519	0.0	16.695	0.1	0.519	0.519	0.519
clearUpdate	1352	0.428	0.0	0.428	0.0	0.000	0.295	0.000

Figure 3.8 *Profiler results.*

It's been quite a journey. We started with a demo that was unbearably slow and we ended with a quick animation. Hopefully, the methods and tools presented here will help you venture further. On that thought, let me suggest some directions you could pursue.

The encoded sprite draw has a lot of room for enhancement. You could extend the operations to include all sorts of effects: darken, lighten, inverse, color filters, and so on. In addition, you could add features to the drawing code such as flipping on x and y and scaling.

The encoding tool could use a lot of work as well. A version with an interface would make working with it much easier. You could also create one that would use QuickTime movies as source material.

There may be even faster ways to draw spites. One possibility is to compile the sprites into C or assembly code. In fact, I created a sprite compiler to use as an example in this chapter. However, it was slower than the encoded sprite drawer because the code it generates is not very efficient and it is so large that it causes problems for the cache and prefetch queue. For that reason—and to save space—I chose not to discuss it. However, the code is on the CD-ROM in the 03 - Advanced Graphics folder as Shape Compiler Code, if you want to take a look at it.

The update rectangle list code is pretty complete, but it may be possible to refine its methods so that it is smarter.

Finally, we only touched on backgrounds here. There are hundreds of things you could do with the `background` module. Some options include tiled backgrounds, scrolling backgrounds, and who knows what else.

You will develop your own set of routines and tools that fit your product's needs. These should provide something of a start. Have fun!

Profile: Jason Jones

Jason Jones is the lead programmer at Bungie Software for the very hot game Marathon. In case you are among the few who haven't seen it, Marathon is a first-person, fully 3-D action/adventure/combat game in the tradition of Pathways Into Darkness, Wolfenstein, Doom, and so on.

Q: Could you give me a general idea of Marathon's architecture? Is there a rendering engine and a refresh daemon a lá Doom? How exactly is the program structured?

A: No, Marathon doesn't have that type of asychronous operation between the game model and the rendering engine. They're clearly separate in the architecture, but all of the game code is executed between frames while we're working on rendering the image. We generate a frame, display it, and we figure what happened in the world while we were generating that frame, update our state of the world, and then draw the next frame.

Q: Basically, the program generates a frame or a screen of animation every so often. In between this, it processes events that occurred, maybe a user interaction.

A: Let me explain that a little better. There are two things that go on. There's a big piece of data sitting in memory somewhere that represents the state of the world at any given time. It tells you where the monsters are, where the projectiles are in the air, and where all the players are. It also indicates if doors are open, if objects are in certain places, and if there's something exploding. The data block has all that information.

Basically, we give that big piece of data to the rendering engine, which turns it into a picture. It [the engine] renders a screen to show to the player, based on where the player is and what he can see from that position. Essentially, the input to the rendering engine is the state of the world, and the output is a bitmap that represents what the player sees. So then we display that bitmap and then, depending on how much time has passed, we may have a lot or a little work to do. If it took a long time to render that frame, we have a lot of

continued

work to do. If that frame was rendered in less than a 30th of a second, then we might not have any work to do. In that case, we actually wait until the state of the world changes.

We have thirty updates a second, or, put another way, our world changes every 30th of a second. We monitor the player's keyboard every 30th of a second, and we check the network for other players' commands every 30th of a second. Then, between frames, we handle all the updates that occurred while we were rendering the previous frame. Finally, we render the next frame. It is not like Doom, where things reportedly happen asychronously. However, we do a lot of stuff asynchronously. A lot of work on the update happens this way. We have an interrupt that fires every 30th of second and a lot of work happens during that interrupt.

Q: What's the interrupt? Is that the frame update or what?

A: No, the frame update is every 1/30th of a second. The state of the world is updated from the interrupt.

Q: So the interrupt handles updating the parameters in the big data block that represents the world?

A: It checks the player's keyboard for keystrokes and checks the network. Stuff like that.

Q: Do you have a really complex event loop that primarily handles updates and everything else is sort of a subsidiary to that?

A: Yes. The complex loop has to update everything. It lets the monsters move. It moves all the bullets, and it updates all the explosions. It lets all the players move according to the commands they've given, if any. By complex I mean it does a whole lot of work. Everything that has to do anything in Marathon has to do it every 30th of a second. So we do have a big loop that basically goes out and handles all of that.

Q: Is this your main event loop? Or is there no similarity to a Mac application event loop at all?

A: There is very little similarity. We actually do have two loops: a game loop and a Mac event loop. The game loop is a little bit subservient to the Mac event loop. We have a very simple Mac event loop. It basically looks for Command-Q to quit the game, and it really doesn't do much more than that. The main Mac event loop then calls the game loop, which does the updates. It goes through as many iterations as it has to, depending on how long the last frame took to render. The main purpose of our Mac event loop is to avoid giving the operating system any time.

Q: That way, Marathon can have most of the processor's time to get its work done. So you just look for look Command-Q for quit?

A: Honestly, the only things I think we look for from the operating system are Command-Q and Command-W, which both do the same thing. That's probably it.

Q: So most of the user interface processing is handled in the game loop as well?

A: Exactly. It's all game-related stuff like updating the weapons—things like that.

Q: On the user event stuff, like the keyboard, do you use Toolbox calls or do you do some other mechanism to grab a keyboard event?

A: We use the Toolbox call `GetKeys()`. We do not read the state of the keyboard through the Event Manager.

Q: You use `GetKeys()` and then look at the low-memory global `keymap` array and pull the keystroke out of there. Now, can you tell us about the rendering engine?

A: The most difficult part of the rendering engine to write is, given a point in the world and the direction in which the user is looking, how do you find out what surfaces must be drawn and in what order they have to be drawn?

Q: In other words, you're trying to figure out if there are objects in the way. You don't worry about drawing the rest of the wall, because there's something in front of it you have to worry about drawing first.

A: The problem is dealt with this way: Given an entire map of the area, the first thing you do is trivially discard all the stuff that is behind you and to the sides of you. Then, for stuff that you can see, you have to figure out what order you have to draw each object. You need to know where it is, what orientation it's in for proper texture mapping, and then whether there are any objects in front of it, such as monsters or missiles. So, a very large part of the renderer is devoted to simply generating a list of 3-D polygons with enough information for texture mapping.

Q: I've seen in the Preferences section of Marathon where you can turn the texture mapping of the floors and ceilings off. Is that to enhance playability on a slower system?

A: You can turn off the ceilings and floors. Right. We have two totally different subroutines: One of them handles floor and ceilings, and the other handles the walls. The technique and texture maps are totally different.

Q: Why do you have the two routines anyway? Was there some compelling design reason you did it that way?

A: Absolutely. There are certain mathematical simplifications you can exploit for speed. They're totally different in the case of mapping a floor and the case of mapping a wall. So, we wrote two. Instead of writing one slow general-purpose routine, we wrote two specialized ones. And, the simplifications are fairly obvious. A wall always moves straight up and down on your screen and is

continued

always perpendicular to your viewing direction. The floors are always parallel to your viewing direction. When we're doing the texture mapping, we're actually stepping along the floors from left to right on the screen, and we're stepping along the walls from top to bottom.

Q: Because the walls are perpendicular, all you have to do for the texture map is crunch the pattern down. But because of perspective, with the floors and ceiling textures there's a lot more math computations involved. Is that the idea?

A: Right. The floors are actually more complicated. When we're texture mapping a wall, we can figure out what column in the texture we have the texture map, and then we know that as long as we're within the same column on the screen, we're coming from the same column of the texture. So all we have to do is worry about that single column. So that's why we do walls in column order.

Q: Quick technology term here: What do you mean by "column"? Is that an architectural element, or is it something else?

A: I just mean a column of pixels on the screen. A vertical column of pixels as opposed to a row of pixels. A row is horizontal, like a scan line, left to right on the screen. Columns are top to bottom and rows are left to right.

Q: Now lighting—do you handle that any special way?

A: This is sort of a general thing. All the lighting in Marathon is done through look-up tables. We don't do any calculations on the fly. We're not sitting there with a pixel, saying, "All right, we need to darken this pixel 15 percent," and do a bunch of multiplications and divisions. That's true for pretty much everything in Marathon. Anything interesting that happens probably goes through a look-up table.

So we have a lighting intensity value and we have a pixel color value, and we combine those in the right ways, send them through a table, and we get out a new pixel value. Everything that goes to the screen gets lighted in some way. Even if a pixel's value is 100 percent intensity, it still passes through a lighting table and remains unchanged. On the lowest level, the lighting is done through look-up tables. On a high level, we can assign intensities to individual sprites walking around the world, and we can assign intensities to the floors and ceilings of the map. The walls then intelligently pick up the intensities of the ceilings and floors above and below them.

Q: So, you don't do any special region mapping or anything. Say there's an open door and there's a beam of light coming out of it, lighting up the corridor.

A: Right. And we can do that with some tricks. If you've seen any of the map editors for Marathon, we can create a small segment of floor that looks like a beam of light coming out of a door. We light that segment very bright and light the area around it very dark, so it looks like there's light coming out of the door.

It's a very common trick with engines like the one that we used. You are able to create some pretty neat lighting effects by changing the geometry of your floor. You plan the geometry of your floor ahead of time so it looks like light casting out of a doorway.

Q: For the rendering engine, how many people did you have working on it, and how long did it take?

A: There were three people working on Marathon, and it took us a little less than a year. We started in May and were finished in December 1994.

Q: Was there one person handling the engine, or did everybody work on it?

A: Two people worked on the texture mapping and the rendering parts of the game. The third programmer primarily handled writing the Mac user interface and some of the less important parts of the actual game itself.

Q: What sort of things did you do to improve game performance? You mention the look-up tables. Were there other techniques, such as using `CopyBits()` for blitting to the screen, or color palette management?

A: Not really. We do use `CopyBits()`—that might be an interesting piece of trivia. We have a lot of 68K assembly code. At the moment, the game doesn't have any PowerPC assembly code. We're relying on the compiler to make our code look good.

Q: What development tools did you use?

A: We used MPW exclusively.

Q: What was the reason for using MPW? Better code optimizations, or just that you were familiar with the environment?

A: Better code optimization. It's also what we happen to be familiar with. We know MPW and have a lot of time invested into MPW scripts and tools. MPW is much more expandable, and it's a more versatile system.

Q: To get back to coding issues, you mention lots of 68K assembly language. Do you have a ballpark idea of what percentage of Marathon is in 68K assembly?

A: Not that much of the whole program itself—I would say five percent at the very bottom. It's located in the texture-mapping routines. Maybe even less than that; it's probably more like two percent, but that is a lot of assembly because Marathon is a very big project. The assembly code probably took me a good two weeks, off and on, to finish. Which is a really high investment for a pretty small part of the project, but you have to do it to make the program run faster.

Q: It sounds like Marathon was written in C.

A: Right. The entire thing is written in C.

continued

Q: Did you just do some code profiling and figured out what 10 percent of the program was called 90 percent of the time, and then fine-tune those routines in assembler?

A: The 10/90 thing is a cliché, but that's right. That's exactly what we did. We took the game after it was done, we profiled it, and we figured out what was slow and rewrote that in assembly. Profiling was actually helpful for other reasons, too. A lot of things can be fixed in C as well. If your algorithm is bad, then the assembly code can be bad. Assembly was a last resort for us.

Q: So you optimized the C code first. You looked for better algorithms in C before resorting to assembly.

A: We pretty much expected to go to assembly code in the texture mapping, but we absolutely refused to go to assembly in any of the high-level game code. If the monsters move around really slowly, it's because their code is slow. If the map monster code is slow, we weren't going to rewrite it in assembly—we just fixed it so that it worked better and faster. Because that's stuff you just don't want to do.

Q: I'm guessing that you're considering porting Marathon to another platform at some point.

A: Obviously for portability, assembly language code is a huge liability.

Q: You mention texture mapping. What other parts of Marathon needed that final performance boost from assembly language?

A: Actually, just the texturing mapping. That's one reason why the texture mapping was kept separate from the rendering. That front end that builds the surfaces loop.

Q: Was there any difficulty in presenting things in 3-D and keeping track of it? I've wandered around in corridors that go one above another.

A: Not really. We planned for that kind of stuff and it works pretty well. There's some tricks, but that was a fairly easy part of the project compared to say, the front end of the rendering engine. The collision detection code is really complicated as well.

Q: Why is that complicated?

A: Just because it's a lot of gross maps. You're dealing with an object, an arbitrary polygon, that's moving around, and you have to make sure it doesn't go through a wall. In fact, you have to stop the object before it goes through a wall, not right up against the wall. You end up with a pretty weird, complicated set of equations.

Q: Did you use signed integer math and keep away from doing floating point calculations?

A: Yes. All the complicated math in Marathon is done in fixed point math for speed. There's no floating point.

Q: How do you manage the game map, or keep things mapped in memory, such as the tunnels? Is there a list, or what?

A: Yes. It's just a bunch of lists. There's a monster list, a projectile list, a list of the walls, doors, stuff like that. It's a fairly straightforward arrangement. Anybody who's seen the map editor can pretty much appreciate what all our low-level data structures look like. There's a set of points, a set of lines that connect points, and a set of polygons, which are made of lines. They're just a bunch of lists in memory.

Q: Now, about Marathon's network capabilities. Suppose several people are playing a networked death match. How do you handle network protocols and communications? How does that big global block of data get updated so that it looks the same on everybody's screen? So that what happens to one person gets relayed (or transmitted) to everyone else?

A: One of the tricks we used in Marathon was to try to make its network operation no different from its local operation. That is, commands from remote machines are really no different than the commands from your local machine. Inside the game engine itself, we don't special-case the networking commands. Marathon is set up so it's almost like there are eight people on the same keyboard as far your machine is concerned. The network protocol itself is packet-based. We don't use any of the fancy, higher-level services of AppleTalk. We don't use any of the data streams stuff. It's all DDP.

Q: Datagram Delivery Protocol.

A: Exactly. We use DDP with some tricks to try to minimize the number of packets you send around the network, especially over LocalTalk, so that we don't overload the network. On an Ethernet network, we're sending packets 30 times a second to make sure that everybody keeps up with the game. The networking code is another thing like the rendering. It's just heinously complicated.

Q: So the game engine sees all the commands and it doesn't matter where they come from, since they're all the same format. However, suppose one guy has the flame thrower and he's flaming some hapless person. For the person on the receiving end of that, you know his state's going to be entirely different from the person wielding the flame thrower.

A: Each person's computer knows who is playing at the keyboard. So, that's the view that they see. If you check the right box in a network game set dialog, you can actually switch between different players' views by hitting the Delete key. That indicates that your computer always knows what is happening to everybody else. So, even if you can't see the stuff, your computer is keeping track of it.

continued

Really the only thing your computer is doing differently from everyone else's computer is the point of view. Rendering a different view of the world.

Q: So, in the game engine, part of that global information is events coming in. They could be from anywhere, say another machine. For one person's state, perhaps he's being hit by a rocket. And for another person, he's the guy that shot the rocket.

A: Each machine has to know who the guy at the keyboard is playing. So the game knows what to show. There's a checkbox in the setup game dialog which allows you to see other players on the map. If that box is checked, you can see other players on your map. You're also able to switch to their point of view by hitting the Delete key. That doesn't work in the regular game—that would be cheating. You can see the weapons they're carrying, how much health they have, everything about them. Even their motion sensor.

Q: For the point of view, when you finally get an image rendered, do you use `CopyBits()` to get it to the screen?

A: Exactly. `CopyBits()` is extremely fast.

Q: Did anything specific stand out when you did performance tuning or the design of the game in general?

A: I think that the most important thing we did was remove the Mac operating system from the game as much as possible. We call the MacOS to draw graphics, play sound, or get something from the keyboard. So we're not doing anything with windows. We're not calling our event loop—`WaitNextEvent()` rather—really often. We call it very infrequently. We're not handling mouse clicks through the operating system, we're not handling keyboard events through the operating system. Most of our graphics are bitmaps, so we're not doing a lot of line-drawing calls, rectangle-drawing calls, or other QuickDraw stuff. So really we don't use many OS calls, just Time Manager calls, `CopyBits()` and `GetKeys()`.

Q: As you built the game, were there instances for which you realized the design was flawed? Flaws where you said, "We need to redesign this for better play," or "It's easier for somebody to navigate around if this code was changed"?

A: Marathon suffered that calamity many times. We did that with the maps. We changed the maps to make the game play different. And we did that with the game code as well. Especially the rendering engine. I forget the exact number, but it was fully rewritten at least three times. It also suffered a lot of small changes here and there. The texture-mapping routines underwent the same thing. They were fully rewritten at least twice. Other parts of the game had the same thing happen. You wrote something and it doesn't work, then you fix it. A lot of times when you do something for the first time, it doesn't work. That happened a lot.

Q: Were there performance issues, where something works, but it's so slow you had to use a different algorithm or change the game architecture slightly or a lot, perhaps to improve performance?

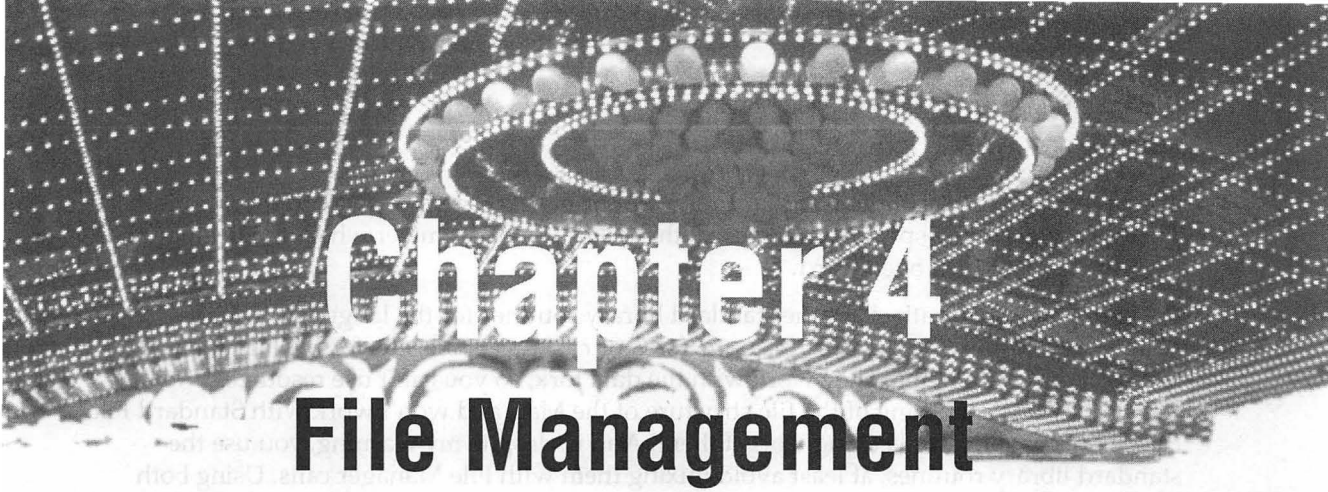
A: Yes, that happened. It's kind of interesting because most of the time, for instance, the rendering engine and the texture mapping are the most speed-critical parts, and we rarely rewrote those for speed. We rewrote them because they didn't do something right or we needed to do more with them. Every time we rewrote them, they would get faster, of course, because we would get better at it. So, the only rewrite that was specifically intended to speed the game up was the one that put the texture mapping into assembly. We never rewrote a huge part of the code just because it didn't run quickly. It was usually because it didn't do something we wanted it to do.

Q: These rewrites were more for design or game play issues than for performance.

A: Right. One of the reasons that Marathon was started only in May was that we were rewriting large parts of it that we were not happy with. We could have shipped it at the end of August, but we weren't happy with it, so we didn't. I think that was a really good decision. I would make the same decision again. If I'm not happy with something, we're not going to publish it. We're going to continue to do cool 3-D stuff and we are working on something now, but we're not talking about it.

Q: Will this something use advanced MacOS services like QuickDraw 3D?

A: No, it will not. The hardware acceleration options in QuickDraw 3D are very interesting to us. I'm really excited about having hardware acceleration for 3-D texture mapping. But the software services of QuickDraw 3D, which is primarily what is available now, are not as interesting to us because we'll always be able to outdo the software because we're trying to code a very specific case, and QuickDraw 3D handles the general cases.



Chapter 4
File Management

By file management, I mean any task that involves long-term information storage. For Mac games, this turns out to be more a question of avoiding unnecessary kludges than a hard problem, since the MacOS gives us some really great tools to work with. File management in games can be divided into the following subproblems:

- Where to store the pregenerated data (graphics, sounds, etc.)
- Where to save settings
- Where to save high scores and/or "save" files
- What data formats to use

The File Manager and the Resource Manager

First of all, let me say a few words on two MacOS managers: the File Manager and the Resource Manager.

The File Manager deals with files, directories, and volumes. You use it to create, open, and delete files, and to read and write to a file (actually to its data fork).

The File Manager has gone through many changes over the years, making it very hard to grasp because of the large number of available calls. There are several ways to specify a file, and which one you should use depends on how simple you want your code to be and how backward-compatible it should be. Generally, you must do a lot of error checking, partially because the user can delete, move, and rename files from the Finder while the program is running.

The Resource Manager, on the other hand, is pretty easy to use. It lets you store many pieces of structured information in the resource fork of a file. Its calls are simple and straightforward, and it seldom goes wrong.

In this section, I will show you how to use the File Manager to create, open, and delete files, and use the Resource Manager for reading and writing information.

Out of the many parts of the File Manager, we'll primarily use the calls with the `FSp` prefix. Those calls take a file specification record, the `FSSpec`, as a parameter which holds all information about a file in one record.

Perhaps you would rather use the standard library routines for the language you use? Believe me, those calls (`fopen`, `fclose`, etc. for C or `reset`, `rewrite`, etc. for Pascal) will just give you more problems. They work with the data fork, so you can't use resources with them. They know nothing of the file structure of the Mac, and won't work with Standard File dialogs without extra tweaking. Forget them! And if, despite my warning, you use the standard library routines, at least avoid mixing them with File Manager calls. Using both standard library routines and File Management routines on the same file at once will just make programming more complicated.

The standard library routines may seem good for making crossplatform code, but if that is your problem, make some high-level routines that use resources when on a Mac and plain files when run on other platforms—if you can't find some replacement for the Resource Manager for those other platforms.

The Two Forks

On the Macintosh, files are not just plain files, a long range of bytes, as they are on other platforms. On the Mac, a file is much more. To begin with, a Mac file can have two *forks*, the data fork and the resource fork. Either fork can be empty (see figure 4.1).

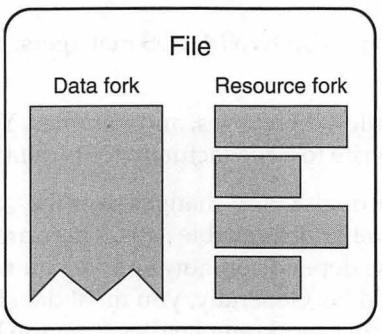


Figure 4.1 *A Mac file has two forks.*

The data fork is used for data just as on other platforms. That is where word processors store all the text. There is no structure that the operating system knows about; the file format is totally up to the applications. To read and write in the data fork, you use the File Manager.

The resource fork, on the other hand, has a strict format that is managed by the Resource Manager. The resource fork holds resources, which are blocks of data. A resource fork can hold thousands of resources, and the application using them does not have to bother with where each resource is stored—it simply asks for a resource and gets it. It is safe and it is simple.

A Quick Look into the File Manager

Even though you should use resources for practically everything, you may, in some rare occasions, need to read or write data to the data fork of a file—for example, if you want to save some information as a text file or make a save file in a format for another platform.

There is also a point when the Resource Manager just won't cut it any more. A file can hold up to 2,727 resources, so if you need more, you must either use several files or save some of the data in the data fork. Actually, it is recommended that you use an even lower upper limit on the number of resources in each file, around 500 resources.

If you flip through *Inside Macintosh: Files*, you will find that the summary alone is 63 pages. Yes, 63 pages full of routines, constants, and data types.

However, if you limit your interest in the File Manager to the few calls you really need, it isn't all that tough. Few games need to be able to do more than the following:

- Ask the player for a file to open (typically an old save file). You use `StandardGetFile()` for this.
- Open the data fork of a file. I would suggest `FSpOpenDF()` for this.
- Read data from an open file. Try `FSRead()`.
- Write data to a file. Try `FSWrite()`.
- Move around in a data fork. Have a look at `GetFPos()`, `SetFPos()`, `GetEOF()`, and `SetEOF()`.
- Close an open file. If it's a resource fork, try `CloseResFile()`, and if it's a data fork, try `FSClose()`.
- Ask the player where to save a new file. Use `StandardPutFile()`.
- Create a new file. `FSpCreate()` works just fine.
- Delete an existing file (done with some caution!). That is done with `FSpDelete()`.
- Initialize the resource fork of a file. That is done with `FSpCreateResFile()`.
- Open the resource fork of a file. I would use `FSpOpenResFile()`.

You get pretty far with this list. As you can see, practically all calls that specify a file have the `FSp` prefix. All these calls take a file spec record, the `FSSpec`, as a parameter. Other calls, particularly the older File Manager calls, require you to specify file name, directory ID, volume, working directory, and so on. It's tedious to move among all the different ways to specify a file, and it's easy to get lost. You only need to bother with all those older calls if you want to be compatible with System versions older than 7.0.

Resources and Handles

Even if you aren't familiar with resources and handles, you should know what a pointer is—the address of a memory block. Handles are not much more difficult to understand, and resources are used through handles.

What Is a Resource?

A resource is a piece of information, a block of data stored in the resource fork of a file.

A resource is described by its type (a 4 byte value, usually four readable characters), its ID number, and its name (which can be an empty string). To find a resource, you need to specify the type and *either* the ID number or the name. The ID number should always be unique, but the name might not be.

When loaded and used from your programs, resources are referenced through *resource handles*.

What Is a Handle?

A handle is a pointer to a pointer, which in turn points to a memory block (see figure 4.2). The most often-used type of handle is the Memory Manager handle. This handle points to a pointer in a master pointer block.

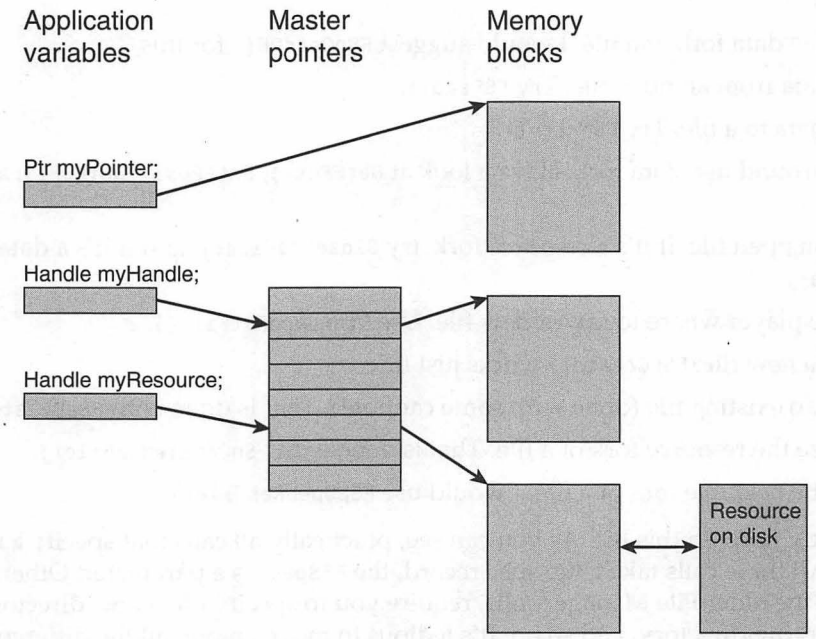


Figure 4.2 *Pointers, handles, and resources.*

On the Mac, you often use the Memory Manager handles instead of pointers for referring to dynamically allocated memory. The reason for this is that the Memory Manager may need to rearrange memory in order to free up larger blocks. Memory blocks referred to by pointers cannot be moved, because the Memory Manager has no control over your program's pointers. When using handles, the Memory Manager changes the master pointer, through which your handle refers to the moved memory block.

You can allocate handles as necessary to get memory as work space (with `NewHandle()`) and free them when you are through with them (with `DisposeHandle()`).

Many MacOS managers call the Memory Manager to create special-purpose handles. The one of most interest for us here is the Resource Manager, which uses *resource handles*. The difference between resource handles and ordinary handles is that the resource handle refers not only to a memory block, but also to a resource in a resource file.

Figure 4.2 illustrates the difference between pointers, handles, and resource handles. The pointer points directly to a memory block. Handles point to a master pointer, which in turn points to a memory block. For a resource handle, there is also a corresponding resource on disk.

You can detach a resource handle to turn it into a normal handle (using `DetachResource()`), or add a normal handle to a resource file (using `AddResource()`), turning it into a resource handle. Note that you can't do `DisposeHandle()` on a resource handle.

There is one more kind of handle to mention: pointers to normal pointers. When you take the address of a pointer variable, the result is a handle, but not one that the Memory Manager will recognize. It will not take part in memory reorganization. I don't consider such a handle a "real" handle, and you should not use it as such.

Loading and Creating Resources

Resources are loaded into memory by the calls `GetResource()` or `GetNamedResource()`. `GetResource()` finds a resource by its ID number, and `GetNamedResource()` finds it by its name.

If you try to load a resource that doesn't exist or doesn't fit in memory, the returned handle will be NIL, and the `ResError()` routine (the Resource Manager's error-reporting routine) will usually, though not always, return a "ResNotFound" error.

When you need to create a new resource, you first allocate the appropriate memory space with `NewHandle()`. That handle can then be made into a resource by calling `AddResource()`.

For examples, see ResourcesDemo later in this chapter.

The Resource Manager has a resource file path. When an application asks for a resource, the Resource Manager searches for the resource in the resource forks of the files in the resource file path. Most of the time, you deal only with resources in the resource fork of your application file.

There are variants of `GetResource()` and `GetNamedResource()` named `Get1Resource()` and `Get1NamedResource()`. The `1` variants search in the current resource file only, while the former search in the full resource file path, including your application and the System file.

Note
In the following section, I usually use the term "resource file" when I mean the resource fork of a file.

Using Resources and Handles

You access the data of the Memory Manager handles and resource handles the same way. Usually, you can use a type definition to a data structure. This is demonstrated in ResourcesDemo later in this chapter.

If you change the data in a resource, you must notify the Resource Manager about the change so that the change is written onto the disk file. That is done with `ChangedResource()` or by writing it immediately with `WriteResource()`.

When using handles of either type, there are some special cases to be aware of. Handles of any type can be locked or unlocked, and resource handles can be purgeable.

Locking Handles

If a handle is locked, the data it refers to will not be moved during memory reorganizations. This is necessary whenever you dereference a handle, that is, when you use an ordinary pointer to the data to which the handle refers. If you do that on an unlocked handle and the memory block gets moved by the Memory Manager, your pointer will point to the data's previous location in memory space instead of where your data has been moved to. This is a common cause of crashes.

You lock a handle with `HLock()` and unlock it with `HUnlock()`.

Locked handles are safer, but unlocked handles result in less memory fragmentation. Locking before dereferencing a handle and unlocking it as soon as the pointer in question is no longer used is generally preferable.

For games, however, the memory reorganizations by the Memory Manager are not always desirable because they can cause unpredictable delays. If you have many memory blocks referenced by handles, be they resources or normal handles, it may be a good idea to move all blocks that are loaded permanently (from the initialization of the application until it quits) as high as possible and lock them there. That is done by the call `MoveHHi()` (short for "move handle high"). This call moves them away from the part of memory where all temporary memory blocks go, and instead forms a tightly packed block of permanent, locked memory blocks.

Purgeable Resources

Resources can also be purgeable. That means that they may be disposed of from memory at any time by the System. When that happens, the master pointer will be NIL. Your resources should usually not be purgeable. To make sure, Get Info on your resources from within ResEdit and make sure the Purgeable checkbox is not checked. (In case you are unfamiliar with ResEdit, there is an introduction to it later in this chapter.)

It is also possible to let your resources be purgeable. If you do, the Resource Manager will be able to free extra space for you by purging resources, which can be considered a kind of Virtual Memory.

You can make a resource purgeable from within your program by calling `HPurge()`. When you need to use a purgeable resource, you check if it's purged, and if it is, you call `LoadResource()` to load it again. When a resource is purged, its master pointer is NIL, which you can check for as follows. If the handle is named `myHandle`, the check is

```
if (*myHandle == nil)
{
   LoadResource(myHandle);
}
```

You can also make a purgeable resource unpurgeable by calling `HNoPurge()`.

Using the Resource Manager as Virtual Memory like this is not necessarily the best solution. It might be better to release and load resources as you need them. If you let the Resource Manager do the purging, you have less control of what will be purged and when. The application memory may get reorganized often, slowing down your game.

Storing Pregenerated Data

Your game probably has quite a bit of pregenerated data, graphics and sounds in particular and possibly world descriptions as well. Storing those elements in the right place, to give both you and the users as few problems as possible, is very important.

For games that are meant to be stored on the user's hard disk, use *as few files as possible.* On certain computers with less advanced file handling (i.e., no resources), games tend to be broken up in dozens or even hundreds of small files. There is no need for that on the Mac. We have resources.

The ideal case is *one file,* the application itself. That is often all you need. That's good for the players, because there are no files that can get lost.

Some games have separate files for certain resources, such as sounds and graphics. There are some reasons programmers would give for this, but few of them are really convincing:

- The programmer has a tiny disk and can't afford to have all the resources copied into every incarnation of the game. Poor excuse.
- The programmer believes that the game will look bigger and more advanced if it has a bunch of files in a folder rather than a single file. Poor excuse.
- The programmer wants the users to be able to replace the application with newer versions without changing the other files, or replace sounds or graphics with alternate files. Both problems can be solved with an updater application. An updater will also be smaller and thus faster to download from archives. However, if you want users to hack your graphics or sounds and pass around copies without having to know ResEdit to use them, you have a point.
- The programmer wants the users to be able to throw away either the black-and-white or the color graphics to save space. Sure, it's nice of you to think about your user's disk space, but why not install either the black-and-white or color version from the original disks instead? Besides, few Macs have only black-and-white graphics these days. I don't think the black-and-white users will mind if they can't trash the color graphics, and the color users won't bother about the little memory space the black-and-white graphics take up.
- The programmer wants to make it possible to put each file on one floppy to make it easier to copy. This reason doesn't hold either. If your game is a shareware game, it shouldn't be that big—it's irresponsible toward all the kids who will get their modems taken away after downloading for too long. If your game is commercial, why encourage piracy? It should be possible to use an installer that handles multidiskette archives. Even a Compact Pro or StuffIt self-extracting archive can do that!
- The game has different scenarios, each one in its own file, and you can plug in more. Okay, I can buy that—but make each scenario *one* file.
- Some files are nonessential to the game and can be trashed to save space, such as QuickTime movies (e.g. the author.qt movie in Spectre Supreme). If it's nonessential, why is it there at all? But okay, allowing us to trash the junk once we've seen it can be a good idea.

- The programmer thinks it isn't good to have such large files and wants to split them into smaller files just to be safe. Well, how large are they? Perhaps you should make a CD-ROM instead?
- The game requires too many resources, e.g. over 500 or even over the limit of 2,727 resources in a file. With too many resources, the Resource Manager will lose performance, slowing down your game. That is a valid reason to have more files! Either split it into two or three files or change format for the data that require many resources.

Again, I recommend that your game is *one single file* (plus the player's save files and scenarios, if any). That is the most elegant solution. It will make the programming easier for you because there's no trouble keeping track of what file holds what data and setting the proper current file; there's also less error checking needed. The user will appreciate this solution because of less clutter on the hard disk and the lack of strange errors caused by a misplaced file.

If you have more than one file, keep them as few as possible and set the file creator of each to the one your game has, so a user who finds an unknown file can Get Info on it and see where it belongs.

If you make a huge CD-ROM game, the answer is, of course: Save it on the CD-ROM! It won't matter if you have a thousand files in a folder, as long as the user doesn't have to see them. The CD-ROM is read-only, so no files can get lost or misplaced.

Note
Save all pregenerated data in the application file if possible!

So, how do you store all that sound and picture data? Use resources! You will save lots of programming time, and all your data will be easily accessed. Some programmers use the data fork of their applications to store data. Don't! With the arrival of the Power Macs, the data fork of applications is now used for the Power Mac program code.

Note
Power Mac code can be stored in resources, but for such code, the Code Fragment Manager will not be able to give any memory savings the way it does for Power Mac code in the data fork of an application.

There is absolutely no pregenerated data that is better stored in the data fork than in resources in the resource fork. Graphics? Using `myPicHandle = GetPicture(kMyPict)` is a lot easier than opening and reading a PICT file. Sounds? You can get and play a sound resource in one line of code. Try doing that with an AIFF file. Icons? Same there, and surely, you didn't consider putting icons in a bunch of separate files, did you?

How about custom data then? Well, even that is better stored in resources than in the data fork of a file! Below, a demo program is listed that demonstrates how to create, change, and use a custom resource:

```
/* ResourcesDemo */
/* by Ingemar Ragnemalm 1995 */

/* Demonstrates how to use custom resources from your programs. */

/* A demo program that has a resource stored in the program file (should rather be ➥
in a */
/* preference file in the system folder in a real program) with a single variable, ➥
myCount. */
/* Each time the program is run, the resource is read, myCount is incremented, and */
/* the program beeps as many times as myCount says. */
```

The format of our resource is easily defined as a data structure and a handle type to it. In this particular resource, we store only a single variable, but we could have a lot of data in there.

```
/* Define a structure and a handle type to it. This is our resource format! */

typedef struct {
   short myCount;
} MyRecord;
typedef MyRecord **MyHnd;
```

It is always a good idea to define resource numbers and resource types as constants, so you don't hard-code it into your code. Avoid making custom resource types that are all caps ('DEMO') or all lowercase ('demo'), because they may be or become reserved as system resource types. This is particularly true for all lowercase types, which Apple has explicitly reserved. I used a mixed-case type ('Demo'), which should be safe.

```
/* Resource type and number. Use mixed-case names to avoid conflicts */

#define kMyResType   'Demo'
#define kMyResNum 0
```

I declare a handle to my resource.

```
MyHnd myResource; /* The handle to our resource */
```

Then I define a loop variable, just an ordinary short integer.

```
short i;          /* An integer for use as loop variable */
```

Then come the usual Toolbox inits (just a few of them this time)...

```
static void InitToolbox()
{
   InitGraf(&qd.thePort);
   InitCursor ();
   MaxApplZone ();
}; /*InitToolbox()*/
```

The main program starts, as it should, to init the Toolbox.

```
void main(void) {
   InitToolbox();
```

First, I try to get the resource.

```
/**** Get the resource ****/

   myResource = (MyHnd)GetResource(kMyResType, kMyResNum);
```

If it didn't load, I assume that it doesn't exist. If you want to be careful, you should check the `ResError()` routine to see what the problem is.

```
/**** Create the resource if needed ****/

/* If it didn't load, we assume it doesn't exist. Create it! */
/* This is done by allocating a handle, initializing the fields, and calling ➥
AddResource(). */
   if ( myResource == 0L )
      {
```


Before creating it on disk, I must allocate space in memory using `NewHandle()`.

```
      myResource = (MyHnd)NewHandle(sizeof(MyRecord));     /* Allocates memory */
```

Next, I initialize all fields to reasonable values. In this case, I just set the `myCount` field to zero.

```
      myResource[0]->myCount = 0;                          /* Init fields */
```

Finally, I can create the actual resource by calling `AddResource()`.

```
      AddResource((Handle)myResource, kMyResType, kMyResNum, "\p");       /* ➥
Create resource */
      };
```

Now let's move on to how to modify a resource. It's easy. Just change the values you like and call `ChangedResource()`! In this program, I increment the `myCount` field.

```
/**** Change the resource ****/

  (**myResource).myCount++;      /* Change the count */
/* Call ChangedResource() to tell MacOS that the resource needs to be written */
/* back to disk. */
  ChangedResource((Handle)myResource);
```

Finally, I will use the information in the resource. I call `SysBeep()` the number of times that `myCount` says. That's all, folks!

```
/**** Use the resource. Beep as many times as myCount says. ****/
  for ( i = 1 ; i <= (**myResource).myCount ; i++)
    SysBeep(1);

} /* ResourcesDemo */
```

Note that this demo saves the changing resource in the application itself. This is convenient, but not necessarily the best option for data that is to be changed. It depends on what kind of resource it is, but if it's a resource holding settings that are likely to change often, you should think twice about where to store it.

Saving Settings

Your game probably has a few settings, like sound on/off, direct-to-screen graphics on/off, key configuration, and so on. There is only one place where you should save these settings: in a preference file in the Preferences folder!

Yes, I know I said that you should use only one file, but that is for the pregenerated data. The preference file is something else. It is created by your game when it runs the first time, it can be lost with no major harm, and it is kept out of sight of the player.

You can save settings in the application file, but that will fail if your game is run from a read-only medium (e.g., CD-ROM) or is shared with AppleShare.

Is your game a commercial game that is installed on the user's hard drive and won't let itself be shared on a network? Do you think that is enough to make it safe to write to the application file? Sorry, but you still can't be sure. AutoDoubler makes all applications read-only when it compresses them (at least in older versions). Who knows what will come in the future? Besides, the user might lock it. The conclusion is that you must always assume that the application file might be read-only.

In the Preferences folder, you should create a file with an appropriate name (e.g., MyGame prefs) and with the same creator signature as your program. I will not go into depth on how to deal with directories, but instead give you a code snippet that opens or creates a preference file.

Note that you should use the `FindFolder()` call to get the folder. You should *not* look for a folder in the System folder named Preferences. The Preferences folder is named differently in different countries, and it is rather irritating to get one's System folder cluttered with several Preferences folders just because some programmer didn't care to learn the correct procedure.

Also note that `FindFolder()` often has glue code in the libraries that makes it work even on older System versions (e.g., System 6) where the actual call is not present. Whether or not such glue is implemented is often poorly documented. The following code assumes that the program using it either has such glue available or will only be used on Macs with System 7 or newer.

```
/* 1) Find the Preferences folder*/
/* 2) Check if the prefs file already exists. If it does, open it and exit. */
/* 3) Create the prefs file and open it.*/

Boolean SetPrefFile(Str255 prefsFileName, OSType prefCreator, OSType prefType, ➥
short *appFile, short *prefFile)
{
   OSErr err;
   short prefsFolder;
   Handle h;
   Str255 s;
   long dirID;
   short vRefNum;
   FSSpec spec;
   Boolean returnVal = false;
```


First of all, I get the current resource file with `CurResFile()`. The current resource file is the file in which the Resource Manager looks for resources first (or the only file it looks in, for certain calls). The calling application must keep this so it can restore it after using other resource files.

I also set the `prefFile` reference to zero, denoting that no file is found yet.

```
   *appFile = CurResFile (); /*get the reference number of the program's own ➥
resource fork*/
   *prefFile = 0;
```

I call `FindFolder()` to get the `vRefNum` (volume reference number) and `dirID` (directory ID number) of the Preferences folder. With the call `FSMakeFSSpec()`, I build a `FSSpec` from them, together with the preference file name.

```
/* 1) find the Preference folder*/
   err = FindFolder(kOnSystemDisk, kPreferencesFolderType, true, &vRefNum, &dirID);
/*Make a file spec to the pref folder*/
   err = FSMakeFSSpec(vRefNum, dirID, prefsFileName, &spec);
```

Next, I try to open the preference file. I use `FSpOpenResFile()`, which opens a file for use with the Resource Manager.

```
/* 2) Check if a prefs file exists. If it does, open it and return it. */
   *prefFile = FSpOpenResFile(&spec, fsRdWrPerm);
   if ( ResError () == noErr )
      {
/* Open succeeded. */
      }
   else
      {
         *prefFile = 0;
```

If I could not open the preference file, I assume that it does not exist.

I now try to create a new preference file using `FSpCreate()`.

```
/* 3) Create prefs file and open it.*/
/*Create the file*/
       err = FSpCreate(&spec, prefCreator, prefType, 0); /*smRoman = 0*/
       if ( err == noErr )
          {
```

`FSpCreate()` creates the file but not the resource fork, so I call `FSpCreateResFile()` to do that.

```
/*Make a resource fork for it*/
            FSpCreateResFile(&spec, prefCreator, prefType, 0); /*smRoman = 0*/
            if ( ResError () != noErr )
               {
/*Error: Couldn't create resource fork*/
               ;}
            else
```

If we managed to create it, we try to open the resource fork, again with `FSpOpenResFile()`.

```
/*Open it*/
               *prefFile = FSpOpenResFile(&spec, fsRdWrPerm);

            if ( ResError () == noErr )
                 return true; /* new pref file! */
         } /*Create succeeded*/
       else
       {
/*Error: Failed to create prefs file*/
       };
```

```
    }; /*No prefs file existed*/
  return returnVal;
}; /*SetPrefFile()*/
```

`SetPrefFile()` gives you the code for creating and opening the preference file. After calling it, you often need to switch between the preference file and the application file. Perhaps you want to copy resources from the application file to the preference file? I will discuss that in the next section.

Switching between Several Resource Files

Even though I said that you should avoid using several files, there are some cases in which you do need to use several. Typically, these include the preference file and save files.

If your program uses several files, preferably resource files, you need to switch between them to access the resources in the appropriate file. For example, if you open the preferences file, you get the settings resource from it, but then you may need to switch to a save file or to the main program. You do this with the call `UseResFile()`. You also often need to get the current resource file in order to restore it. That is done with `CurResFile()`.

As an example, here is a code snippet that copies a resource from one resource file to another. I originally wrote it for copying the default setting resource from my application file to my preference file.


```
/********** CopyResource() **********/

/* Error-handling macros for CopyResource(). */
/* If error, set back to the previous resource file and return */

#define CleanupAndExit  {UseResFile(oldResFile);return ResError();}
#define CheckError {if ( ResError()!= noErr) CleanupAndExit}
```

Note that `CheckError` and `CleanUpAndExit` are not functions, but macros without any parameters, and that is why they don't have any `()` at the end when they are used in the code. This is a trick to simplify the code that should be used with some caution.

```
/* CopyResource(): Copy a resource from one res file to another: */
OSErr CopyResource(short fromFile, short toFile, ResType theResType, short id)
{
  short oldResourceFile;
  Handle theResource, theResourceCopy;
  Boolean wasLoaded;
  short theID;
  ResType theType;
  Str255 theName;
  Boolean returnValue = noErr;
```

I save the current resource file number so I can restore it before leaving! If I didn't, the routines calling this routine may cause big problems, because they may be unable to find resources—assuming that the current resource file is what the routines expect it to be.

Then I set the current resource file to the source file.

```
oldResourceFile = CurResFile ();
UseResFile(fromFile);
CheckError;
```

Getting the resource could have been just a straight `GetResource()` call. However, this procedure is a bit more ambitious: When it returns, it leaves everything as it was, including leaving resources loaded or not loaded as they were. That means that I can't just unconditionally load the resource. I must check first if it were loaded before.

To find out if a resource is loaded, I call `SetResLoad(false)`, which means that subsequent calls to `GetResource()` will not load the resource if it isn't already loaded.

Caution

If you call `SetResLoad(false)` and do not set it back, you will get a system error later!

I get the resource with `Get1Resource()`. `Get1Resource()` is a variation on `GetResource()` that looks only in the current resource file. If I use `GetResource()`, the Resource Manager would search all files in the resource search path, which means that I might end up with a resource from the System file. With `Get1Resource()`, I know that I will get either a resource from the current resource file or none at all. In a function like `CopyResource()`, when I copy a resource from one specific resource file to another, I certainly don't want to get a resource from some other resource file by accident!

Then I call `SetResLoad(true)` to get back to the normal state.

```
   SetResLoad(false);
   theResource = Get1Resource(theResType, id);
/*Don't CheckError before doing SetResLoad(true)!!!*/
   SetResLoad(true);
   CheckError;
```

If the resource handle is not NIL but the master pointer that it points to is, then the resource exists but is not loaded.

```
if ( theResource != nil )
{
   wasLoaded = (*theResource == nil);
   LoadResource(theResource);
   CheckError;
```

Now I move over to the destination file—that is, set the destination file to be the current resource file.

```
UseResFile(toFile);
CheckError;
```

I call `GetResInfo()` to get the name of the resource. I already know the type and ID number, so I ignore those reports.

```
GetResInfo(theResource, &theID, &theType, theName);
CheckError;
```

With `HandToHand()`, I create a copy of the resource. The copy is not a resource—yet—but a handle to a relocatable memory block, which contains the same information as the resource.

```
theResourceCopy = theResource;
if ( HandToHand(&theResourceCopy) != noErr )
   CleanupAndExit;
CheckError;
```

With `AddResource()`, I add the copy to the destination file.

```
AddResource(theResourceCopy, theResType, id, theName);
CheckError;
```

Then I release it with `ReleaseResource()`, which tells the Resource Manager that we are done with the copy and it can dispose of it.


```
ReleaseResource(theResourceCopy);
```

If the source resource were not loaded before, let's release that too. (Note: This assumes that the application doesn't reload purged resources with `LoadResource()`, but rather uses `GetResource()`.)

```
    if ( ! wasLoaded )
      ReleaseResource(theResource); /*If it weren't loaded before, it shouldn't ➥
be afterwards.*/
    CheckError;
    returnValue = noErr;
  }
else
  {
    returnValue = resNotFound;
  };
  UseResFile(oldResourceFile);
  return returnValue;
} /*CopyResource()*/
```

This code snippet is part of the preference file handling unit in the ScoresDemo program, which you will find on the CD-ROM in the 04 - File Management folder. The snippet is not actually used by ScoresDemo, but provided as a utility for the preference handling unit.

Saving High Score Lists

Many games, classic arcade games in particular, use some kind of high score list. Handling a ten-entry straight high score list is a simple task, but before you settle for that easy way out, you may want to consider some alternatives.

From the player's point of view, the big drawback with the simple ten-entry high score list is that it tends to be filled with only one name, the name of the best player in the house. The simplest solution to that is to make the list longer, say 30 items. That will help a little. Now, Big Brother will hold the 15 top scores; Big Brother and Little Brother have a few each of places 16 to 25; Daddy has a few scores after place 25; however, Little Sister can't be seen on the list. Her scores disappeared from the list a few weeks ago, and now she finds it so humiliating to try to compete that she has given up.

A nice but rarely seen solution is the one found in the shareware game Continuum, made by Randy Wilson in 1987. In Continuum, each player has one entry in the high score list (see figure 4.3). The list is also used for starting the game: You click on your name and the game starts. This idea can be expanded to give a high score list of, say, ten entries to every player.

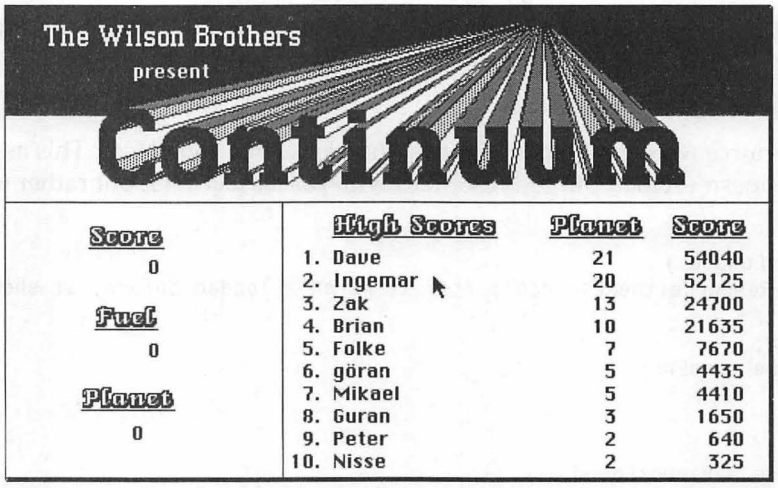


Figure 4.3 *The high score list in Continuum allows one entry for each player.*

Another variation is to list "today's high scores," as Crystal Quest does. The advantage is that you can run competitions over a day and the less skilled players will at least show up on that list.

Okay, now we've agreed that there are a few ways to make a high score list. Next question: Where do you store it?

First, I recommend that you use resources for the high scores. That is extremely easy to program.

In what file, then? The safest bet is a file in the Preferences folder, in the same file as your settings. This has one drawback from a game player's point of view: If the user moves the game, perhaps to another computer or to floppies during a period when it won't be used, the high scores won't come with it.

You can also save high scores in the application itself. This may not always work, for reasons mentioned in the section on saving settings.

The third possibility, a separate file in the same folder as the application, has most of the same drawbacks as saving high scores in the application itself. Besides, it messes up the user's game folder.

You could save high scores in a place the user picks, and then save a pointer to that folder in the Preferences, but that's really a kludge. Don't do it.

I recommend that you save your high scores in the Preferences folder, in your preference file, since that's what Apple tells us to do. You may also want to consider some "export/import high scores" feature to help the user save the high scores without having to dig in the often horribly crowded Preferences folder.

I'll end this section with some sample code, implementing a simple high score list and saving it in a resource. The list is just a plain ten-entry list of the standard kind.

The following snippet, which is a part of the ScoresDemo program, contains two procedures, `InitScores()` and `UpdateHighScore()`. `InitScore()` should be called during startup to load the high score list into memory from the preference file. `UpdateHighScore()` checks if a given score is a high score, and if it is, it asks for the player's name with a dialog box (not included below) and inserts it in the list.

```
/****************** High score handling ******************/

#define kNameLength 15 /* Must match the length of the name string type */
```

In a global, I keep the index of the latest high score in order to display that in boldface or a color.

```
static short gLastHigh;
```

I declare a structure that defines my resource format, a handle type to it, and a variable of that type. This is our high score list.

```
/* High score record type */

typedef struct
{
   long score[11];
   Str15 name[11];
} HighScoreRec;
typedef HighScoreRec **HighScoreHnd;

/* Global handle to the high score resource */
HighScoreHnd gHighScoreHandle;
```

When I have a new score that I want to test against the current high scores, I call the following routine, `UpdateHighScore()`. It takes the new score as a parameter.

```
/* Call this on game over! */
static void UpdateHighScore(long score)
{
   short rank;
   Str255 name;
```

I check if the new score is higher than the last entry in the list. If not, I have no more business here.

```
   if ( score > (**gHighScoreHandle).score[10] )
   {
      rank = 10;
```

I call a routine `AskHigh()` that asks the player for a name. This routine is not listed here.

```
/*Call some function that asks for the name.*/
     AskHigh(name);
```

Since I use `Str15`s in my structure above, I cannot allow longer strings than that. If necessary, I truncate it by changing the length byte.

```
/*Max kNameLength characters! We take some extra trouble to append '…' too.*/
     if ( name[0] > kNameLength )
        name[0] = kNameLength;
     if ( name[0] == 0 )    /* length of name = 0 */
        return;
```

Now I start walking up the high score list. As long as the entries I find are smaller than my own, I copy the name and the score from the next highest entry down to the current position.

The call `CopyString()` is a routine I wrote myself that copies a Pascal string. The C library routines (`strcpy()` and others) work only on C strings, so they are useless (or at least awkward) for this purpose.

```
/*Decrement rank until we get to a position at which the one above is better!*/
     while (((**gHighScoreHandle).score[rank - 1] < score) && (rank > 1))
        {
          (**gHighScoreHandle).score[rank] = gHighScoreHandle[0]->score[rank - 1];
          CopyString( (**gHighScoreHandle).name[rank] , gHighScoreHandle[0] ➥
->name[rank - 1]);
          rank—;
        };
```

Now I have the right position for my new score. I remember it in `gLastHigh` for the display.

```
/*Remember last high for the high score display*/
     gLastHigh = rank;
```

Finally, I write in the score and the name into the proper place and notify the System that it should write the resource to disk by calling `ChangedResource()`.

```
/*Write in the score in the proper place and save*/
     (**gHighScoreHandle).score[rank] = score;
     CopyString( (**gHighScoreHandle).name[rank] , name);
     ChangedResource((Handle)gHighScoreHandle);
   }
} /*UpdateHighScore()*/
```

The next routine, `InitScores()`, loads the high score list from a resource and creates it if necessary. This is done the same way as in ResourcesDemo, but with more fields.

```
/*InitScores()*/
/*This procedure loads the high score list from a resource in the current */
/*resource file, creating a new one if there is none.*/

static void InitScores()
{
   short rank;
```

Get the resource if it exists.

```
   gHighScoreHandle = (HighScoreHnd)GetResource('Bäst', 0);
   if ( gHighScoreHandle == nil )     /*Didn't exist -- create it!*/
   {
```

If none is found, then allocate memory for a new one.

```
     gHighScoreHandle = (HighScoreHnd)NewHandle(sizeof(HighScoreRec));
```

If it couldn't create it, then it must be out of memory! In a real program, I would give an error message here.

```
if ( gHighScoreHandle == nil )
  ExitToShell();        /*Insert error message here*/
```

I fill in the resource with meaningful data.

```
for ( rank = 1 ; rank < 10; rank++ )
{
  (**gHighScoreHandle).score[rank] = 0;
  CopyString( (**gHighScoreHandle).name[rank] , "\pNobody");
};
```

I create the resource with `AddResource()`. Note that this code assumes that the calling application has set the right resource file, which probably is the preference file.

```
  AddResource((Handle)gHighScoreHandle, 'Bäst', 0, "\pHigh scores");
}
```

Now, let me suggest a little safety trick. If I indeed got a resource, I check that it has the right size with `GetHandleSize()`. If it is smaller than `sizeof(HighScoreRec)`, the size of the data structure, I call `SetHandleSize()` to enlarge it.

Will the resource ever be too small? It just might! Many times when developing a program, I have added fields to my resource formats. If the program then uses the old resource, it will write outside its memory space into whatever is after it. This is a sure way to get mysterious errors—and it is easy to avoid.

```
  else /*Did exist -- check the size!*/
    if (GetHandleSize((Handle)gHighScoreHandle) < sizeof(HighScoreRec))
      SetHandleSize((Handle)gHighScoreHandle, sizeof(HighScoreRec));
} /*InitScores()*/
```

The routine `InitToolbox()` does the standard inits again. I've skipped it here because it's listed in previous demos.

Finally, here is the main program for ScoresDemo:

```
void main(void) {
  WindowPtr myWindow;
  Str255 tempString;
  Rect windowRectangle;
  short score, rank;
  short appFile, prefFile;

  InitToolbox();
```

I call `SetPrefFile()`, which is listed earlier in this chapter, to get a preference file.

```
/*Get the prefs file -- create as necessary.*/
   if ( SetPrefFile("\pScoresDemo Preferences", '????', 'pref', &appFile, ➥
&prefFile) )
      ;
```

I set the resource file to the preference file and call `InitScores()`. Afterwards, I set the resource file back to the application's resource fork.

```
/*Set the current resource file to the preference file before InitScores()*/
   if ( prefFile != 0 )
      UseResFile(prefFile);
   InitScores();
   UseResFile(appFile);
```

I seed the random number generator and generate a random positive value as my score.

```
/*Let's seed the random number generator so we don't get the same all the time!*/
   qd.randSeed = TickCount ();

/*Update the high score list with a random score.*/
   score = Random();
```

I send the `score` to `UpdateHighScore()`.

```
   UpdateHighScore(abs(score));
```

I set up a window and make it the current port.

```
/*Set up the window*/
   SetRect(&windowRectangle, 100, 100, 400, 260);
   myWindow = NewCWindow(nil, &windowRectangle, "\pHigh scores demo", true, 0, ➥
(WindowPtr)-1L, false, 0);
   SetPort(myWindow);
```

In the window, I write the high score list. When the current score is equal to `gLastHigh`, that score should be drawn in boldface or red. In this case, I use red.

```
   for ( rank = 1 ; rank < 10; rank++)
   {

/*We draw the latest high score in red.*/
      if ( rank == gLastHigh )
         ForeColor(redColor);
      else
         ForeColor(blackColor);

/*Draw the position number*/
      MoveTo(10, 15 * rank);
```

```
        NumToString(rank, tempString);
        DrawString(tempString);

    /*Draw the name*/
        MoveTo(50, 15 * rank);
        DrawString((**gHighScoreHandle).name[rank]);

    /*Draw the score*/
        MoveTo(200, 15 * rank);
        NumToString((**gHighScoreHandle).score[rank], tempString);
        DrawString(tempString);
      };
```

Finally, I wait until the user clicks the mouse button, so the high score list does not disappear immediately.

```
  while ( ! Button () )
    ;
} /*ScoresDemo*/
```

This code is just the plain standard ten-entry type, but do not let that limit you. More elaborate high score systems can be made following the same principles.

Saving the Player's "Save" Files

In games that take a long time to play all the way through in one session—for example, adventure games, strategy games, and puzzle games—the player must be able to save the results in order to continue later. In different games, you find several different approaches to this problem:

- Save files defining the current state
- Internal saving of the game state
- Passwords to access a certain level

Save Files

The user gets a Standard File dialog and can save the results/game state in a file with any name in any place. To continue from a saved game, the user gets a Standard File dialog and chooses the file or simply double-clicks on it in the Finder.

This is a very nice solution with which users will feel comfortable. However, it will clutter their disks a bit. It will also give them plenty of opportunity to cheat by going back to an older save file. Whether or not this is desirable depends on the game. For most games, this is the solution you should use.

Internally Recorded Game State

The game saves the game state in a private place, typically in the preference file in the Preferences folder. The player may access a saved game by picking the name from a list of active players or games or by typing in the name. Picking from a list is good. Having to type in the name is not so good. On a Mac, we should work by recognition, not by recall.

Some games use a limited number of save slots and you select one of the slots. That, however, is not a solution to the problem, but rather an insult to the players; it suggests a programmer who didn't make an effort to make a nice and flexible solution or who has mimicked the Save interface from some machine with lower standards than the Mac. If there isn't any particular reason that your game should limit the number of saved games, don't limit it.

Passwords

Some games use passwords. This solution is used mostly in puzzle games as level passwords. (Multiplayer strategy games may have password-protected save files, but that is a different matter.) Using level passwords is excusable if you develop a game for a toy computer with no secondary memory. This, however, is a Mac. It's a real computer. You can take for granted that users have hard disks, and they definitely have diskettes. There is no reason to demand that the user should have to use a piece of paper to write down a stupid password.

Don't make us think that you don't know how to save—don't use level passwords! The computer has all the power needed to help the user. Let it.

Note

The computer should help the user save as often as the user likes, and make that as simple as possible.

Dungeon 3

To show an example of how to handle save files, I have expanded on Dungeon a bit more. Dungeon 3 does all that Dungeon 2 does, plus adds save file handling. It also processes Apple Events in order to accept save files dropped on the game file from the Finder. When you're running the game, all you notice is that a few menu items have been added to the File menu (see figure 4.4).

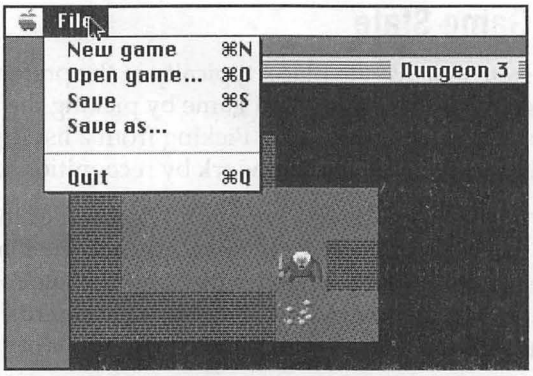


Figure 4.4 *Dungeon 3, which supports save files!*

I use the same calls as for the preference file, except that I also use Standard File dialogs for selecting files to load and selecting file name and folder when using Save as (see figure 4.5).

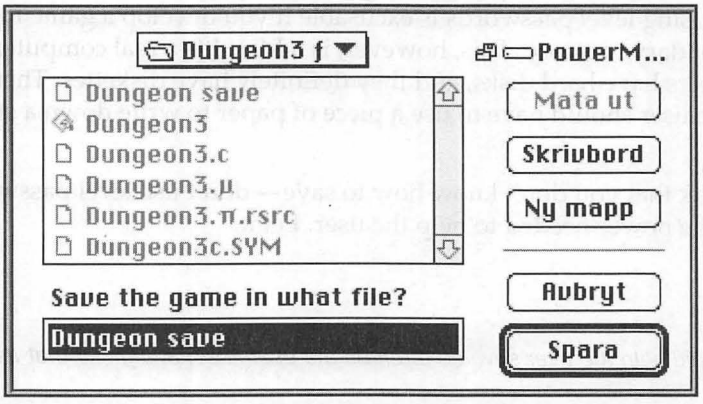


Figure 4.5 *A Standard File dialog (for saving).*

Note

If you think the dialog in figure 4.5 has weird names on some buttons, that's because I use Swedish System software on my Mac.

The first part of the added sources are the routines for file management: `SaveGame()` and `OpenGame()`. Both take a parameter that tells whether or not a Standard File dialog should be shown. There are four different cases:

- Open Document from an Apple Event—In this case, no Standard File dialog should be shown.

- Open from the menu—Always shows a Standard File dialog.
- Save from the menu—Shows a Standard File dialog if there is no old file spec that should be used.
- Save as from the menu—Always shows a Standard File dialog.

Dungeon 3's Save File Management Routines

First of all, I move all the data that should be saved to a game state structure, define a handle type, and declare a handle—to be able to write all this information straight to a resource.

```
/*Variables defining the game state*/
typedef struct {
   TileState tileArray[kArraySizeH][kArraySizeV];     /* What is in each tile? */
   Boolean tileKnown[kArraySizeH][kArraySizeV];       /* What tiles have we seen? */
    Point playerPosition;                             /* Current position */
    short playerHitPoints;                            /* How much health left? */
   } GameState;

typedef GameState **GameStateHnd;                     /* Define a handle type to it */


/* Define a handle for the game state */
GameStateHnd game;
```

I could copy all this information from separate variables to the structure when saving, but that would just be more trouble. I could also consider separating it into several resources. This option is most interesting when I have some data that is of variable length. Then, I could put all fixed-length data into one resource and variable-length data into others.

I define the creator and type codes for the save files. The type should be anything that is unique. The creator code should be the same as the creator code for the application. I use `'????'` since this is just a demo.

```
/*The creator code should match the one of the application.*/
/*The type should be defined in the 'BNDL' resource in the app.*/

#define saveFileCreator   '????'
#define saveFileType 'DSav'
```

Note

Once you want your own creator code, i.e., when you want custom icons and the ability to double-click save files to open them, you should register the creator code with Apple. If you have email, you can mail to DEVSUPPORT on AppleLink.

The `StandardFileReply` is a structure to which the Standard File routines return results. I make it global in order to save the last file specification so I can use it for Save.

```
StandardFileReply lastReply;
```

`SaveGame()` is called when the player selects Save or Save as.

```
/*Save/Save as:
The Boolean is TRUE if we are required to show a Standard File dialog.*/
static void SaveGame(Boolean as)
{
   StandardFileReply reply;
   SFTypeList typeList;
   OSErr err;
   short oldFile, fileRef;
```

If Save is selected and `lastReply` holds a valid file reference, I use that. Otherwise, I fire up a Standard File save dialog with `StandardPutFile()`.

```
  if ( ! as && (lastReply.sfGood) )
   reply = lastReply;
  else
   StandardPutFile("\pSave the game in what file?", "\pDungeon save", &reply);
```

If the reply is good (i.e., if the player didn't cancel), I go on saving the game. First of all, I save the old resource file.

```
  if ( reply.sfGood )
   {
    oldFile = CurResFile ();
```

If the file already exists, I delete it with `FSpDelete()`. This is not a 100 percent safe solution, however. If I wrote a program that saves valuable information, I would not delete the old file until the new one is successfully written. I could, for example, move or rename the old file.

```
/* Delete in case there's an old file */
    err = FSpDelete(&reply.sfFile);
```

Next, I create the save file with `FSpCreate()` and its resource file with `FSpCreateResFile()`.

```
    err = FSpCreate(&reply.sfFile, saveFileCreator, saveFileType, 0); /*smRoman = 0*/
    FSpCreateResFile(&reply.sfFile, saveFileCreator, saveFileType, 0); /*smRoman = ➥
0*/
```

I open the file with `FSpOpenResFile()` and make it the current resource file.

```
    fileRef = FSpOpenResFile(&reply.sfFile, fsRdWrPerm);
    UseResFile(fileRef);
```

Actually saving the game is very easy now. I simply create a resource from my game using `AddResource()`, tell the Resource Manager to write it immediately by calling `WriteResource()`, and then call `DetachResource()` to make it a nonresource handle again.

```
    AddResource((Handle)game, 'Save', 0, "\pGame save information");
    WriteResource((Handle)game);
    DetachResource((Handle)game);
```

Finally, I set back the resource file, close the save file, and save the reply from Standard File into `lastFile`.

```
    UseResFile(oldFile);
    CloseResFile(fileRef);

    lastReply = reply;
   };
 }; /*SaveGame()*/
```

The next routine is `OpenGame()`, the counterpart to `SaveGame()`.

```
/*Open game:
The Boolean is TRUE if we should show a Standard File dialog */
static void OpenGame(Boolean as)
{
   StandardFileReply reply;
   SFTypeList typeList;
   OSErr err;
   short oldFile, fileRef;
```


I use `lastReply` or a Standard File dialog, depending on what the caller asks for. I use `StandardGetFile()` to get an existing file. Before calling it, I fill in a field in a type list to tell `StandardGetFile()` what file types to accept.

```
  if (as) {
     typeList[0] = saveFileType;
     StandardGetFile(nil, 1, typeList, &reply);
     }
  else
     reply = lastReply;

  if ( reply.sfGood )
   {
    oldFile = CurResFile ();
```

To open the resource file, I call `FSpOpenResFile()`, which returns the file reference number `fileRef`. Then, I set the current resource file to `fileRef` with `UseResFile()`, as usual.

```
    fileRef = FSpOpenResFile(&reply.sfFile, fsRdPerm);
    UseResFile(fileRef);
```

Since I am loading an old game, I don't need the current game state. I dispose of it with `DisposeHandle()`.

```
DisposeHandle((Handle)game);
```

I get the game state from the file with `GetResource()`.

```
game = (GameStateHnd)GetResource('Save', 0);
```

I detach it so it is no longer a resource handle.

```
DetachResource((Handle)game);
```

I restore the resource file, close the save file, and save the reply in `lastReply`.

```
UseResFile(oldFile);
CloseResFile(fileRef);
lastReply = reply;
```

Finally, I call `InvalRect()` to force an update event for the entire game window. An alternative would be to just redraw everything right away.

```
/*Force an update of the entire window*/
    SetPort(myWindow);
    InvalRect(&myWindow->portRect);
   };
 }; /*OpenGame()*/

/* End of file management routines */
```

Now I have gone through all the code that actually handles the save files. The actual file management is fully covered, but there is one more thing we need to add to make our save files work perfectly: Apple Events.

Apple Events in Dungeon 3

Apple Events are high-level events used for sending messages between applications, even over networks. An application can be written with no Apple Event support and work just fine, but your application will work more as expected if it supports at least the required Apple Events.

There are four "required" Apple Events:

- Open Application (`kAEOpenApplication`) occurs when the application starts.
- Open Document (`kAEOpenDocuments`) occurs when a document is double-clicked on in the Finder or is dragged onto the application.
- Print Document (`kAEPrintDocuments`) occurs when Print is selected in the Finder with a document belonging to my application.
- Quit Application (`kAEQuitApplication`) occurs when the Finder wants your application to quit.

The Open Application event is sent when the program starts up if there are no Print or Open Document events. I could use this to make "New Game" when it arrives. In this demo, I just ignore it and make "New Game" at startup no matter what events may be coming. Printing is usually not supported by games, at least not from the Finder, so I ignore Print Document events too. That leaves us with the two Apple Events that are interesting to support: Open Document and Quit Application.

```
/* ******* MultiFinder and Apple Events: ******* */
```

`MyGotRequiredParams()` is a little routine that tells whether or not we have done everything we are supposed to with an Apple Event.

```
/*MyGotRequiredParams(): tells whether we have handled all we have to or not.*/
static OSErr MyGotRequiredParams(const AppleEvent *theAppleEvent)
{
  DescType returnedType;
  Size actualSize;

  if ( AEGetAttributePtr(theAppleEvent, keyMissedKeywordAttr, typeWildCard, ➥
&returnedType, nil, 0, &actualSize) == errAEDescNotFound )
    return noErr;
  else
    return errAEParamMissed;
};
```

As mentioned above, I ignore Open Application events, but even to ignore it takes a few lines of code.

```
/* Open Application Apple Event received */
static pascal OSErr DoOpenApp(const AppleEvent *theAppleEvent, AppleEvent *reply, ➥
long refCon)
{
/*What am I supposed to do here?*/
    return MyGotRequiredParams(theAppleEvent);
};
```

Here is how I handle an Open Document Apple Event.

```
/* Open Document Apple Event received */
static pascal OSErr DoOpenDoc(const AppleEvent *theAppleEvent, AppleEvent *reply, ➥
long refCon)
{
    OSErr  err;
    AEDescList  fileSpecList;
    short  i;
    long count;
    Size actual;
    FSSpec desc;
    AEKeyword keyword;
    DescType  type;
```

`AEGetParamDesc()` extracts the file spec list.

```
err = AEGetParamDesc(theAppleEvent, keyDirectObject, typeAEList, &fileSpecList);
```

With `AECountItems()`, I extract the length of the list. I really just need one, though.

```
err = AECountItems(&fileSpecList, &count);
```

I call `AEGetNthPtr()` to extract a file spec. I have a loop just in case one would fail, but as soon as I have one, I continue. There's no point in getting more, because this game can have only one file open at a time.

```
    for (i = 1; i <= count; i++) {
      err = AEGetNthPtr(&fileSpecList, i, typeFSS, &keyword, &type, (Ptr)&desc, ➥
sizeof(FSSpec), &actual);
      if (err == noErr) {
```

I copy the file description into the `lastReply` structure, because that is where `OpenGame()` looks for an existing file spec. Then, I call `OpenGame()`.

```
/* Copy the file description into lastReply so Open can get it from there. */
          lastReply.sfFile = desc;
          lastReply.sfGood = true;
          OpenGame(false);
/* Since we allow only one file at a time, let's exit DoOpenDoc() once we have ➥
one. */
          return MyGotRequiredParams(theAppleEvent);
          }
      }
    return MyGotRequiredParams(theAppleEvent);
}; /* DoOpenDoc() */
```

Next follows my Print Document handler. I just reply that it is not handled, since I don't support it.

```
/* Print Apple Event received */
static pascal OSErr DoPrintDoc(const AppleEvent *theAppleEvent, AppleEvent *reply, ➥
long refCon)
{
  return errAEEventNotHandled; /*We don't print any documents!*/
};
```

A Quit Application Event is easy to handle. I just set `gDone` to TRUE.

```
/* Quit Apple Event received */
static pascal OSErr DoQuitApp(const AppleEvent *theAppleEvent, AppleEvent *reply, ➥
long refCon)
{
  gDone = true;       /*If I'm told to quit, I'll quit.*/
  return MyGotRequiredParams(theAppleEvent);
};
```

During initialization of my program, I call `AppleEventInit()`, where I install all my Apple Event handlers with the `AEInstallEventHandler()` call.

```
/*Initialize Apple Events*/

static void AppleEventInit()
{
     OSErr error;

     error = AEInstallEventHandler(kCoreEventClass, kAEOpenApplication, ➥
&DoOpenApp, 0, false);
     error = AEInstallEventHandler(kCoreEventClass, kAEOpenDocuments, &DoOpenDoc, ➥
0, false);
     error = AEInstallEventHandler(kCoreEventClass, kAEPrintDocuments, ➥
&DoPrintDoc, 0, false);
     error = AEInstallEventHandler(kCoreEventClass, kAEQuitApplication, ➥
&DoQuitApp, 0, false);
/*I ignore errors.*/
};

/* End of Apple Event handling */
```

There is much more to say about Apple Events, but these are the only four that Apple insists every application must respond to. The previous code is enough to make a game behave acceptably.

Other Changes in the Dungeon Source

Since I moved a bunch of variables into a structure, there are lots of changes all over the code, but usually just by adding `(**game).` before all occurrences of `tileArray`, `tileKnown`, `playerPosition`, and `playerHitPoints`.

Finally, there are a few minor changes. The game state must at some time be allocated. I do that in `InitDungeon()`:

```
/* Allocate the game world */
     game = (GameStateHnd)NewHandle(sizeof(GameState));
```

The File menu has gotten longer, so I add a call to `OpenGame()` or `SaveGame()` for each item.

```
/* Handle selections in the File menu */

static void DoFileMenu(short item)
{
  switch ( item )
  {
     case 1:
       NewGame();break;       /* New */
```

```
    case 2:
      OpenGame(true);break;  /* Open */
    case 3:
      SaveGame(false);break; /* Save */
    case 4:
      SaveGame(true);break;  /* Save as */
    case 6:
      gDone = true;break;    /* Quit */
  }; /*case*/
}; /*DoFileMenu()*/
```

The event loop must handle Apple Events. This is a very simple addition to the `MainLoop()` routine:

```
      case kHighLevelEvent:
        if (AEProcessAppleEvent(&theEvent) != noErr)
          { /* error -- ignored */ };
```

Finally, `AppleEventInit()` must be called at some point during startup. That is done in the main program.

```
/* Main program */

void main(void)
{
  InitToolbox();
  AppleEventInit();
  InitDungeon();
  SetupMenus();
  NewGame();
/*Initializations done! Run the game loop until the game ends.*/
  do
    {
    MainLoop();
  } while (!  gDone);
} /*Dungeon*/
```

I have also changed the File menu resource, adding the new items. That menu looks almost like a standard File menu now! All that it lacks is printing, and Dungeon sure doesn't need to print anything!

As you see in Dungeon 3, handling save files is a question of using not only Standard File dialogs and File Manager routines, but also Apple Events. Many game programmers skip the Apple Event part, but the result is that their games don't handle double-clicks on save files in the Finder. Using Apple Events, you make that work as the users expect it to.

Using Data Formats

So far, I have only discussed custom resource formats. However, there are plenty of standard formats for both resources and files. Actually, there are so many different formats that it is impossible to cover them all. In this section, I discuss some formats that are of interest for you, especially for pregenerated data.

Since I recommend that you store just about everything as resources, the data formats I recommend are resource formats. You should use standard resource formats as much as possible, except for data that are secret and must not be readable.

Putting Together Your Resource File with ResEdit

The standard tool for manipulating resource files is ResEdit. ResEdit is short for resource editor. It is used for constructing and editing resource files, particularly by programmers. It is not exclusively a programmer's tool, though. A normal user can have great fun with ResEdit, changing the Command-key equivalents of menus, translating strings to another language, changing graphics and sounds, and so on. The applications of this program are countless.

In theory, ResEdit can edit any kind of resource, using its "hex editor" to edit resources by the hexadecimal values of their contents. It also includes many editors for specific resource formats, such as icons, dialogs, and strings. There is even a disassembler, the 'CODE' editor, if you feel like trying to find out how a program works and you don't have any source code for it. That's not for the faint-hearted, however.

You probably already have a copy of ResEdit in your programming toolkit, but if not, you can find it on `ftp.info.apple.com`. The lovely and talented icon looks like this:



You should copy ResEdit to your hard disk if you haven't got it already. You will be using it a lot. I always have an alias to ResEdit on the desktop so I can drag and drop files onto it.

However, ResEdit is not perfect. In case you find ResEdit lacking in the long run, there is an alternative that is quite a bit better in many ways, namely Resorcerer. It is not free, but can be well worth the money.

If you start ResEdit by double-clicking on its icon, you first get an amusing welcome screen with the animated Jack-in-the-Mac. After you click once to get rid of the welcome screen, ResEdit shows a Standard File dialog, allowing you to open an existing file or create a new one. Click on New to create a new file. ResEdit then shows the Standard File save file dialog. Select a folder, type a name, and click on the New button.

Note

By convention, resource files in programming projects end with .rsrc. This is no rule written in stone, since file extensions are usually not required on the Mac. But if you share code with other programmers, they will like it if you use this naming convention.

You now get an empty window, because the new file is empty, that is, contains no resources. Depending on what kind of resource you want to create, you create them in one of three ways:

- Select Create New Resource (Command-K) from the Resource menu to create a new resource.
- Open some other file from ResEdit and copy resources from there.
- Copy data from some other program, switch to ResEdit, and paste.

You use Create New Resource for resources that you can edit from within ResEdit. They include icons ('cicn'), icon families ('ICN#'), finder information ('BNDL' and 'vers'), string lists ('STR#'), and many others.

The standard System sound files are resource files, so you can copy resources from those files to other resource files by opening them from ResEdit, copying, and pasting.

PICTs cannot be edited in ResEdit, so you should draw them in some drawing program, copy, switch to ResEdit, and paste.

Navigating in ResEdit

If you open an existing file, a window opens in which you see icons representing all the resource types occurring in the file. This is the top level of the file. An example is shown in figure 4.6.

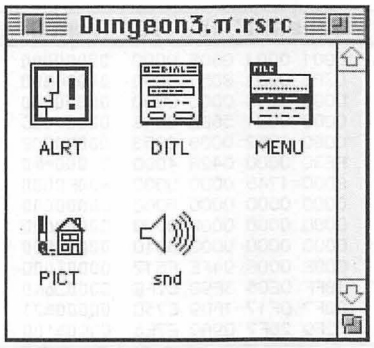


Figure 4.6 *At the top level of a resource file, ResEdit displays icons of all resource types represented in the file.*

If you double-click on a resource type, you get a list of all the resources of that type (see figure 4.7). This is usually a straight list with ID numbers, sizes, and names, although for some resource types you get a different display.

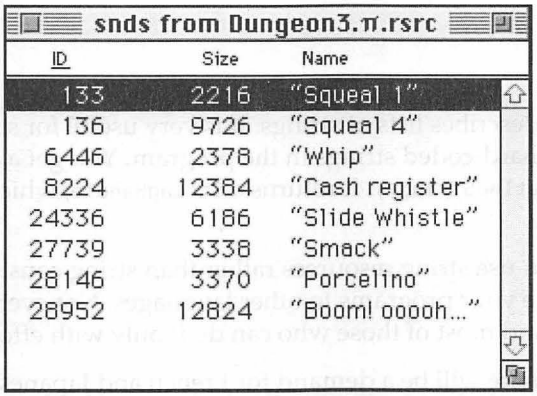


Figure 4.7 *A list of all resources in the file, in this case sounds ('snd ').*

If you double-click on a resource, the resource is opened in the editor for that resource type.

It is possible to view any resource in hexadecimal form by selecting Open using hex editor from the Resource menu instead of double-clicking on a resource. That is usually not what you want, but it may be useful in some situations, when you want to know exactly what value each byte of a resource has. Figure 4.8 shows a sound opened in the hex editor.

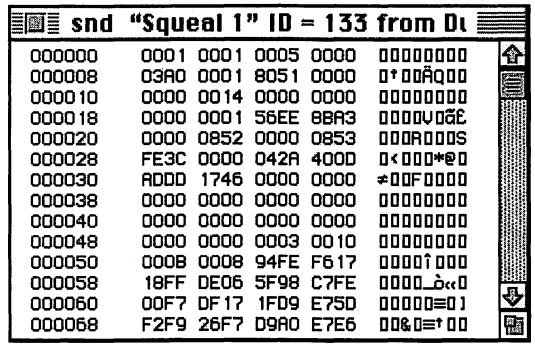


Figure 4.8 *The hexadecimal resource editor.*

Obviously, this editor is not for editing anything. Personally, I rarely *edit* in the hex editor, but I sometimes use it for inspecting resources to see what format they have. Most of the time, you'll use resources for which there are much nicer editors.

Now it's time to move over to those standard resource formats and learn what they are good for.

'STR#' Resources

The 'STR#' resource format describes lists of strings. It is very useful for storing strings in resources rather than using hard-coded strings in the program. You get a string from a 'STR#' resource by calling `GetIndString()`. It returns a `StringHandle`, which is a handle to a `Str255`, that is a Pascal string.

I highly recommend that you use string resources rather than string constants. If you do, you make it very easy to translate your programs to other languages. Not everyone in the world speaks English, you know, and most of those who can do it only with effort.

If your game gets popular, there will be a demand for French and Japanese versions, for example. If you can provide those versions with little effort, you can win quite a few extra customers. You don't necessarily have to do any work on translations yourself. Just provide the users with the possibility. It's a good service that might widen your market.

Adding String Lists

Adding a string list to your resource file is easy, but not totally intuitive. When you've created an 'STR#' resource, you get an empty list. To add new strings, you must click on the ***** box and then select Insert new field(s) from the Resource menu. Then, just type in the strings or copy from some text file (see figure 4.9)!

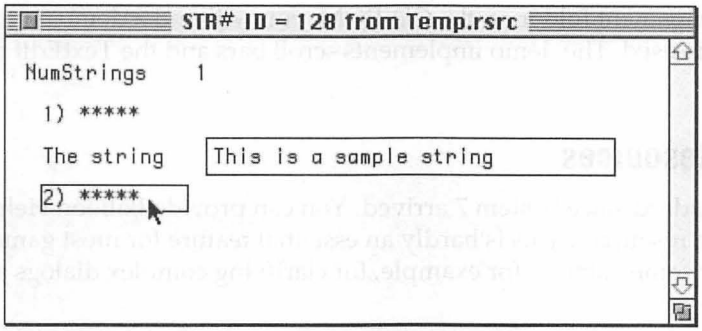


Figure 4.9 *Before adding an item to a 'STR#' resource, you must select the ***** box.*

'STR ' Resources

The 'STR ' resource is one `Str255` string in a resource. It has the same use as an 'STR#' resource, but is somewhat easier to manipulate from within your program. Instead of being a list of strings, 'STR ' stores individual strings as individual resource files. I prefer 'STR#' resources, but that is just a matter of taste. You can pick either of them.

Creating a string is trivial. Just select Create New Resource and pick the 'STR ' type. You get an empty text box in which to type the string.

The Toolbox routine `GetString()` gets an 'STR ' resource to a string handle.

'TEXT'/'styl' Resources

'TEXT' resources are just that—plain text. If they come with a 'styl' resource, the combination of the two is text with formatting—different styles, fonts, and sizes. ResEdit provides editing capabilities for them that come close to the capabilities of a simple word processor.

This resource pair is very useful for making styled text for your help system. By using styled TextEdit, the enhanced version of the MacOS TextEdit package, you can insert the contents of a 'TEXT'/'styl' resource pair into the TextEdit record.

The actual insertion of a 'TEXT'/'styl' in a TextEdit record `textResourceHandle` is done as follows:

```
/*Get the 'TEXT'/'styl' resources and insert them into the TextEdit record.*/
  textResourceHandle = GetResource('TEXT', textResourceID);
  stylResourceHandle = (StScrpHandle)GetResource('styl', textResourceID);
  HLock(textResourceHandle);
  TEStylInsert(*textResourceHandle, SizeResource(textResourceHandle), ➥
stylResourceHandle, styledTextTEHandle);
  HUnlock(textResourceHandle);
  ReleaseResource(textResourceHandle);
  ReleaseResource((Handle)stylResourceHandle);
```

In the 04 - File Management folder on the CD-ROM, you will find a demo named StylDialog in which this code is used. The demo implements scroll bars and the TextEdit management, so it is fairly large.

Balloon Help Resources

Balloon Help is standard since System 7 arrived. You can provide Balloon Help without any code just by adding resources. This is hardly an essential feature for most games, but it can be of some help for some games—for example, for clarifying complex dialogs and menu items.

The easiest way to add Balloon Help, in my opinion, is to use Apple's BalloonWriter. I know not all developers would agree with me, since BalloonWriter is not entirely bug-free and won't solve all Balloon Help problems, but it does the job for most games. It is a quick and painless way to add balloons for your menus and dialogs.

Another alternative is Resorcerer, the ResEdit replacement. It has templates for editing the Balloon Help resources that make it quite workable. It is, in my opinion, not quite as easy as BalloonWriter, but more stable. If you use Resorcerer, try adding 'hmnu' resources for adding Balloon Help to menus, 'hdlg' resources for dialogs, and 'hwin' resources for windows.

Let's look at how to use BalloonWriter to add just a little Balloon Help to Dungeon 2. First of all, BalloonWriter must be properly installed. It comes with an INIT that should be put in the Extensions folder. Then, I launch BalloonWriter and select New from the File menu to create a new document.

Next, I select Create standard balloons from the menu. I get a Standard File dialog in which I should select the application for which I want to create balloons—in this case Dungeon 2. BalloonWriter searches through the resources in Dungeon 2 and creates balloons for the ones it finds. It comes up with the list you see in figure 4.10.

I can now start editing my balloons. For menus, as this figure shows, I get four boxes for each item. The same holds for dialog items. Windows, however, have only one box. In your balloons, fill in the Normal item and any others that apply. If your menu items can be checked, enter the appropriate text.

It is not always easy to identify all items, but I can live with a little trial and error. When I am in doubt, I just fill in all boxes with numbers and see what happens. I can type a 1 in the Normal box, a 2 in the Checked box, and so on, with different numbers for each balloon. When I try the balloons, balloons with numbers in them will show up for menus and dialogs, telling me in what field in my BalloonWriter document I should type the message I want to be displayed there.

When I want to try my balloons, I select Install Balloons. BalloonWriter creates a copy of Dungeon 2 with my balloons installed. When I run it and turn on Balloon Help, I find that my menu balloons work just fine (see figure 4.11).

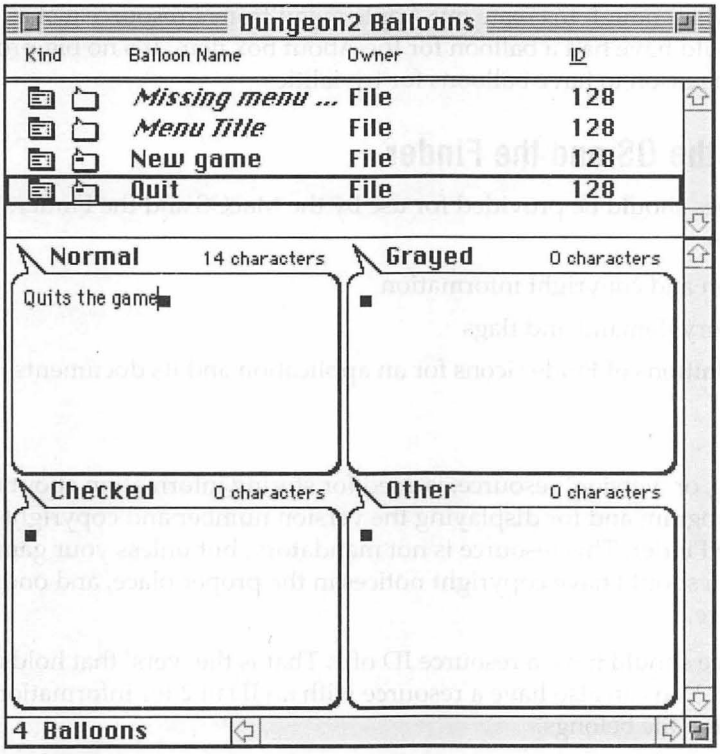


Figure 4.10 *Editing balloons in BalloonWriter.*

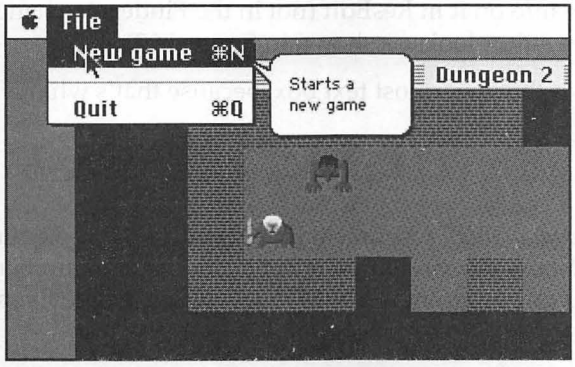


Figure 4.11 *A little balloon help in menus and dialogs—enough for most games.*

Now, had I been wise enough to create my Apple menu from a resource just as I did with the File menu, I could have had a balloon for the About box item. It's no big problem, though. There's no reason to have balloons for trivialities.

Resources for the OS and the Finder

Some resource types should be provided for use by the MacOS and the Finder. They include the following types:

- 'vers'—version and copyright information
- 'SIZE'—memory demand and flags
- 'BNDL'—definitions of Finder icons for an application and its documents

'vers'

The 'vers' resource, or 'version' resource, is used for storing information about the version number of your program and for displaying the version number and copyright notice in the Get Info box in the Finder. This resource is not mandatory, but unless your game is in the public domain, you should have copyright notices in the proper place, and one proper place is the 'vers' resource.

Your 'vers' resource should have a resource ID of 1. That is the 'vers' that holds information about the file itself. You can also have a resource with an ID of 2 for information on the package to which the file belongs.

In figure 4.12, I use TeachText as an example. It shows what I get with Get Info in the Finder.

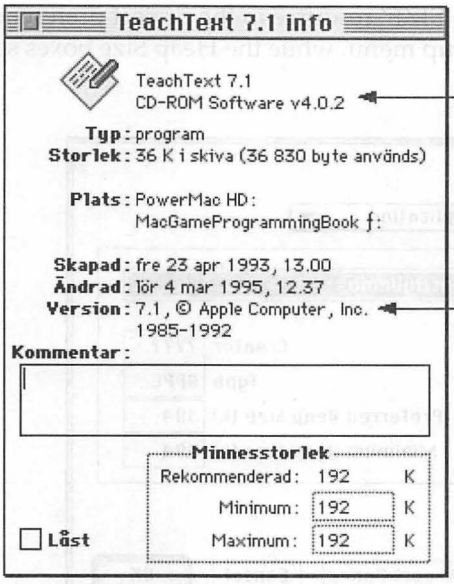
You must add the 'vers' to your resource file manually using ResEdit. After creating the resource, you should Get Info on it in ResEdit (not in the Finder) to change its ID number to 1 or 2. The 'vers' resource editor looks as shown in figure 4.13.

The most important box is the lowermost text box, because that's what is displayed in Get Info.

'SIZE'

Your program should have a 'SIZE' resource with a resource ID of -1 (minus one). The resource includes the memory demand of your application and a number of flags, including information on whether your program is MultiFinder-aware, Apple Event-aware, 32-bit compatible, and more.

Many development systems, including CodeWarrior, create this resource automatically, but if you don't use CodeWarrior, you should see how your development system handles 'SIZE' resources.



Here is a message from the 'vers' resource with ID 2, telling us what package the file belongs to.

Here is the copyright notice from the 'vers' resource with ID 1.

Figure 4.12 *The 'vers' resources in TeachText hold information displayed by Get Info in the Finder.*

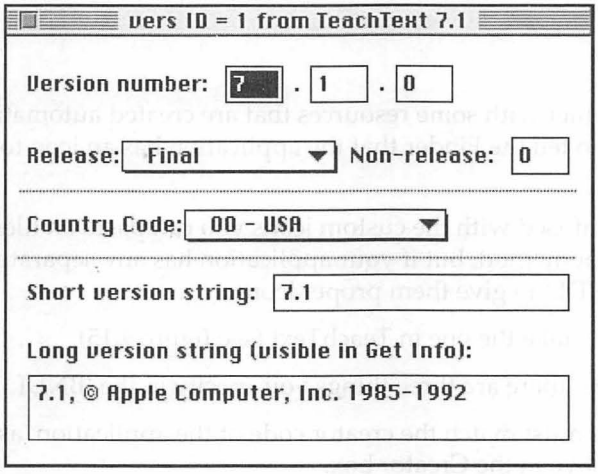


Figure 4.13 *The 'vers' editor, here with 'vers' 1 from TeachText.*

In CodeWarrior, you set the parameters for the 'SIZE' resource in the project preferences dialog. All the flags are in the 'SIZE' Flags pop-up menu, while the Heap Size boxes specify the memory demand (see figure 4.14).

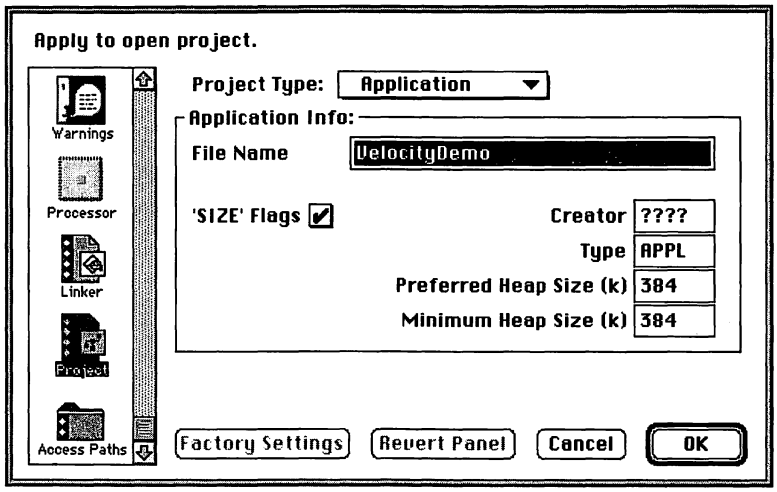


Figure 4.14 *The project preferences in CodeWarrior include all information for the 'SIZE' resource.*

'BNDL'

The 'BNDL' resource, together with some resources that are created automatically when you create a 'BNDL', are used to tell the Finder that the application has an icon to use in the Finder.

This icon should not be confused with the custom icons you can paste on files and folders in the Finder. You can use that method, but if your application has any separate files (e.g., save files), you must have a 'BNDL' to give them proper icons.

As an example of a 'BNDL', take the one in TeachText (see figure 4.15).

As you can see in the figure, there are three things you specify in the 'BNDL' resource:

- The creator code. This must match the creator code of the application, as specified in the project preferences above in the Creator box.
- Document type codes used by your application.
- Icons for each type. What you see in the figure are entire "icon families" for each document type.

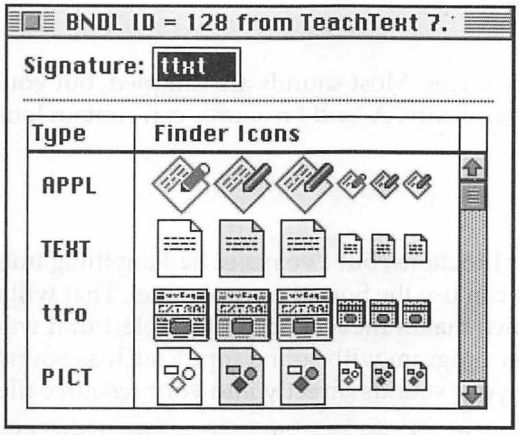


Figure 4.15 *The 'BNDL' of TeachText, as displayed in the 'BNDL' editor of ResEdit.*

Note
After adding or editing a 'BNDL' resource or the icons it uses, you may have to rebuild the desktop to make the new/changed icons show up there.

Menu Resources

The 'MENU' resource is what you should use to define your menus. While it is quite possible to create menus directly from the code, that will make it hard or impossible for users to change the Command-key equivalents. That ability might be more important than you think. I often have to edit the Command-key equivalents myself when programs use keys that are not available on my keyboard.

See chapter 5, "User Interaction," for more information on and demos of menu resources.

Dialogs and Alerts

Dialogs and alerts are defined in resources using 'DLOG' and 'ALRT' resources, respectively. Both use 'DITL' resources, dialog item lists, for describing the contents of the dialogs.

You should use 'DLOG's for complex dialogs with editable text, checkboxes, radio button sets, and user items. If you just want to put up a message or question with static text, pictures, and buttons, use an 'ALRT' instead.

In the next chapter, you'll find more on dialogs, including sample code.

Sound Resources

Sounds are stored as 'snd ' resources. Most sounds are sampled, but you can create 'snd ' resources with much more complexity. A 'snd ' resource can contain looped sound samples and even melodies.

Adding Sound Resources

ResEdit includes sound input functions, but I've never had anything but problems with it. When recording sounds, you can use the Sound control panel. That will make your sounds end up in the System file, which makes me feel uncomfortable, but it works. I would recommend that you use some other program with sound input, such as SoundEdit. From such programs you can often save your sounds directly into your resource file.

You may also want to check out the shareware alternatives available. One shareware sound utility that has been available for a long time is Sound Mover by Riccardo Ettore (see figure 4.16). It not only moves sound resources between files, but also allows some editing and resampling features.

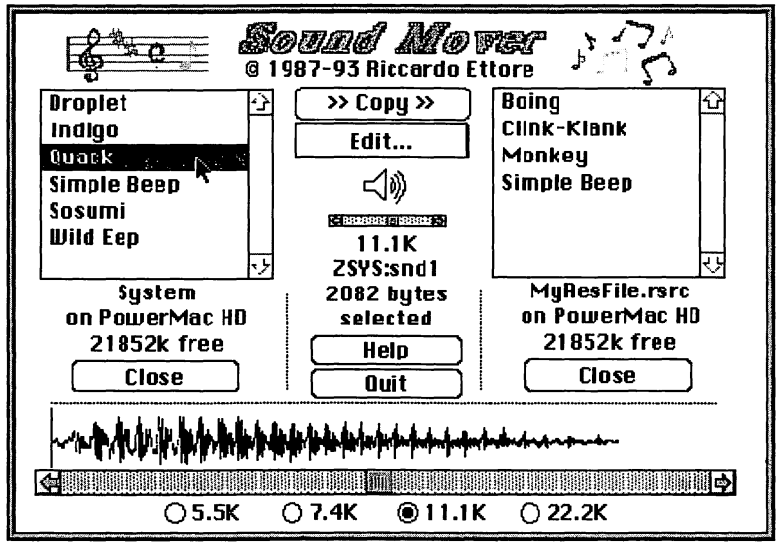


Figure 4.16 *Sound Mover is one of many utilities available for moving and editing sounds.*

Graphics Formats

There are a large number of formats out there for storing graphics. Some important portable ones are GIF, TIFF, and JPEG, and on every computer platform, there are one or more custom formats. On the Mac, we have had two standard file formats for picture documents from the very beginning: PICT and MacPaint.

There is little reason to bother with the portable or foreign formats. All you should do with them is take a converter and convert them to PICT. There is even less reason to bother with the MacPaint format, which is black-and-white. The PICT format is the most useful.

However, all these formats are data fork formats, with one picture for each file. When making a game, you should store all graphics as resources. The standard resource formats for graphics are:

- 'cicn'—color icon (masked)
- 'clut'—color look-up table
- 'crsr'—color cursor
- 'CURS'—black-and-white cursor
- 'icl4'—4-bit color icon (Finder-style)
- 'icl8'—8-bit color icon (Finder-style)
- 'ICN#'—black-and-white icon (Finder-style)
- 'ICON'—black-and-white icon
- 'ics#'—small black-and-white icon (Finder-style)
- 'ics4'—small 4-bit color icon (Finder-style)
- 'ics8'—small 8-bit color icon (Finder-style)
- 'PAT '—black-and-white pattern
- 'PAT#'—black-and-white pattern list
- 'PICT'—picture resource
- 'pltt'—color palette
- 'ppat'—color pattern
- 'ppt#'—color pattern list
- 'SICN'—small black-and-white icon list

'cicn'

The 'cicn' masked color icon format is very flexible, allowing size from 8 x 8 pixels up to 64 x 64 pixels. It includes a color icon, a black-and-white icon, and a mask. The format is quite useful for games. See the section coming up, "Choosing a Graphics Format."

'clut'

A 'clut' is a color look-up table in a resource. You should make one when you need a color table other than the standard one. Many games can use the standard palette and have little need for 'clut's. However, a custom 'clut' can improve the appearance of high-quality graphics. Other games need custom 'clut's to simplify drawing, by using pixel values that can be calculated by some simple formula.

'crsr'

The 'crsr' is the color cursor resource format. If you want to use custom cursors in your game, you should use this format. Using it in your program is quite simple. You load the cursor with `GetCCursor()` and activate it with `SetCCursor()`.

'CURS'

'CURS' is the old, black-and-white cursor format. It is perfectly usable, but does nothing that a 'crsr' won't do for you. I would use 'crsr' even for a black-and-white cursor. Why deny any ResEdit hackers some fun, colorizing your cursors?

'ICON'

The old 'ICON' resource format for black-and-white icons has only one use that I can see: for 'ICON' items in dialogs. If you have an 'ICON' and a 'cicn' with identical IDs and an 'ICON' item in the dialog that refers to the same ID, the Dialog Manager will automatically display the proper one when displaying a dialog with icons.

'icl4', 'icl8', 'ICN#', 'ics#', 'ics4', and 'ics8'

All these resources are part of icon families. An icon family is used by the Finder for displaying the icon of a file. 'icl4' and 'icl8' are 32 x 32-pixel color icons in 4-bit and 8-bit; 'ICN#' is a 32 x 32 black-and-white icon plus mask; 'ics4', 'ics8', and 'ics#' are corresponding small icons, 16 x 16 pixels. These are the Finder icons. When you create a 'BNDL' (bundle) resource, in order to give your game its Finder icon, you can include any or all of the above resources. The 'ICN#' is mandatory and the 'ics8' is highly desirable.

'PAT ' and 'PAT#'

The black-and-white patterns ('PAT ') and pattern lists ('PAT#') are of no interest for modern programs. Use 'ppat' instead.

'PICT'

The 'PICT' resource is a QuickDraw Picture stored in a resource. Thus, the Toolbox utility function `GetPicture()` is identical to a `GetResource()` and a type cast to `PicHandle`. The format is very flexible, because it can do anything that a QuickDraw Picture can. See "Choosing a Graphics Format" coming up.

'pltt'

The 'pltt' is a color palette. If you use the Palette Manager, you may need resources of this type. Just as with 'clut's, you often don't need them.

'ppat' and 'ppt#'

The 'ppat' and 'ppt#' formats describe color patterns. The 'ppat' holds a single pattern (actually one color pattern and one black-and-white pattern), while the 'ppt#' holds a list of patterns. Since they always include a black-and-white version of the patterns, there is no need to include any 'PAT 's or 'PAT#'s (black-and-white–only formats) in your program.

Color patterns are quite useful for games, especially if you want a moderately space-conservative but still decorative way to fill the background with different patterns.

You load a 'ppat' by calling `GetPixPat()`. You set it to the foreground pattern with `PenPixPat()`. You can also set the pattern to be the background pattern of a port using `BackPixPat()`. I find the latter more useful. After setting a pattern to the background pattern, an `EraseRect()` call paints the pattern over the erased area.

'SICN'

The 'SICN' format, describing a list of small black-and-white icons (16 x 16 pixels), is not of any interest for modern game programmers. Use 'cicn' instead.

Adding Graphics Resources

You probably draw your graphics in some drawing program, e.g., Photoshop or Color-It. When you have a graphic that you want to put in a 'PICT' resource, you can either save it to a 'PICT' resource, if that option is provided, or select the entire picture (or a part of it, if you don't want all of it) and copy it to the clipboard (see figures 4.17 and 4.18).

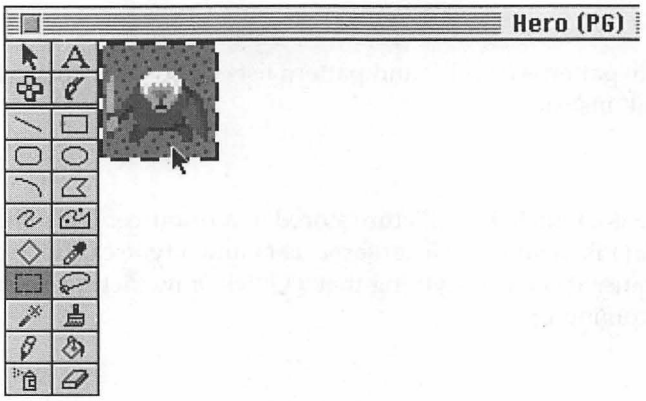


Figure 4.17 *You copy your graphics from a drawing program (in this case, ClarisWorks)...*

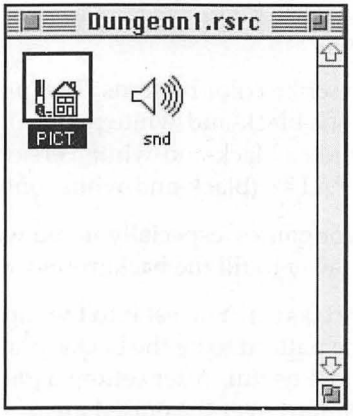


Figure 4.18 *...and paste it into ResEdit, where it will become a 'PICT' resource.*

Photoshop, for instance, can save in 'PICT' resources. You can open that file from ResEdit and copy it with copy and paste. However, I find that procedure more tedious than necessary.

For making 'cicn' resources, you select Create New Resource (Command-K). This fires up a type dialog box. Select the 'cicn' type (see figure 4.19).

You now get an empty 'cicn' resource (see figure 4.20).

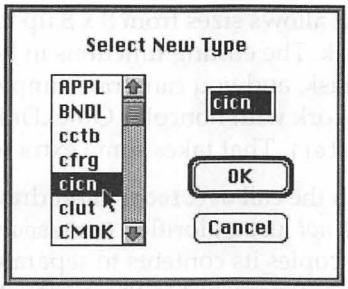


Figure 4.19 *The resource type selection dialog in ResEdit.*

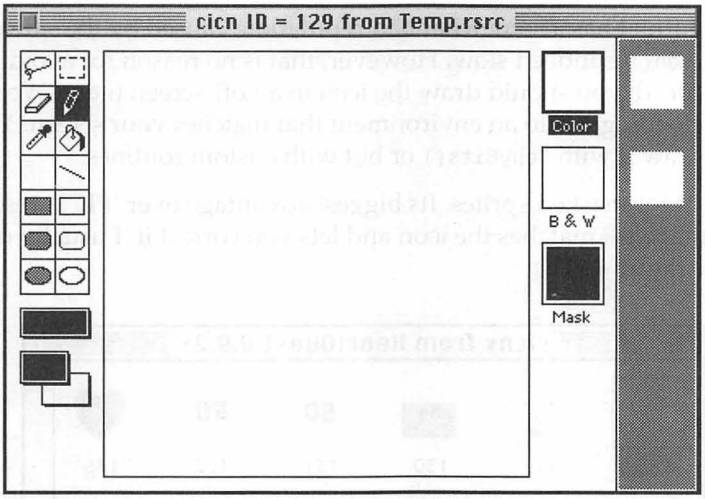


Figure 4.20 *A brand-new 'cicn' resource.*

You can now paste in a picture copied from a drawing program (or just start drawing right here)!

Choosing a Graphics Format

There are only two graphics formats that I find truly worthwhile for game graphics, the 'PICT' and the 'cicn' resources.

The 'cicn' is a very useful format. It allows sizes from 8 x 8 up to 64 x 64 pixels and includes a black-and-white version and mask. The editing functions in ResEdit make it very easy to verify that the icon(s) match the mask, and you can draw simple 'cicn's in ResEdit as needed. You can even make routines that work with noncolor QuickDraw, drawing the black-and-white part of the 'cicn' with `CopyBits()`. That takes some extra work, though.

You load a 'cicn' into memory with the call `GetCIcon()` and draw it with `PlotCIcon()`. You should be aware that `GetCIcon()` is *not* just a glorified `GetResource()` call like `GetPicture()`. `GetCIcon()` loads the resource and copies its contents to separate data structures. The returned handle does *not* refer to the resource, but to these data structures. Note that the resource is not being released, so if you want to save memory space, you must either explicitly release it or make it purgeable.

The 'cicn' resource format has a somewhat bad reputation, caused by the slowness of `PlotCIcon()`. `PlotCIcon()` is indeed slow. However, that is no reason for avoiding the 'cicn' format. To get full speed, you should draw the icon in an off-screen (i.e., GWorld) during startup, thereby preloading it into an environment that matches your screen. You can then, during animation, draw it with `CopyBits()` or blit with custom routines.

The 'cicn' is excellent for masked sprites. Its biggest advantage over 'PICT' resources is how it shows you how the mask matches the icon and lets you correct it. I usually use 'cicn's for my own games (see figure 4.21).

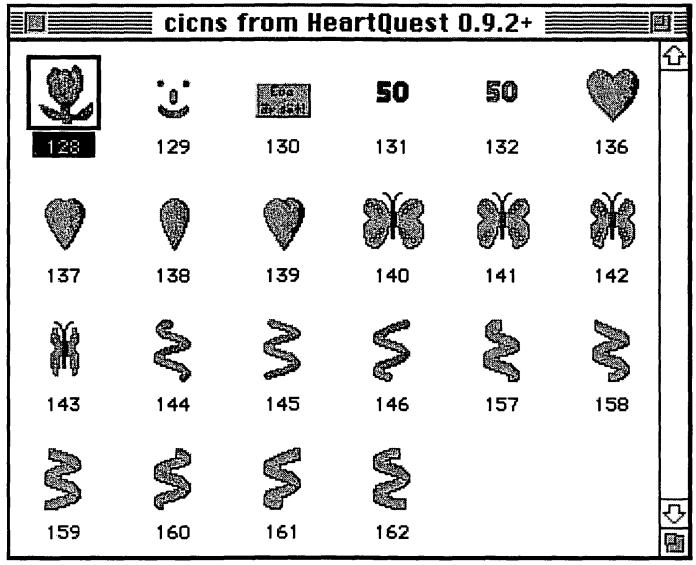


Figure 4.21 *My own game HeartQuest uses 'cicn's for storing sprite graphics.*

For 'cicn's as well as many other icon resource types, ResEdit includes editing capabilities that make it possible to design the icons directly in ResEdit (see figure 4.22). However, for best results, you should draw in a real drawing program. You can still use 'cicn's—just copy an appropriate area from your drawing program and paste in the 'cicn' editor in ResEdit!

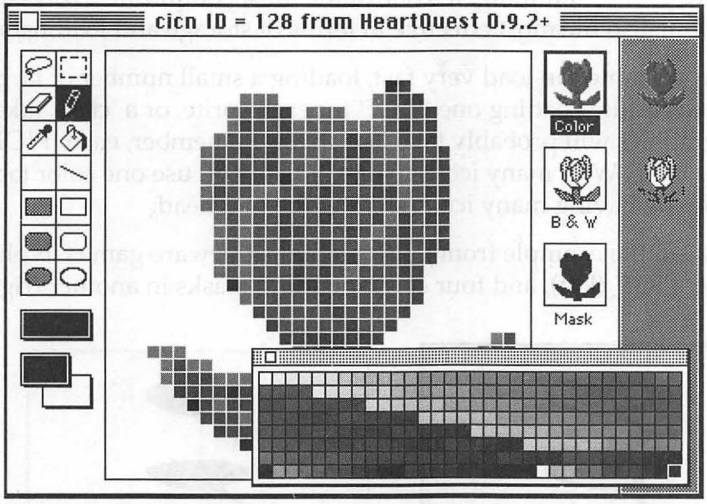


Figure 4.22 *For color icons ('cicn'), ResEdit includes simple drawing features.*

While the 'cicn' has some flexibility in size, the 'PICT' resource format is much more versatile. The Picture is the bread and butter of Mac graphics. It can contain just about anything—pixmaps, polygons, lines, text—and it can be masked by regions. It can even handle pictures compressed by QuickTime if QuickTime is installed. As if that weren't enough, nothing can be easier to use. Any Mac, even the oldest ones, can do `GetPicture()` and `DrawPicture()`, and those calls will work with just about any 'PICT'.

What are the drawbacks, then? 'PICT's cannot be edited in ResEdit, so you must go back to the drawing program for making changes. There are no quick fixes, in other words. Many drawing programs will not allow you to create 'PICT's that are masked, so you might need a better drawing program or provide the mask separately.

Finally, `DrawPicture()` is often far from fast enough, but that is equally true for all other standard formats as well. For top speed, graphics need to be preloaded to off-screen pixmaps (i.e., GWorld) and drawn with `CopyBits()` or custom routines.

In drawing programs, you often see the PICT file format. 'PICT' resources and PICT files are almost the same, but in the PICT file, the picture is stored in the data fork of the file. It starts with a 512 byte header, and after that you will find a QuickDraw Picture.

The 'PICT' is the obvious choice for splash screens, background pictures, and so on. It can be used for holding icons for sprites, but it isn't quite as straightforward as using 'cicn's.

There are two ways to store sprite icons in 'PICT's: either with one icon per 'PICT' or with combined 'PICT's with many icons. The only case in which I prefer to use the one-icon-per-'PICT' method is when I need sprites so large that they won't fit in a 'cicn'.

When putting many icons in one 'PICT', you get some important advantages as well as drawbacks. With regard to programming, it's one of the most complicated ways to store sprites that I know of, but also the most effective in terms of storage and loading speed.

On the positive side, your game can load very fast, loading a small number of large 'PICT's and drawing them in GWorlds. Loading one 'PICT' for each sprite, or a 'cicn', takes much more time. Also, the graphics will probably take less space. Remember, each 'PICT' and 'cicn' has its own color table. With many icons in one 'PICT', you use one color table for all of them! You save space by having many icons sharing the overhead.

In figure 4.23, I show a simple example from john calhoun's freeware game Glypha III. It holds four icons in one 'PICT' (left), and four corresponding masks in another (right).

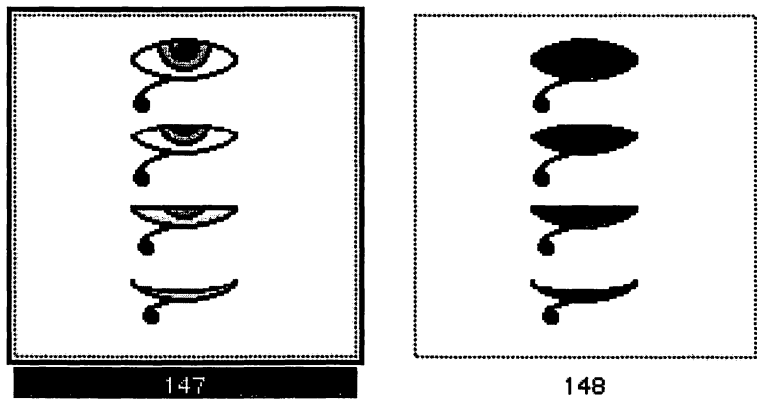


Figure 4.23 *In Glypha III, john uses 'PICT' resources to hold several icons, in this case the blinking eye.*

On the negative side, it's harder to ensure a perfect match between the 'PICT' and the mask than it is for the 'cicn'. You also get the not entirely trivial problem of getting the right rectangle for each part of the 'PICT'. I have seen games that hard-code each and every part as a rectangle. That is tedious and takes lots of time to get right the first time—and even more if you make modifications.

Using 'DITL' to Size Rects

There is, however, one solution that makes this problem a bit easier. In ResEdit you can edit dialogs in the form of 'DLOG'/'DITL' pairs. In a 'DITL', the dialog item list, you can put in 'PICT's, user items, and many other items. The 'PICT' item is a reference to a 'PICT' resource. The user items are just rectangles (see figure 4.24).

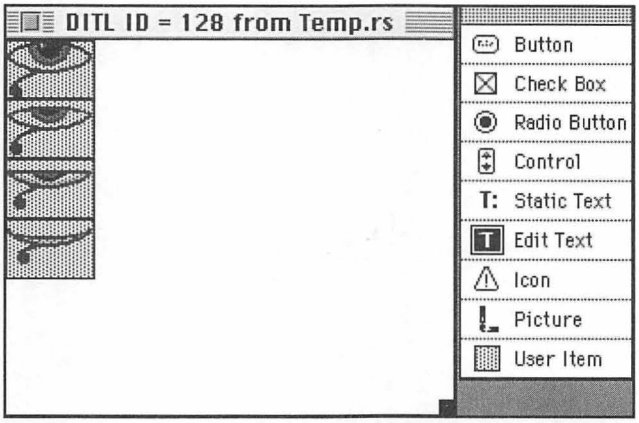


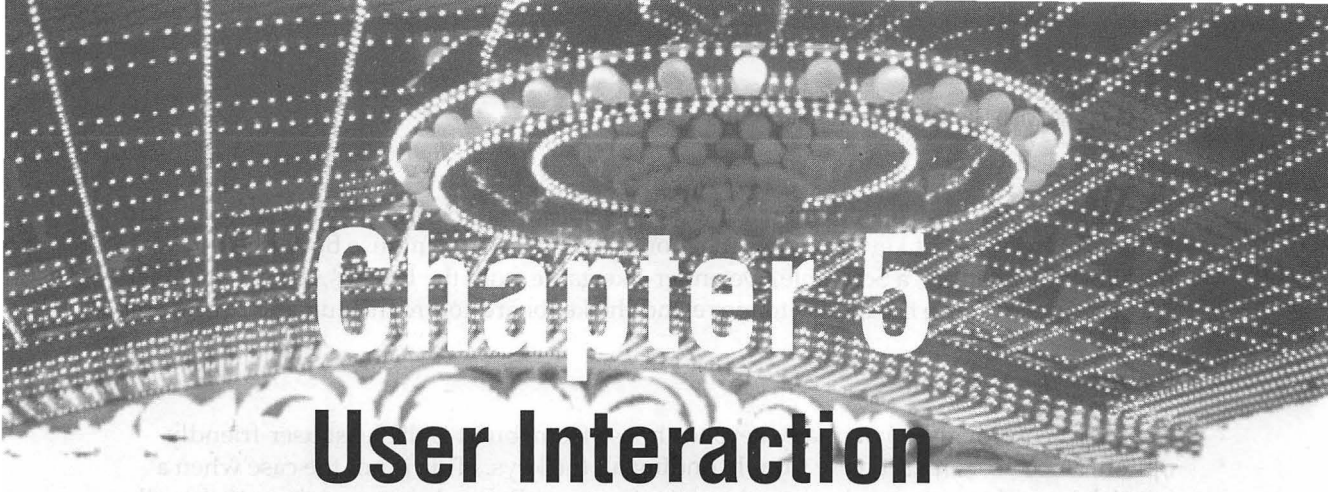
Figure 4.24 *The Glypha eye, as a 'PICT' item in a 'DITL' resource, plus four user items describing the positions of each part.*

This way, you can specify exactly what part of the 'PICT' to use for each icon, doing the fine-tuning in ResEdit instead of changing constants in the program code and recompiling.

If you make the dialog invisible (unchecking the box Initially visible in the 'DLOG' resource editor), you can now get the dialog without disturbing the user and get rectangles from it with `GetDialogItem()`, which will be familiar to you once you've read the next chapter, "User Interaction." If you are a bit more ambitious, you can write your own code for extracting the rectangles. You can find information about the 'DITL' format in *Inside Macintosh*.

Adding and Editing Other Resources

At this point, you should know more than enough to start exploring ResEdit on your own. Open files and applications and have a look at what hides inside them. For some resource types, you get a nice editor, for others a primitive editor, and for some, you just get the hexadecimal editor.



Chapter 5
User Interaction

Games are *interactive* computer programs. Information flows in both directions between the player and the computer.

The player gets information from the graphics and sounds that the computer creates. The graphics must not only be pretty, but also deliver the information they are there for. The same holds true for sounds. Some sounds add to the mood of the game (for example background music) while others are there to inform the player of important events.

The information flowing in the other direction, from the player to the computer, is completely different. The computer gets information from the input devices: the mouse and/or the keyboard, and special game devices such as joysticks or gamepads.

Information to and from the player, output and input, are usually closely coupled. When the computer gets input from the player, it should give appropriate feedback, and the player's options should, as much as possible, be obvious to the player, or at least easy to find. You can save yourself a lot of manual writing and support problems by following Apple's user interface guidelines (as far as game-making allows) and adapting its underlying ideas to your game. Some aspects of Apple's user interface guidelines may not be appropriate to the game you are developing, but in general, if players are comfortable with the user interface, they are less likely to give up on your game.

In this chapter, I discuss how to get input from the user through the mouse or keyboard, and how to make a user interface that is appropriate for your game.

Mouse and Keyboard Input

On the Mac there are only two input devices that you need to bother with: the mouse and the keyboard. There are joysticks and gamepads for the Mac, but they send their signals as keyboard key-downs or mouse movements, so these devices will probably work even if you don't code with them in mind. If possible, you should try your game with a joystick at some point, but it isn't critical.

Should your game use keyboard or mouse controls? That depends on the game, of course, but there is one solution that you should strongly consider: Include support for *both* keyboard and mouse control and let the user choose.

Some games even utilize both mouse and keyboard. An early example is Dark Castle, in which you use a set of keys to move, the mouse to aim, and the mouse button to throw missiles. Sky Shadow, a Scramble/Defender-like game from the late 80s, does it the other way around: You use the mouse to move and the keyboard to fire the gun and drop bombs.

Keyboard Control

Keyboard control gives you many keys to choose from, but it is the least user-friendly option, since users might have trouble finding all the keys. (This is not the case when a joystick is used to simulate keyboard input, of course.) Before I show you how to actually read the keyboard, let me make just a few notes.

Keep the Number of Controls Reasonably Low

If every key on the keyboard has a function in your game, that function had better be intuitive. Chances are high that it isn't. I sure wouldn't play an action game in which I must be able to pick one key out of 40 in a fraction of a second.

If one can play the game two-handed, all keys should be reachable without moving the hands. If a hand must be moved to hit some special key, then that key must be one that is very easy to find—one in the corner such as Backspace (and Escape on some keyboards).

Are the keys arranged in sets with related functions, with positions that reflect their function? If that is the case, a larger number of keys can be used. Players can probably handle up to six keys per hand without picking the wrong key all the time. On the numeric keypad, it is possible to use nine keys, but you can't trust all users to have a numeric keypad. Try to keep the essential keys fewer than that.

If the various keys have no relation to each other, then you must use fewer keys. It's easy to keep track of three keys that trigger left, forward, and right movement, but less so for keys with functions like thrust, fire, and hyperspace. Three keys are plenty in the latter case.

For example, in Spectre you can control the tank with the keypad using eight directional keys. However, you can choose to use only four—forward, backward, left, and right. You also need a key to fire, a key for switching special weapons, and a key to fire the special weapons. That gives you a total of three to four *essential* keys per hand. That is okay with me.

Let the User Configure the Keys!

This is important! Opinions on layouts differ. Some people will want to play your game one-handed, and others will do it two-handed. (Not to mention three-handed or four-handed. What? Am I talking about alien creatures from outer space? No, but there can be two kids who want to codrive the game.)

However, there is an even more important reason to provide configurable keys. Occasionally you see games that are hard-coded for the numeric keypad or hard-coded for the U.S. keyboard (see the section about `GetKeys()`). Those games might not be playable *at all* for some users! If they are hard-coded for the main keyboard with the U.S. layout, it might be incredibly frustrating having to hit + (plus) when the manual says - (minus). Also, some keyboards lack certain keys. For example, the older standard keyboards have no function keys and no Page Up or Page Down keys. PowerBook keyboards have only the main keyboard, with no numeric keyboard.

Now, imagine a European user with a PowerBook! Oops.

Note
Don't hard-code the keys!

Reading the Keyboard

Having decided what keys to use, you have to get at them. There are two ways to read the keyboard: using the `GetKeys()` call or getting key-down events.

Normal programs get all key inputs through events using `WaitNextEvent()`. If speed is not critical in your game (i.e., if you are not using animations requiring 15-60 frames per second), this is what you should do too. It is easy to do and makes your program compatible with other programs running in the background.

`GetKeys()`, on the other hand, is the fast way to read the keyboard. The call delivers a `KeyMap` structure which is 128 bits or four longwords. Each bit represents a key on the keyboard; the bit is 1 if that key is pressed. Note that the keyboard can't read more than a few keys at once—two is a common limit—so you can't ask the user to hold down more than it can read. Modifier keys—Shift, Command, Option, and Control—do not fall under this limitation.

The `KeyMap` is a packed array of 128 Booleans, 1 bit for each, which means that it is stored in 128/8 = 16 consecutive bytes. C cannot represent a packed array of Booleans, so in C the `KeyMap` is an array of four longs. The following snippet solves this problem, returning a Boolean if the key with the key code passed to it is being pressed. The trick is to use shifts and binary AND operations to check 1 bit in one of the four longs.

```
static Boolean CheckKey(short keyCode)
{
   KeyMap  myKeyMap;
   short   byteIndex;
   char theByte, theBit;
   char *thePointer;
```

```
  GetKeys(myKeyMap);                                  // Get the KeyMap
  byteIndex = keyCode >> 3;                           // What byte in the KeyMap?
  thePointer = (char *)&myKeyMap[0];                  // Make pointer
  theByte = *(char *)(thePointer + byteIndex);        // Point to the byte
  theBit = 1L<<(keyCode & 7);                         // What bit in the byte?
  return ( (theByte & theBit) != 0 );                 // Check that bit
} /*CheckKey()*/
```

Even though `GetKeys()` is a lot faster than reading events, I would not use `CheckKey()` as it is if I needed to test many keys for every frame. Rather, I would rewrite `CheckKey()` to read the `KeyMap` *once*, and then check for all the keys I need using the same `KeyMap`.

The problem with `GetKeys()` and `KeyMaps` is that you don't know for sure what key each bit represents. You can easily figure out what key it is on *your* keyboard, but keyboard layouts vary a lot. Far too many game makers assume that all users have the U.S. layout with the U.S. key code, but the key codes are *not* bound to specific characters! Believe me, your game gives an unprofessional impression when it asks the player to press keys that don't exist or when the player must use other keys than the ones mentioned in the documentation.

Note

Don't assume that key codes necessarily represent the same characters they do on your own keyboard.

What you must do if you use `GetKeys()` is write your key configuration dialog so that it records both the character being typed and the key number. To do this, you should let the user type the key and catch the generated `keyDown` event. You will then find both the character and the key code for use with `GetKeys()`.

The following code snippet gets an event, and if it is a `keyDown` event, it gets both the character and the key number, which you can use as index to a `KeyMap`.

```
if (WaitNextEvent(eventMask, &theEvent, sleepTicks, mouseRgn)) then
 switch (theEvent.what)
…
  case keyDown:
    theKeyNumber = (theEvent.message & keyCodeMask) >> 8;
    theChar := (char) (theEvent.message & charCodeMask));
    break; /* keyDown */
…
```

I will return to this topic later in this chapter, in the demo KeyConfig, in which I will show you how to make a configuration dialog complete with key configuration, radio buttons, and checkboxes.

Mouse Control

While you must be careful not to make keyboard control too complex, mouse control is always easy to use—especially considering that we have one-button mice on the Mac. That is usually a blessing.

The mouse can be used for games in several ways.

Absolute Positioning

The most straightforward way to use the mouse is as I use it in Dungeon: to point and click some place in the game window, allowing the mouse cursor to indicate the position to be used. This method is appropriate for strategy games and puzzle games—and just about any kind of mouse-using game except action games.

Relative Positioning

In action games, you will often find mouse control that gives relative positioning, in which you don't control the position directly, but rather the speed in some way. Some examples include Crystal Quest, Sky Shadow, HeartQuest, and Ingemar's Skiing Game. I will return to this method later in this chapter and describe how to implement this kind of mouse control.

Modifier Keys

Sometimes, especially when using absolute positioning, you may want to be able to make modified clicks—some kind of click other than a "plain" mouse click. There are two ways to do that, the right way and the wrong way.

The *right* way is to hold down a keyboard key and then click the mouse in some position. The standard modifier keys, which you should use if possible, are Shift, Command, Alt (Option), and Control. You can use other keys as well, like SimCity does.

The *wrong* way, which you may see sometimes, is to hold the mouse cursor over the space you want to do something with, and then press a keyboard key without clicking the mouse button. This is not the Mac standard—avoid it! The mouse button is the "mouse action" key. Keyboard keys should *only* serve as modifiers.

Reading the Mouse

The mouse, like the keyboard, can be accessed either through quick routines or through events. However, the mouse can almost always be read through the quick routines with no drawbacks.

You get the current mouse position with `GetMouse()`, which gives you a `Point` structure. You read the mouse button with the `Button()` function, which returns a Boolean that is TRUE if the button is down. It can't be much easier.

The only hard part in reading the mouse is that you must not be careless in setting the port. If you are not careful, you could get the mouse position relative to the wrong port.

In Dungeon 1, I use `GetMouse()` and `Button()` to read the mouse. Let's have a look at the main loop of the game again to see how that works! It is really easy:

```
/*Initializations done! Run the game loop until the game ends.*/
  do
  {
    if (Button()) {
    GetMouse(&clickPoint);                  /* Get the position of the click */
    clickedTile.h = clickPoint.h / kTileSizeH;      /* Convert to grid */
    clickedTile.v = clickPoint.v / kTileSizeV;
    MovePlayer(clickedTile);                /* Try to move there */
    do {} while (Button());                 /* Wait until the mouse click ends */
    };
  } while (!  gDone);
```

This is a typical way to handle mouse reading for a program that doesn't process events. I poll `Button()` to see if the mouse is pressed. I get the mouse position to a `Point` structure named `clickPoint` using `GetMouse()`. Then I perform the action for the click, moving the player. Finally, I poll `Button()` again, waiting for the mouse click to end so I don't process one click as two.

Nothing more to it—until we start processing events.

Setting the Mouse Position

The mouse may sometimes appear limited and not very suitable to games compared to a joystick. This is mainly a question of what you do with the mouse. The game Crystal Quest demonstrated how good the mouse can be for games when used wisely. The mouse can be rather clumsy for absolute positioning, but it is great for relative movements.

When making mouse control as in Crystal Quest, we can set the speed of whatever we are controlling—let's say a spaceship—to be proportional to the distance between the mouse position and the center of the screen.

When we start, we probably want the ship to be still. This is fixed by setting the mouse position to the center of the screen.

But hey, wait a minute! Apple says that we can't move the mouse cursor! That is forbidden according to User Interface laws!

Well, Apple also tells us not to draw directly to the screen, and that we must process events all the time and allow time to other applications. Apple has good reasons for its rules, but when making games there are some rules we have to break—but only after thinking twice about it.

Moving the mouse cursor is a strict no-no as long as the cursor is visible, since it will make users feel as if the computer is taking control over them. It feels as if the Mac grabs your finger and drags it to some place where the computer wants you to go. *Never* do that!

However, consider Crystal Quest. The cursor is hidden! The game gives a good response to the mouse movements in the ship speed. There is no problem here. Crystal Quest breaks the User Interface rule wording, but not the spirit. The user is in control, so it's okay.

Right. So you can move the cursor as long as you let the user retain the feeling of control. Here's the code to do it:

```
/* SetMouse(), sets the mouse position */

static void SetMouse(Point newMousePosition)
{
   CursorDevicePtr   theCursorDevice = nil;
   OSErr        errorCode;
```

I call `NGetTrapAddress()` to learn if the trap `AADB`, which is the Cursor Device Manager trap, is available.

```
   if (NGetTrapAddress(0xAADB, ToolTrap) !=
      NGetTrapAddress(_Unimplemented, ToolTrap)) /* Is Cursor Device Mgr available? */
   {
```

If `AADB` is available, then I call `CursorDeviceNextDevice()` to find a device. We probably just want the first device (there may be a digitizing tablet connected as well), but I look for a one-button device.

```
/* Find a device with one button. That ought to be the standard mouse! */
   do {
      errorCode = CursorDeviceNextDevice(&theCursorDevice);
      } while ((theCursorDevice != nil) &&
         (*theCursorDevice).cntButtons != 1);
   }
```

If I did find a mouse, I set its position with `CursorDeviceMoveTo()`.

```
   if (nil != theCursorDevice)
      {
      LocalToGlobal(&newMousePosition);
      errorCode = CursorDeviceMoveTo(theCursorDevice, newMousePosition.h, ➥
newMousePosition.v);
      }
```

If I failed to find a mouse, or if the Cursor Device Manager were not available, I set the mouse position by writing into some low-memory globals (82C, 828, and 8CE in hexadecimal). This is not the recommended way, but is needed for many old Macs. You should use the Cursor Device Manager if possible, because it is more likely to stay the same in the future.

```
  else /* If Cursor Device Mgr is not around, do it the old way. */
     {
     LocalToGlobal(&newMousePosition);
     *(Point *)0x82c = newMousePosition;
     *(Point *)0x828 = newMousePosition;
     *(short *)0x8ce = 0xffff;
     }
} /* SetMouse() */
```

Mouse-Controlled Velocity—VelocityDemo

Previously, I mentioned controlling the velocity of an object through the mouse position, as in Crystal Quest. Perhaps you would like to try a mouse control similar to Crystal Quest? It seems simple enough: Just read the mouse position and set the speed of the object from that. If you do it in the most straightforward way, using integers everywhere, you will run into a problem: The movement is too fast, or insensitive.

This particular problem was very apparent in the game Crystal Raider, which was a forerunner to Crystal Quest. The problem is that the speed vectors, represented by integers, are too insensitive. The difference between moving one pixel per frame and no pixels per frame is too big. You need some decimals!

If you use *fixed point* numbers for the position, you can make it work really well. This is what I do in the demo VelocityDemo coming up.

I keep two `Points` as the position. One is the fixed point position, and the other is the position in integers. Both are really integers, but the integer position is found by shifting the fixed point version. I will describe this technique in more detail in chapter 7, "Game Mechanics: Environments and Entities."

This method gives a control that is nice, intuitive, and flexible when you need to tweak it a little.

The demo program VelocityDemo demonstrates how this is done.

```
/* VelocityDemo */

/* Demo program by Ingemar Ragnemalm 1995 */
/* This program demonstrates a way to get mouse control like what you find in ➥
Crystal Quest™, */
/* in which the mouse position controls the speed rather than the position of an ➥
object. */
```

```
/* We use fixed point numbers, by shifting integers four binary digits. That makes */
/* the movement nice and smooth. */
```

`CursorDevices` is included because it is needed for `SetMouse()`.

```
#include <CursorDevices.h>
```

A window pointer is also needed...

```
    WindowPtr myWindow;           /* Our window*/
```

Here follows all the variables for the animation, the object position, screen center, a `Point` to get the mouse position into, speed, and a rectangle for temporary use.

```
    Point integerPosition, position; /* The position of our object, in integer and ➥
fixed point*/
    Point center;                 /* The center of the screen*/
    Point theMouse;               /* A Point that we get the mouse position in*/
    Rect drawingRectangle;        /* A rectangle used for drawing*/
    Point speed;                  /* The fixed point speed*/
```

The size of the animated object is defined by constants.

```
#define kHeight 16
#define kWidth  16
```

The constant `kFixedPointDigits` is the number of "binary decimals." That term is, of course, an oxymoron when talking about the binary system rather than the decimal system. See chapter 3 for more details on fixed point numbers.

```
#define kFixedPointDigits 4
```

`SetMouse()` is the same as the previous snippet. Since it is explained already, I've skipped it here. `InitToolbox()`, the standard inits, are also skipped—you have seen them before.

All that is left, then, is the main program.

```
/* VelocityDemo main program */

void main(void)
{
  long loopStartTime;

  InitToolbox();

/* Set up the window */
  myWindow = NewCWindow(nil, &qd.screenBits.bounds, "\pMemoryGame", true, 0, ➥
(WindowPtr)-1, false, 0);
  SetPort(myWindow);
```

To make it look a bit less cheap, I use a black background and a red foreground. The calls here are the old-style eight-color calls. They are not interesting for general color usage, but when you want just a bright standard color, they are pretty handy.

```
/* It will look a little bit fancier on a black background */
  ForeColor(redColor);
  BackColor(blackColor);
  EraseRect(&myWindow->portRect);
```

I hide the cursor. The feedback comes from the animated object, so the cursor would just be distracting.

```
/* Hide the cursor -- it won't make sense anyway */
  HideCursor();
```

I calculate the center of the screen from the `screenBits` structure. A note on the QuickDraw global `screenBits`: `screenBits` is a bitmap. Its bounds are still useful, but you should never use other parts of the structure (base address or bytes per row) when running on a color-capable Mac.

A good alternative to `screenBits`' bounds is `GetMainDevice()`, which gives a `GDHandle`, in which you can use the `gdRect` field. That is the bounds of the main screen. You can also loop though the list of devices in order to find the selected screen if you allow the game to be played on screens other than the main screen.

```
/* Calculate the center */
  centerOfScreen.h = (qd.screenBits.bounds.right - qd.screenBits.bounds.left) / 2;
  centerOfScreen.v = (qd.screenBits.bounds.bottom - qd.screenBits.bounds.top) / 2;
```

Once I have the center, I set the mouse to the center and initialize the position to the center as well—or any other position I'd like.

```
/* Let the cursor start in the center */
  SetMouse(centerOfScreen);

/* Initialize the position */
  integerPosition = centerOfScreen;
  position.h = (integerPosition.h << kFixedPointDigits);    /* Make fixed point by ➥
shifting left*/
  position.v = (integerPosition.v << kFixedPointDigits);    /* Make fixed point by ➥
shifting left*/

/* Initialize the rect drawingRectangle, too. */
  SetRect(&drawingRectangle, integerPosition.h, integerPosition.v, ➥
integerPosition.h + kWidth, integerPosition.v + kHeight);
```

Initializations are finished! Now it is time for the main loop, which is repeated until the mouse is pressed.

```
do
   {
```

I store the time when the loop starts. This is used in the end to make sure that enough time has passed since the loop started to get controlled animation speed. Otherwise, the game would run too fast on newer machines.

```
/* Get the current time */
     loopStartTime = TickCount();
```

In the loop, a point named `theMouse` is set to the current mouse position. The speed is set to the difference between `theMouse` and the center.

```
/* Get the current position */
     GetMouse(&theMouse);
/* Set the speed to the distance between the mouse position and the center */
     speed.h = theMouse.h - centerOfScreen.h;
     speed.v = theMouse.v - centerOfScreen.v;
```

The speed is added to the point named `position`. Note that this really implies that speed is also in fixed point.

```
/* Move the fixed point position by that much */
     position.h = position.h + speed.h;
     position.v = position.v + speed.v;
```

The object must stay on the screen, so some border checks are made and position is changed as needed. The speed is also set to zero when the object hits the border. That just feels right.

```
/* Stay on the screen! */
     if ( position.h < 0 )
     {
       position.h = 0;        /* Keep on screen*/
       speed.h = 0;           /* Zero speed*/
     };
     if ( position.v < 0 )
     {
       position.v = 0;        /* Keep on screen*/
       speed.v = 0;           /* Zero speed*/
     };
     if ( (position.h >> kFixedPointDigits) > qd.screenBits.bounds.right - ➥
qd.screenBits.bounds.left - kWidth )
     {
       position.h = (qd.screenBits.bounds.right - qd.screenBits.bounds.left - ➥
kWidth << kFixedPointDigits);
```

```
        speed.h = 0;        /* Zero speed*/
     };
     if ( (position.v >> kFixedPointDigits) > qd.screenBits.bounds.bottom - ➥
qd.screenBits.bounds.top - kHeight )
     {
        position.v = (qd.screenBits.bounds.bottom - qd.screenBits.bounds.top - ➥
kHeight << kFixedPointDigits);
        speed.v = 0;        /* Zero speed*/
     };
```

Erase the previous rectangle before drawing the new one using the rectangle `drawingRectangle`.

```
/* Erase from the previous position*/
     EraseRect(&drawingRectangle);
```

The fixed point numbers in position are converted to integers by shifting them right by `kFixedPointDigits` binary positions.

```
/* Divide the fixed point position by 16 -- which is the same as shifting it four ➥
steps (kFixedPointDigits)!*/
     integerPosition.h = (position.h >> kFixedPointDigits);
     integerPosition.v = (position.v >> kFixedPointDigits);
```

The object is drawn in the new position using the rectangle `drawingRectangle`.

```
/* Draw!*/
     SetRect(&drawingRectangle, integerPosition.h, integerPosition.v, ➥
integerPosition.h + kWidth, integerPosition.v + kHeight);
     PaintRect(&drawingRectangle);
```

Finally, the animation must be delayed a little bit, and that should be done in some way that is not dependent on the speed of the Mac it is used on. I use the `TickCount()` call here and wait until `TickCount()` is at least one more than it was when the loop started—which is 1/60th of a second.

```
/* Limit speed -- not more than 60 fps! */
     do {} while (TickCount() < loopStartTime + 1);
  } while (!  Button ());
  ShowCursor ();
} /* VelocityDemo*/
```

There is yet another problem worth considering when using mouse control. The OS will not let the mouse move outside the screen borders. That puts a limit on the speed (depending on screen size), which means that games may behave differently depending on the size of the screen—not a particularly nice limitation.

This can be fixed by setting the mouse cursor to the center of the screen and collecting the difference from the center. The difference is collected to a vector that can be used as position or speed, depending on whether you want the player to perceive absolute or relative movements. Try the following sequence:

- Get the mouse position.
- Set the mouse position to the center.
- Take the difference between the center and the acquired mouse position and add that to an accumulated vector.

I have had good results using this technique with the old-style `SetMouse()` (writing to globals), but the Cursor Device Manager seems too slow. You may have to use the old-style way if you need that kind of speed.

Another solution that makes your game fair for all users (which I used in Ingemar's Skiing Game) is to limit the mouse movements to the smallest screen that the game supports. If `GetMouse()` returns a point outside the allowable area, you just move it inside the bounds.

User Interface

Now let's have a look at the user interaction from a broader perspective, the overall user interface design.

You may argue that games don't have to bother with user interfaces. They often fill the whole screen with what they do, hiding everything that is related to the MacOS. That is *not* a valid excuse for having a lousy user interface!

Games do have user interface rules that are very different from ordinary applications. Since games come in so many varieties, many guidelines apply only to certain games. For example, some games use windows and menus just like ordinary applications. For such games, most of Apple's guidelines apply. For other games, such as action games that fill the whole screen and need the maximum performance possible, many rules do not apply—but some still do.

My first advice, which is more a matter of staying compatible and not limiting yourself, is to use real windows everywhere you draw, whether you draw with QuickDraw or not.

If you don't use real windows, you make it hard to fix a decent user interface once you decide that you really need it, and you might run into compatibility problems.

The next thing you need to know to be able to make an appropriate user interface is *event processing*. Most games need event processing of some kind. For many games it is absolutely mandatory. I would even claim that if a Mac game doesn't use the `WaitNextEvent()` call anywhere, it is not a well-written game.

There are plenty of arguments about this, and if you are an experienced game writer, you probably have some objections already. Don't worry, I will return to this issue later, once I have dealt with events.

Events

Normal Macintosh applications are almost completely event-driven. Almost everything an application does is triggered by some kind of event. There are many kinds of events, but let's have a look at the most important ones.

Mouse Events

The `mouseDown` event is one of the most obvious and important events. The `mouseUp` event is much less useful and is often ignored. You should consider using it when clicking and dragging pieces in board games, among others. For processing a `mouseDown` event, there are some standard actions you should always take in order to handle menu selection, window dragging (if applicable), and so on. Your game may or may not handle `mouseDown` events in the game window(s)—that depends on the game.

Keyboard Events

Some games can use `keyDown` events for keyboard control. If you need to know how long a key is pressed, you can use the `keyUp` event as well.

Using keyboard events is often better than `GetKeys()`. With `GetKeys()`, you may miss short key clicks if a key is pressed and released between two calls to `GetKeys()`.

Update Events

What did you say? Your game has only one window, always in front, so it never needs updating? Wrong. Have you ever had a system alert pop up, for example, saying that a fileserver has shut down? Sure you have. Do you really intend to leave a part of the screen white, with no easy way to have it updated? If you don't, you need to support update events.

Ignoring update events or handling them improperly is a very common beginner's error. Make sure that you handle update events correctly. In most games, this is very simple: When you get an update event, you call `BeginUpdate()`, make a full redraw of the entire window, and end by calling `EndUpdate()`. For a game using off-screens, the full update is just a `CopyBits()` from the off-screen to the screen.

Disk Events

Mac games should also process disk events and handle insertions of bad disks. This may seem like an overambitious thing to do, and you won't get too many complaints if you miss it, but some users will expect you to deal with it.

If you don't handle disk events and a bad or unformatted disk is inserted, the user will never get any notification about it, and the disk will not show up on the desktop. It just sits there in the drive, "dead." This is known as the "dead drive" problem.

Handling a disk event takes just a few lines of code, and it will save some users from the "dead drive" problem. You can copy what I do in the code examples and you will be safe. See the `MainLoop()` code snippet coming up.

Suspend/Resume Events

You may wish to take some action when the application gets switched in or out. For example, if the player can switch out of the game while it is running, it makes sense to pause the game. Handle a suspend/resume event like this:

```
case osEvt:
   if ( ((theEvent.message >> 24) & 0xFF) == suspendResumeMessage )
   {
      isInForeground = (event.message & resumeFlag) != 0;
      DoSuspendResume(isInForeground);
   };
   break;
```

What I'm doing is inspecting certain parts of the message. The high byte tells whether it is a suspend/resume message, and the low byte defines whether it is suspend or resume. But why should you bother with those details? Once your `DoSuspendResume()` routine is being called, the messaging system isn't interesting.

Apple Events

Many games have little interest in Apple events, but these events are quite useful for some games. For networked games (not action games, but board games, strategy games, and so on), Apple events provide a safe and fairly easy way to pass messages between users. Other games that should support Apple events are games in which the user may open a save file in the Finder by double-clicking on it or dragging it to the game application.

Event Processing

An event-using application (i.e., most Macintosh applications) usually has an *event loop* in which it gets events and processes them, calling its menu selection handlers, key-down handler, mouse-down handler, update event handler, and so on.

First of all, we get the event. This is usually done with `WaitNextEvent()`. Older programs generally use `GetNextEvent()`, but you have little interest in that as long as you don't want to be backwards-compatible to a downright paranoid level. From System 6 and up, `WaitNextEvent()` will always work.

The problem with `WaitNextEvent()` is that you can't know how much time it will take. `WaitNextEvent()` allows background processes to run, which may make games that call `WaitNextEvent()` slow if the user has other programs running in the background.

Many action game designers choose not to call `WaitNextEvent()` at all during times when animations are running (i.e., when a game is in progress). This is a safe way to get as much CPU power as possible. Other authors call `WaitNextEvent()` with small sleep values (i.e., 0 or 1). That works well as long as no or few other processes are running, but it can make the game very jerky at times. Personally, I prefer running without `WaitNextEvent()` as default, possibly providing an option to allow background processing.

When I advise against calling `WaitNextEvent()` for action games, that does not mean that you can't use events at all in them. When you need the maximum animation speed possible, you must avoid the calls that allow background processing. `WaitNextEvent()`, as well as its predecessor, `GetNextEvent()`, do that. However, if you want to get key input through events, for example, you can use `GetOSEvent()`, which is a lot faster since it will only get events within your application and not allow background processing.

The event is delivered in an `EventRecord` structure. When we have an event, we inspect the `what` field to see what the event is all about, and then proceed depending on that.

Following is the code for a simple event loop. The snippet calls a number of application-specific routines that deal with the interesting events. All such routines have names starting with `Do` (i.e., `DoMouse()`, `DoKey()`, `DoUpdate()`, `DoMenuCommand()`, and `DoBackground()`).

```
/* Main event loop */

static void MainLoop(void)
{
#define kSleep 5
```

The sleep value is a constant here. That is usually okay, because most games can't background, so they won't take any processing time when put in the background. If you handle suspend/resume events (see above), you can use a variable for the sleep value and change it depending on whether or not the program is in the front. When the program's in the front, the value should be small (5 is high in such a case), but when in the background, it should be higher (10 to 60 is fine, depending on what kind of background processing you do).

```
    EventRecord theEvent;
    char theKey;
    long whatSelection;
    short whichPart;
    WindowPtr whichWindow;
```

I get events with `WaitNextEvent()`.

```
/*Get the next event with WaitNextEvent().*/
  if ( WaitNextEvent(everyEvent, &theEvent, kSleep, nil) )
  switch ( theEvent.what )
  {
```

It is rather tedious to process a `mouseDown` event. `FindWindow()` tells where the mouse is when the button goes down, and we need a new case switch on the result.

```
    case mouseDown:
/*We must find out what kind of mouse-down this was.*/
      whichPart = FindWindow(theEvent.where, &whichWindow);
      switch ( whichPart )
      {
```

If the mouse is in the menu bar, I call `MenuSelect()` and pass the result to my menu handler routine, `DoMenuSelection()`.

```
        case inMenuBar:
/*Click in menu bar. Let the system call MenuSelect() track it.*/
          whatSelection = MenuSelect(theEvent.where);
          DoMenuSelection(whatSelection); /*Our own routine for handling menu ➥
selections*/
          break;
```

If the mouse is in a "system window," I just call `SystemClick()`. The click is none of my business.

```
        case inSysWindow:
/*Click in some window that isn't ours*/
          SystemClick(&theEvent, whichWindow);
          break;
```

If the click is in the close box of the window, I call `TrackGoAway()` to track it, and if that returns TRUE, I should close the window. "DA-style" games typically quit. Many games have no close boxes at all.

```
        case inGoAway:
/*Click in close box of window. For "desk accessory"-style games, that is a good ➥
quit signal*/
          if ( (TrackGoAway(whichWindow, theEvent.where)) )
            gDone = true;
          break;
```

If the click is in the drag bar, there are some standard actions to take. Clicking in an inactive window means selecting it, bringing it to the front. Command-clicking in the drag bar of an inactive window means dragging it.

Even if your game has nonmoveable windows, including window dragging code won't hurt. The nonmoveable windows should have no drag region anyway.

```
          case inDrag:
/*Drag a window*/
            if ( (whichWindow != FrontWindow ()) && ((theEvent.modifiers & cmdKey) ➥
== 0) )
               SelectWindow(whichWindow);
            DragWindow(whichWindow, theEvent.where, &qd.screenBits.bounds);
            break;
```

If the click is in the grow box, the window should be resized. This is even more uncommon than dragging or closing, so I skipped it here.

```
          case inGrow:
            ;  /*Ignored -- we don't resize*/
            break;
```

Finally, there's the click in the content region of a window! This is the interesting one. If the click is to the front window, I pass the click to my mouse click handler, `DoMouse()`. If the click is to a nonactive window, the window is brought to front.

```
          case inContent:
/*Click in the window.*/
            if ( (whichWindow != FrontWindow ()) )
               SelectWindow(whichWindow);
            else
               DoMouse(theEvent.where, theEvent.modifiers); /*Go to application- ➥
specific mouse-down handling*/
        }; /*case whichPart*/
        break; /*mouseDown*/
```

So that was the `mouseDown` event. Complicated? Don't worry, you won't spend too much time digging in the event loop, and it won't get worse hereafter.

The next event to deal with is `keyDown` and its cousin, `autoKey`. The first is the event you get when a key is pressed, and the second is the key repeat you get when you hold a key down. A `keyDown` can mean one of two things: just a key being pressed, or a Command-key combination. In the former case I call my key-down handler `DoKey()`, and in the latter case I treat it as a menu selection, so it goes through the `MenuKey()` routine and from there to my menu handler, `DoMenuSelection()`.

```
     case keyDown:
     case autoKey:
```

```
/*If the Command key is pressed, it is a menu selection*/
        theKey = (char)(theEvent.message & charCodeMask);
        if ( ((theEvent.modifiers & cmdKey) != 0) )
          DoMenuSelection(MenuKey(theKey));  /*Our own routine for handling menu ➥
selections*/
        else
/*Otherwise, it's a normal key-down.*/
          DoKey(theKey, theEvent.modifiers);
        break;
```

The next events to deal with are update events. I verify that the update event is for a window I know about, and I call my window update routine `DoUpdate()`.

```
    case updateEvt:
/*Find out what event the update event is for.*/
        if ( (WindowPtr)theEvent.message == myWindow )
          DoUpdate();
        break;
```

The last event I handle is `diskEvt`, disk events. What happens here is that I call the disk initialization package in case an unformatted or bad disk is inserted. This isn't much to worry about. Just include it in the event loop and forget about it.

```
    case diskEvt:
/*Handle bad disk insertions*/
        if (HiWord(theEvent.message) != noErr)
        {
          DILoad();
          DIBadMount(diskInitPt, theEvent.message);
          DIUnload();
        }
        break;
    default: /*Other events are ignored*/
  } /*case*/
```

Finally, I call my "background task" routine, `DoBackground()`, once per turn through `MainLoop()`. In that routine, I can do anything that I want to do as a background task. If your game is a simulation game like SimCity, this is a perfect place to process the simulation.

```
/*For each turn to the event loop, we may to call some "background" process, e.g., ➥
animations.*/
  DoBackground();
} /*MainLoop()*/
```

Now I will use this event loop to improve the Dungeon game. I'll build on Dungeon 1 Plus from chapter 1, "Your First Game."

Dungeon 2

A major weakness with Dungeon 1 and Dungeon 1 Plus is that they have no event processing. All background tasks are stopped and the player cannot access menus.

When I wrote Dungeon 2, I plugged in event handling using the `MainLoop()` routine we just saw. Then I added a few other routines called by `MainLoop()`. I added a routine that sets up menus, and I added 'MENU' resources to the resource file using ResEdit.

The improved Dungeon 2 now has a new structure. After it initializes, everything that happens is triggered by events. Instead of polling the `Button()` function, the program waits for events. If it gets a `mouseDown` in the window, it runs the same procedure as in the original Dungeon 1 demo. It also handles `mouseDowns` in other places, such as the menu bar and the drag bar of the window. And it handles update events and key-downs (in this particular game only for handling Command-key equivalents).

This is enough to make the game acceptably Mac-friendly. In figure 5.1, you see how it looks in the game. The menus work, and so do window dragging and switching to other applications. Now I'll show you what I had to do to make this work.

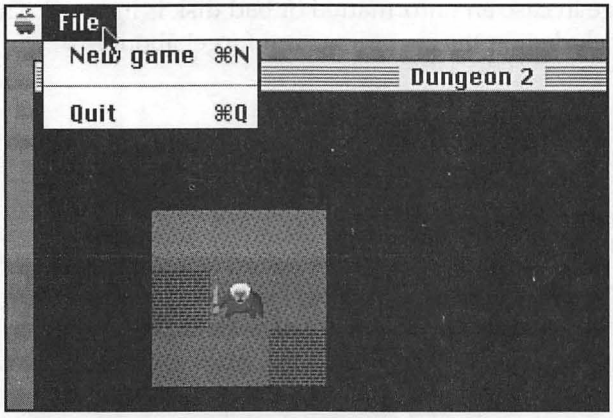


Figure 5.1 *Wow, Dungeon now has menus!*

The new flowchart (see figure 5.2) may seem complex, but it isn't more than two case switches and code for each case. The gain is that the program now responds in the expected way to all actions the player takes.

I added the menu to my resource file using ResEdit. This is done by opening the resource file and selecting Create new resource. In the resource type list, I pick MENU. The menu editor is displayed (see figure 5.3).

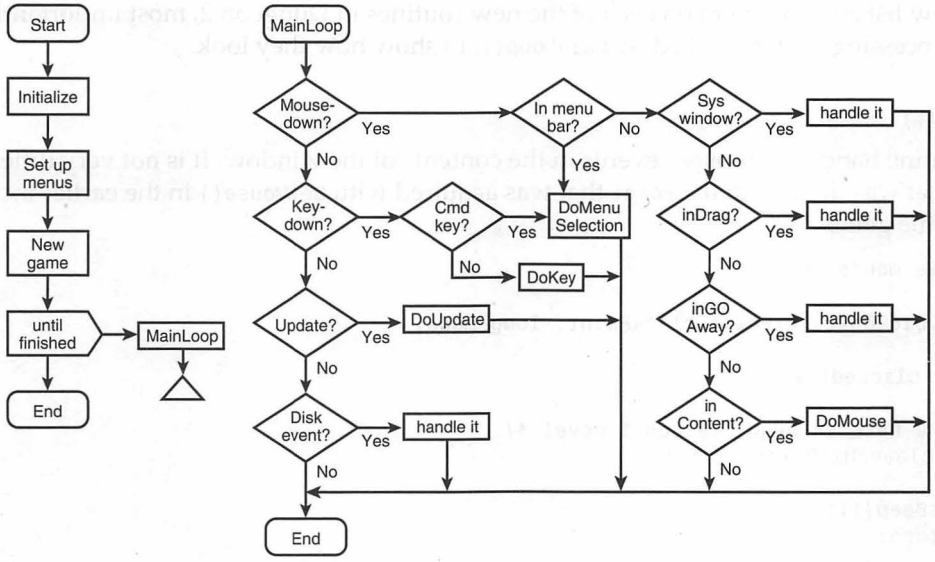


Figure 5.2 *Flowchart for the event handling in Dungeon 2.*

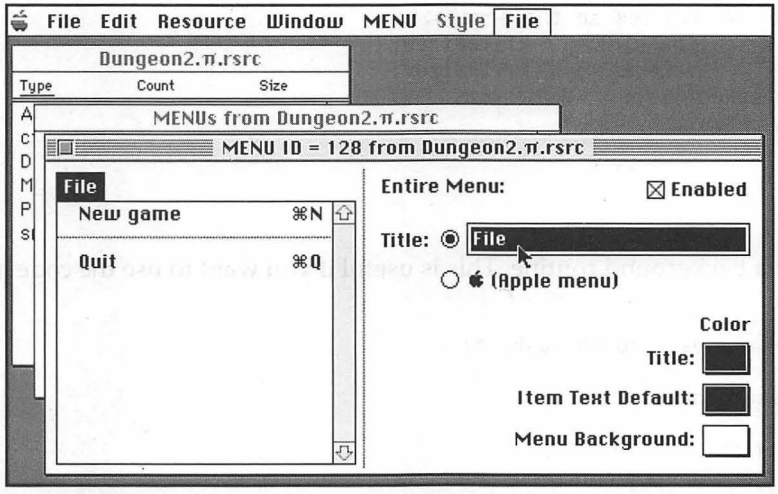


Figure 5.3 *Building a 'MENU' resource in ResEdit.*

Note that the resource ID—128 in this case—is by default the same as the menu ID number (found in the top of the MENU menu when the menu editor is open). But if you change the resource ID, the menu ID number will not follow automatically. If you use several menus with the same menu ID number, you will run into problems, typically with some menus not showing up.

I will now list and comment on each of the new routines in Dungeon 2, most importantly the event processing routines called by `MainLoop()`, to show how they look.

`DoMouse()`

This routine handles `mouseDown` events in the contents of the window. It is not very different from what was done with the `Point` that was acquired with `GetMouse()` in the earlier incarnations of the game.

```
/* Handle mouse-downs */

static void DoMouse(Point clickPoint, long mods)
{
  Point clickedTile;

/* If the hero is dead, we can't move! */
  if ( playerHitPoints < 1 )
  {
    SysBeep(1);
    return;
  };

/*Convert clickPoint to local coordinates*/
  GlobalToLocal(&clickPoint);
/*Convert it to coordinates in the arrays.*/
  clickedTile.h = clickPoint.h / kTileSizeH;
  clickedTile.v = clickPoint.v / kTileSizeV;
  MovePlayer(clickedTile);
} /*DoMouse()*/
```

`DoBackground()`

Empty stubs for a background routine. This is useful if you want to use the code as an application shell.

```
/* Empty stub for a background task. */

static void DoBackground()
{
} /*DoBackground()*/
```

`AddPoints()`

`AddPoints()` is a little utility routine that `DoKey()` uses. It simply adds each component of two points, returning the sum in a new point.

```
/* Add two points -- simplifies the key-down handler */

static Point AddPoints(Point p1,Point p2)
```

```
{
   Point dest;

   dest.h = p1.h + p2.h;
   dest.v = p1.v + p2.v;
   return dest;
}; /*AddPoints()*/
```

`DoKey()`

This routine is called when a key is pressed. If the hero is dead, the game simply beeps, but otherwise the key is translated to the desired move and `MovePlayer()` is called.

```
/* Handle key-downs */

static void DoKey(char theKey, long mods)
{
/* If the hero is dead, we can't move! */
   if ( playerHitPoints < 1 )
      {
         SysBeep(1);
         return;
      };

/* For now, we use a hard-coded key mapping. */
   switch ( theKey )
      {
      case 'd':
      case '6':
         MovePlayer(AddPoints(playerPosition, directionTable[0]));break;
      case 'e':
      case '9':
         MovePlayer(AddPoints(playerPosition, directionTable[1]));break;
      case 'w':
      case '8':
         MovePlayer(AddPoints(playerPosition, directionTable[2]));break;
      case 'q':
      case '7':
         MovePlayer(AddPoints(playerPosition, directionTable[3]));break;
      case 'a':
      case '4':
         MovePlayer(AddPoints(playerPosition, directionTable[4]));break;
      case 'z':
      case '1':
         MovePlayer(AddPoints(playerPosition, directionTable[5]));break;
      case 'x':
      case '2':
         MovePlayer(AddPoints(playerPosition, directionTable[6]));break;
      case 'c':
      case '3':
         MovePlayer(AddPoints(playerPosition, directionTable[7]));break;
```

```
    default:
      SysBeep(1);
  }; /*case*/
}; /*DoKey()*/
```

DoFileMenu()

Here, I process selections from the File menu—New game and Quit. `DoFileMenu()` is called from `DoMenuSelection()`.

```
/* Handle selections in the File menu */

static void DoFileMenu(short item)
{
  switch ( item )
    {
    case 1:
      NewGame();break;
    case 3:
      gDone = true;break;
  }; /*case*/
}; /*DoFileMenu()*/
```

DoAbout()

This routine calls up an alert, which is my About box. It is called from `DoMenuSelection()`.

```
/* The "About" item was selected */

static void DoAbout()
{
  short ignore;

  ignore = Alert(128, nil);
}; /*DoAbout()*/
```

DoUpdate()

In `DoUpdate()`, I update the game window, which has for some reason been partially or totally erased. Within a `BeginUpdate()/EndUpdate()` pair, I draw all tiles.

```
/* Handle update events */

static void DoUpdate()
{
```

```
    short h, v;

    BeginUpdate(myWindow);
/*Draw all tiles!*/
    for ( h = 0 ; h < kArraySizeH ; h++)
      for ( v = 0 ; v < kArraySizeV ; v++)
        DrawTile(h, v);
    EndUpdate(myWindow);
}; /*DoUpdate()*/
```

DoMenuSelection()

This is the menu handling central station, from where all menu handlers are called. The long `mSelect` is the result from `MenuSelect()`, which was called in `MainLoop()`. It can be taken apart into a menu ID number and the item selected in that menu.

```
/* Handle menu selections */

static void DoMenuSelection(long mSelect)
{
  short menuID;
  short menuItem;
  GrafPtr savePort;
  Str255 name;
  short ignore;

  menuID = HiWord(mSelect);
  menuItem = LoWord(mSelect);

  switch ( menuID )
    {
    case appleID:
      if ( menuItem == 1 )
        DoAbout();
      else
        {
          GetPort(&savePort);
          GetItem(GetMHandle(appleID), menuItem, name);
          ignore = OpenDeskAcc(name);
          SetPort(savePort);
        };
      break;
    case fileID:
      DoFileMenu(menuItem);break;
    default:
  ;}; /*case*/
  HiliteMenu(0);
}; /*DoMenuSelection()*/
```

`SetupMenus()`

I have a separate procedure just for setting up menus. This could just as well have been part of `InitDungeon()`, but should this program ever be expanded to have many menus, it makes sense to do this as a separate routine.

Note the assignments to `tempStr` and how I set `tempStr[0]` to 1 and `tempStr[1]` to `char(20)`. That creates a one-character-long string (Pascal style) with the Apple menu symbol.

```
/* Set up menus */

static void SetupMenus()
{
   MenuHandle appleMenu, fileMenu;
   Str255 tempStr;

   tempStr[0]=1; tempStr[1] = (char)20;
   appleMenu = NewMenu(appleID, tempStr); /*Apple menu symbol*/
   InsertMenu(appleMenu, 0);
   AppendMenu(appleMenu, "\pAbout Dungeon…;(-");
   AddResMenu(appleMenu, 'DRVR');

   fileMenu = GetMenu(128);
   InsertMenu(fileMenu, 0);
   DrawMenuBar ();

}; /*SetupMenus()*/
```

`MainLoop()`

`MainLoop()` is, as in the listing above, our event processing routine. It gets events with `WaitNextEvent()` and calls the handlers as appropriate.

Designing the User Interface

Let's step back for a moment and have a look at how your game is set up. To what extent your game needs to conform to the user interface guidelines depends on how close it is to a Mac application. With that in mind, we can distinguish between several different kinds of Macintosh game user interfaces.

Take-Over Interface

This game covers the whole screen with one nondraggable window. The menu bar is hidden most of the time (in some games always, though that is not advisable) and events are usually not processed. You will find this interface on most action games.

Examples: Prince Of Persia and Delirium

Special Considerations

Since take-over games are generally action games, they should *not* call `WaitNextEvent()` during the game, when maximum performance is essential. However, even a take-over game should process events and have a menu *when paused*. That will allow the user to switch out and update areas that are cleared out for whatever reason.

Also, as mentioned earlier, you can consider using `GetOSEvent()` even during animation if you want to get events instead of polling mouse and keyboard with `GetMouse()` and `GetKeys()`.

The game area should be *centered* on the screen, and the window should probably fill the whole screen, filling areas outside the game area with something (such as black). Don't assume that all your users have the same screen size as you have! 640 x 480 is getting small these days!

Screen-Filler Interface

This game covers the whole screen with a single, nondraggable window, but the menu bar is visible and events are processed.

Examples: 3 in Three and Monkey Island

`Special Considerations`

This looks almost like the previous one, but is generally used for a different category of games. The presence of a menu brings us much closer to the normal Mac interface. A screen-filler interface fits adventure games fairly well.

The graphics should fill either the entire window or a part of it that is centered. Make sure that you fill the entire screen with the window, and not just a 640 x 480 area. Consider an option to use a smaller, draggable window when running on a larger screen than the minimal size.

Application Interface

This game uses several draggable windows. Although the game probably has fancy graphics, it works just like any Macintosh application. This kind of interface fits simulation and strategy games very well.

Examples: SimCity, Civilization, Spaceward Ho, Bolo, and Warlords II (see figure 5.4)

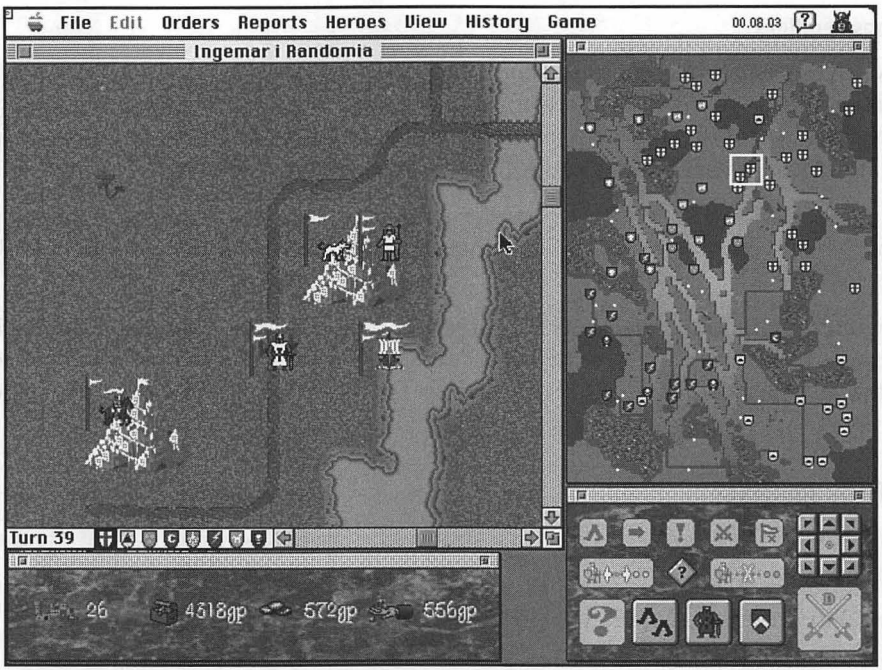


Figure 5.4 *Warlords II, a typical application interface-equipped game, with moveable windows, menus, and features conforming well to the user interface guidelines.*

Special Considerations

The game should save its window positions and have a way to reset them to let the player rearrange them as desired.

When making an "application" game, you should follow Apple's user interface guidelines closely. It's fine to replace the flat buttons with 3-D-style ones—you probably should—but it is not okay to use them in a nonstandard way.

Desk Accessory Interface

This game probably has only one window, which might be rather small, much smaller than the screen. Closing that window quits the game. This kind of interface is mostly found on "toy" games, short diversions that you can fire up for a few minutes of relaxation while working. Note that there is absolutely nothing *wrong* with "toy" games—they are just as important as the big ones, but for a different audience.

Examples: Apple's Puzzle (see figure 5.5), Hexmines, and Bikaka

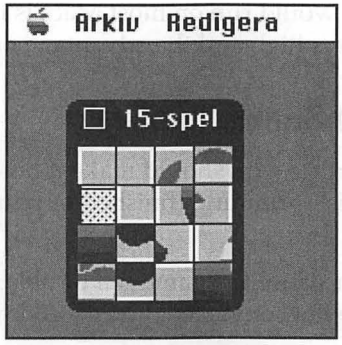


Figure 5.5 *Apple's Puzzle game (in this case the System 7.1 version), a game with a desk accessory user interface, with a moveable window, menus, and small memory footprint.*

Special Considerations

The big difference between this interface and the previous ones is that while all the others more or less fill the screen with the game, the desk accessory-style game is the kind of small toy game that sits over there in the corner as one out of many windows around the desktop.

Note that this kind of game is likely to sit in the background a lot and be switched in now and then. These games should be relatively memory-conservative, and should not slow the computer down when in the background. (Not that any other games should either.)

Which one to use? That depends on the game, and I will leave that up to you for now. (For some pointers, see chapter 12, "Game Aesthetics.")

Can I Take Over the Machine Completely to Win Performance?

Some programmers, mainly newcomers to the Mac, wonder if they can kick out the MacOS altogether to get top performance, perhaps booting from some special minimal system. The general answer is *no*.

You can win very little by doing so. Since the Mac has cooperative multitasking, you can choose not to give background processes any time. That will give you close to 100 percent of the CPU anyway.

Anything that you can win by taking over the machine completely, you can get in easier ways. You can advise your users to turn off certain features that may intervene. File sharing and virtual memory can prevent games from running smoothly, for example. Certain extensions may also cause some loss of performance.

If you did manage to take over the Mac completely and run your own show, nobody would use your game. No Mac users want to reboot their Mac for playing a game.

Writing a special-purpose OS that would run on most Macs is not realistic. Apple could do it, if it had any reason, but the cost is high and the advantage none.

Menus? We Don't Need No Stinkin' Menus!

Even if you use the take-over interface, you should make it possible to pause the game and access the menus when the game isn't running (finished or paused), even if the menu bar is hidden while playing.

The big point here is to let the user pause the game *and* be able to switch to other applications. That is important. If you are not convinced, you should read the following little story, in both its variations.

Scenario 1

Little Tommy is playing Astro Bazooka on his Mom's Mac. He is on level 38 and knows that he is close to the high score. In comes Mom.

"Tommy, I must use the Mac NOW! I must print out some important documents for an important meeting, and the bus leaves in ten minutes!"

"But I almost have the high score!"

"TOMMY! GET AWAY FROM MY MAC! NOW!"

"No!"

(Click.)

Mom turns off the Mac, turns it on again, throws the crying Tommy out of the room, waits for the Mac to restart (and wishes that she hadn't turned it off, but Tommy would never had told her how to quit)... and misses the bus.

The same evening, Mom erases all games from the hard drive. Tommy isn't happy—and he probably won't buy Astro Bazooka II.

Scenario 2

Little Tommy is playing Astro Bazooka on his Mom's Mac. He is on level 38 and knows that he is close to the high score. In comes Mom.

"Tommy, I must use the Mac NOW! I must print out some important documents for an important meeting, and the bus leaves in ten minutes!"

"Sure, Mom!"

Tommy hits Command-P (which pauses Astro Bazooka), selects Hide Astro Bazooka from the application menu, and leaves the Mac. Five minutes later, Mom rushes out the door with the documents in her hand, and Tommy brings Astro Bazooka to the front again and resumes the game. After another 15 minutes of intense playing, he gets his high score.

And they live happily ever after.

It isn't at all hard to fix this, either. Suppose you have a routine `RunGameLoop()` that runs one frame of animation and two global Booleans, `gGameOver` and `gGamePaused`, to tell if the game is paused. Whenever you start or resume a game—perhaps triggered by a menu selection—you do the following:

```
gGameOver = false;
gGamePaused = false;
while (!gGameOver && !gGamePaused)
  RunGameLoop();
```

The Booleans must, of course, be set by `RunGameLoop()` or some routine called by it. When the game ends, `gGameOver` is set to TRUE, and when some key or key combination denoting pause is hit, `gGamePause` it set to TRUE. Command-P is a good standard for pausing, as long as printing isn't supported. You can also consider allowing the user to abort the game, in which case you set `gGameOver` to TRUE. Command-period is a good combination for that.

Rules for Standard Menu Layout

So, I hope you agree that your game should have menus (at least sometimes). The minimum is a standard Apple menu and a File menu with Quit. If you use bigger menus, there are a few rules to conform to:

The first menu from the left is the Apple menu. You should have an About MyGame selection first (with the name of your program, not just About), and possibly Help if you like (though that is better off in the Help menu). The rest should be what the System wants, which you get by the following call:

```
AppendResMenu(appleMenu, 'DRVR');
```

The resulting menu will include everything the user has in the Apple menu folder, just like in all other programs. Figure 5.6 shows how the menu looks from inside Dungeon when I run it. (You may notice some differences between your Apple menu and mine, since I use the Swedish OS.)

The second menu should have a Quit item at the end and should preferably be named File. If you have New, Save, Open, or even Print (in the rare cases that exist in a game), they should be in the File menu, at the standard positions, with the standard Command-key equivalents (see figure 5.7).

The third menu should be the Edit menu and hold Undo, Cut, Copy, and Paste in the standard positions and with the standard Command-key equivalents (see figure 5.8). Even if you don't support this menu, it will automatically be used if you use dialog boxes with editable text boxes, including the Standard File dialogs.

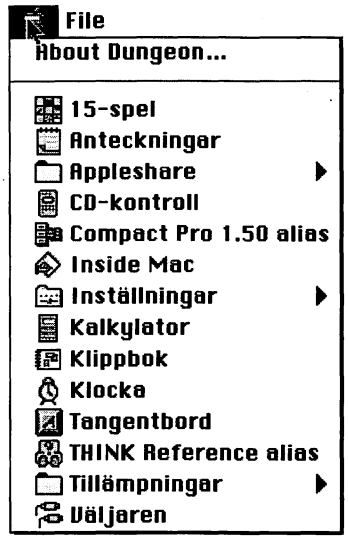


Figure 5.6 *Your game should support the standard Apple menu.*

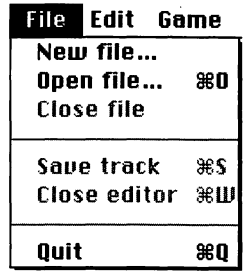


Figure 5.7 *A typical File menu for a game that supports external files (Ingemar's Skiing Game).*

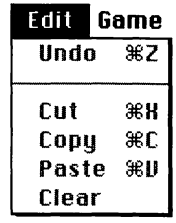


Figure 5.8 *A game should provide an Edit menu even if editing is not directly supported.*

There's more advice to find in Apple's guidelines, but not much more that really applies to games. Finally, some short notes:

- Any menu selection that fires up a dialog box should have the "..." (ellipsis) character at the end.
- Use hierarchical menus when appropriate, but not more levels than necessary.

Centering the Window

If you design only for your own configuration, you probably place the window(s) in proper positions for your screen. If you have a 640 x 480 screen, that implies that you put the window in the coordinates (0, 0, 640, 480). However, if the placement is exactly the same if the game is moved to a Mac with a bigger screen (or smaller, for that matter) then the game looks bad. Nondraggable windows should be centered on large screens if you can't make the game rescale to fit the entire screen. If you have draggable windows, it isn't that bad, but do consider saving the window positions and sizes between sessions.

Figure 5.9 below shows an example. If your game is designed for a certain screen, which it fills completely (A), then the game should not end up in the corner of a larger screen (B), but centered (C) or rescaled to fit (D).

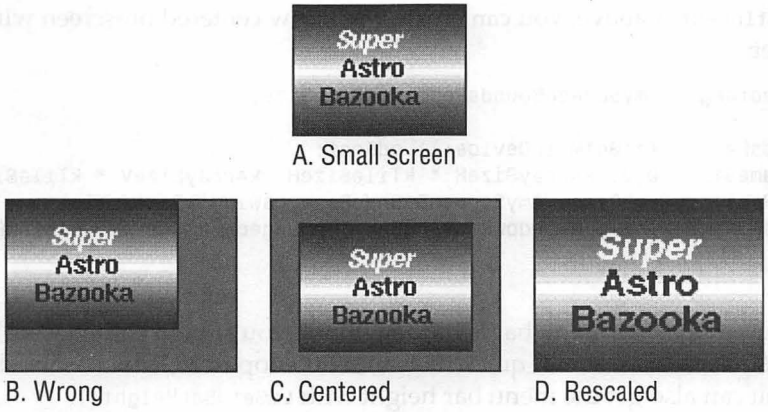


Figure 5.9 *Your game should place itself correctly on any screen.*

When centering the window, you should get the rectangle of the screen from the `GDHandle`. For the main screen, you get it as:

```
myScreenBoundsRect = (**GetMainDevice()).gdRect
```

If you support noncolor QuickDraw (i.e., 68K-based Macs), you should use `screenBits.bounds` when running on such a Mac.

To center a rectangle `rect1` in a bigger rectangle `rect2`, you can do this:

```
# define RectHeight(rect) ((rect)->bottom - (rect)->top)
# define RectWidth(rect) ((rect)->right - (rect)->left)

/************************************************************
*
* CenterRectInRect()
*
* Center source rect 1 in source rect 2 and put the resulting rect in
* destination rect.
*/

static void CenterRectInRect(Rect *source1Rect, rect *source2Rect, Rect
*destinationRect)
{
   short dx = RectWidth(source1Rect), dy = RectHeight(source1Rect);

   destinationRect->left = source2Rect->left + (RectWidth(source2Rect)>>1) - (dx>>1);
   destinationRect->top = source2Rect->top + (RectHeight(source2Rect)>>1) - (dy>>1);
   destinationRect->right = destinationRect->left + dx;
   destinationRect->bottom = destinationRect->top + dy;
} /*CenterRectInRect()*/
```

Using `CenterRectInRect()` above, you can create a window centered on-screen with the following snippet:

```
   Rect windowRectangle, myScreenBoundsRect, myGameSize;

   myScreenBoundsRect = (**GetMainDevice()).gdRect;
   SetRect(&myGameSize, 0,0, kArraySizeH * kTileSizeH, kArraySizeV * kTileSizeV);
   CenterRectInRect(&myGameSize, &myScreenBoundsRect, &windowRectangle);
   myWindow = NewCWindow(nil, &windowRectangle, "\pDungeon 2", true, plainDBox, ➥
(WindowPtr)-1L, false, 0);
```

This snippet assumes that the menu bar is hidden. If not, you should compensate by changing the top of `myScreenBoundsRect`. A quick fix is to set the top to 20, which is the default menu height. You can also get the menu bar height with `LMGetMBarHeight()`.

Now try it yourself: Modify the Dungeon demo to get a centered window!

Hiding the Desktop

As mentioned previously, many games fill the rest of the screen with a solid color (usually black) when run on a screen that is larger than the game area. In the optimal case, this is given as an option so the user can choose.

Hiding the desktop takes away some of the user's freedom. The user can't click in a Finder window or a window from another application. However, this is no major problem as long as the menus are available.

There are two ways to cover the desktop that I would recommend. One is to enlarge the window to cover the entire screen and use `SetOrigin()` on the window to place the game area centered on the screen. The other is to put another window behind the game window. In the latter case, just watch out so you don't bring that window to the front, hiding your game.

I personally prefer using one window and changing the origin. You can do it like this:

```
Rect windowRectangle, myScreenBoundsRect, myGameSize;

myScreenBoundsRect = (**GetMainDevice()).gdRect;
SetRect(&myGameSize, 0,0, kArraySizeH * kTileSizeH, kArraySizeV * kTileSizeV);
CenterRectInRect(&myGameSize, &myScreenBoundsRect, &windowRectangle);
myWindow = NewCWindow(nil, &myScreenBoundsRect, "\pDungeon 2", true, plainDBox, ➥
(WindowPtr)-1L, false, 0);
SetPort(myWindow);
SetOrigin(-windowRectangle.left, -windowRectangle.top);
```

Note that I still use `CenterRectInRect()`, but only to get the origin, which is the top left corner of the resulting rectangle.

This leaves an unused area around the game area. It usually looks nicer if you fill that with black or some other color or pattern that makes sense within the game. As a quick fix, try a `PaintRect()` on your game window before doing the rest of the update:

```
PaintRect(&myWindow->portRect);
```

Hiding the Menu Bar

Let's get into another thing that most game authors do—hide the menu bar. Just like moving the mouse pointer, this is something Apple discourages and for which it provides no quick and simple solution.

Again, you must understand why hiding the menu bar is bad before doing it. The bad part is simple enough: You don't let the user issue commands the normal way. The user can't go to the application menu to switch programs, can't go to the File menu to Quit, and can't go to the Apple menu to launch applications or control panels.

The correct solution is to hide the menu bar but provide a way for the user to get it back at any time. You can show the menu bar if the player clicks where the menu bar used to be or when some key combination is made. When I make action games, I show the menu bar whenever the game is paused.

Okay, let's move on to how to do it. You hide the menu bar by expanding the visRgn of your game window to cover the menu bar. To show it again, you restore the visRgn and call the DrawMenuBar() procedure.

For saving the visRgn, we need a static variable, gSaveVisRgn. With the following code, gSaveVisRgn can be used for determining if the menu bar is hidden. If it is NIL, then the menu bar is visible.

```
static RgnHandle  gSaveVisRgn = nil;

void HideMenuBar()
{
```

If gSaveVisRgn is not NIL, then the menu bar is already hidden.

```
  if (gSaveVisRgn != nil)
    return;
```

Create a region and copy the visRgn to it.

```
  gSaveVisRgn = NewRgn();
  CopyRgn(myWindow->visRgn, gSaveVisRgn);
```

Change the visRgn to the entire window.

```
  RectRgn(myWindow->visRgn, &myWindow->portRect);
```

Signal that an update event is wanted. An alternative here is to call some drawing routine directly.

```
  InvalRect(&myWindow->portRect);
} /*HideMenuBar()*/
```

Having hidden the menu bar, you must also be able to show it again. That is done with the following routine.

```
void ShowMenuBar()
{
  RgnHandle tempRgn;
```

First of all, if gSaveVisRgn is NIL, the menu is already visible!

```
  if (gSaveVisRgn == nil)
    return;
```

The Mac screen is clipped by a rounded-corner rectangle. I must paint those corners, or they will stay whatever colors my game painted in its window.

This is done by creating a temporary region, `tempRgn`, and setting it to the difference between the current (full-screen) `visRgn` and the saved one. The resulting area is painted black.

```
tempRgn = NewRgn();
DiffRgn(myWindow->visRgn, gSaveVisRgn, tempRgn);
ForeColor(blackColor);
PaintRgn(tempRgn);
DisposeRgn(tempRgn);
```

You restore the `visRgn` by copying from `gSaveVisRgn`. Note that this will cause weird results if you have moved the window or if you have several windows!

```
CopyRgn(gSaveVisRgn, myWindow->visRgn);
```

Then, `gSaveVisRgn` is disposed of and set to NIL.

```
DisposeRgn(gSaveVisRgn);
gSaveVisRgn = nil;
```

Finally, a call to `DrawMenuBar()` makes the restoration complete!

```
  DrawMenuBar();
} /*ShowMenuBar()*/
```

However, if your game processes events while the menu bar is hidden, you must either change the `mouseDown` handling, so clicks in the menu bar are interpreted as clicks in the game window, or change the menu bar height.

The menu bar height is a global at the address 0BAA (hexadecimal). If you set it to zero, `FindWindow()` will no longer recognize any mouse-downs as being in the menu bar. I personally dislike relying on low-memory globals, so if you can solve the problem as suggested above, it is preferable.

Dialogs

When you want to ask the player for key configuration, other preferences, or the player's name for a new high score entry, it makes perfect sense to use the Dialog Manager in the Mac's Toolbox. You can then use ResEdit for all or most of the dialog design, storing the dialog in a 'DLOG' resource. This saves coding time and reduces the risk of introducing bugs, but most of all it gives the game an interface that the players are familiar with.

I have seen games in which dialogs hardly look like dialogs at all, and the buttons—if they can be called buttons—have the same meaning as the standard OK and Cancel, but with icons that make no sense to me. The odd interface makes the game experience more frustrating and less enjoyable.

On the other hand, the standard flat controls may look too dull for a game. Fortunately, there are Public Domain controls definitions (CDEFs) that you can plug into your game. A particularly nice set is 3D Buttons by Zig Zichterman. His controls not only look nice, but also are so similar to the standard, flat controls that they are neither confusing nor distracting. I use 3D Buttons in the figures that follow.

Dialog Items

A dialog, defined by a 'DLOG' resource, has an accompanying 'DITL' resource, which is the dialog item list. When you edit the item list, ResEdit shows a window that looks like figure 5.10.

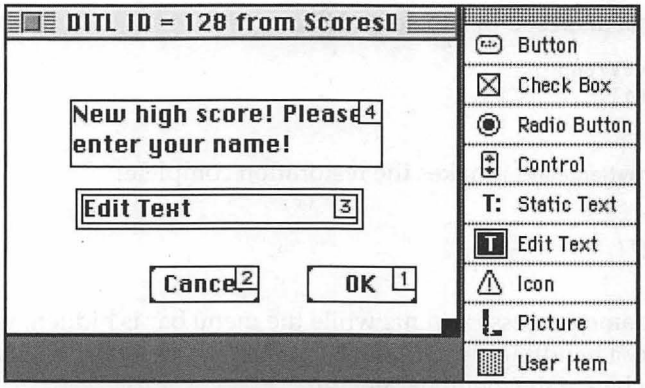


Figure 5.10 *Editing dialog items.*

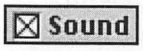
The dialog item list can contain the following item types:

Pushbuttons



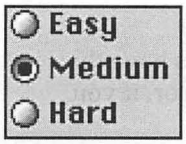
Pushbuttons are used for immediate actions. OK and Cancel buttons are recommended for dismissing a modal dialog.

Checkboxes



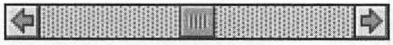
These are used to answer a yes/no question. However, if your dialog poses only one yes/no question, that is better done with two pushbuttons, both dismissing the dialog when hit.

Radio Buttons



These are used for selecting one choice out of several. You never use a single radio button.

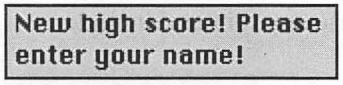
Controls



Control items are usually used for scroll bars, but can also be used for other kinds of advanced controls (most of them similar in function to a scroll bar). To include a scroll bar in your dialog, you must create an accompanying 'CNTL' resource in which you specify its bounds, values, and a `procID` of 16. Scroll bars are used for setting a value in a large or variable range, or for scrolling through text or pictures. Scroll bars in dialogs are fairly rare in games. They have some use in strategy or simulation games.

Scroll bars, with their many parts, are rather tedious to program. The StylDialog demo, which you can find in the 05 - User Interaction folder on the CD-ROM, uses one. You can also look in any TextEdit sample code to learn how they work.

Static Texts



Static text items are texts that can't be edited but can be changed from your program. When you put in text that is just information, consider using a PICT instead. Static text items are of most interest when you need text that needs to be changed from your program (with `SetDialogItemText()`).

Editable Texts



Obviously, editable text items are used for text input. They require no programming effort except for putting in text before running the dialog and taking out the result once the user has dismissed the dialog. If you provide a standard Edit menu, you will get cut and paste functions automatically.

Icons

An icon item must be 32 x 32 pixels. Pictures are generally preferable, but icons have the advantage that they can switch automatically between black and white and color, if you provide both a 'ICON' resource and a 'cicn' resource (see figure 5.11).

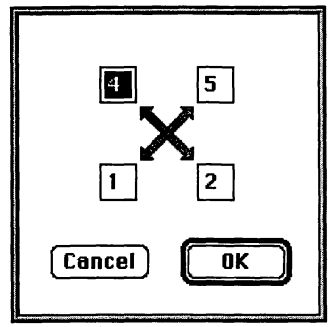


Figure 5.11 *A dialog with an icon item to clarify the meaning of four editable text fields (movement keys in four directions).*

Pictures

These items specify a 'PICT' resource in the game file. The picture is resized to fit the item's rectangle, but you can order ResEdit to set it to the size of the 'PICT' resource. Since PICTs are so extremely flexible, this is the preferred way to get graphics into your dialogs.

A picture is also often the best way to put static texts into your dialogs. A static text is quicker and has no problems with missing fonts, but PICTs can use any font and include pictures copied from a painting program or even a raytracer.

A drawback is that if the PICT is a pixmap (e.g., copied from Photoshop or a screen dump), it is hard to translate the text to other languages without ruining the text. If, on the other hand, a PICT contains text objects (e.g., copied from FreeHand), the text must be in a standard font, or it might be missing on the user's computer. Another disadvantage of a PICT is that it uses more memory than does static text.

User Items

These are just rectangular areas—the catch-all in dialogs. You use them for anything that the standard types aren't good for. You can, for example, implement a list using the List Manager and a user item to define its position. User Items require handling of update events in a

dialog filter function (see the "Dialog Filter Functions" section later in this chapter) or by a routine installed in the item by `SetDialogItem()` (passing a procedure pointer instead of a handle). I've never needed user items for games myself, though.

Always use each kind of control for what it was meant for. If you use a button, don't make it behave like a checkbox.

Dialogs in Games

There are two kinds of dialogs that are most common in games, and I will give examples of both with source code. One is the preferences dialog, which may include key configuration and other settings (see figure 5.12). The other is the simple dialog box asking for your name when you have a high score (see figure 5.13).

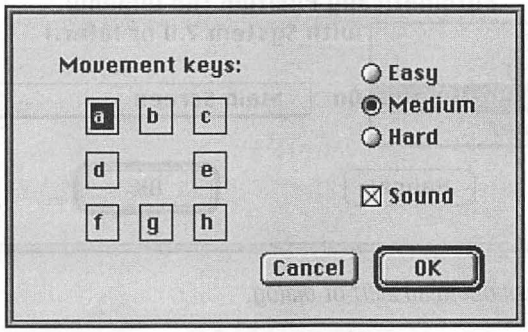


Figure 5.12 *A preferences dialog.*

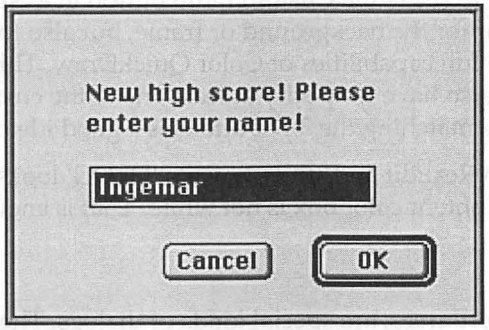


Figure 5.13 *A dialog asking for the player's name.*

Using a Dialog

The easiest way to make dialogs is to make modal ones. Although modeless dialogs (dialogs that do not limit access to other parts of the program) are generally preferable, modal dialogs are adequate for most needs of a game.

You use a modal dialog in the following way:

- Define and design the dialog as a 'DLOG' resource.
- Load it into memory with `GetNewDialog()`.
- Use it with `ModalDialog()`.
- Get rid of it with `DisposeDialog()`.

To center a 'DLOG', you open it in ResEdit and select Auto position from the DLOG menu. A dialog with two pop-up menus appears, as shown in figure 5.14. Select whatever you think appropriate. Center or Alert Position are usually good picks.

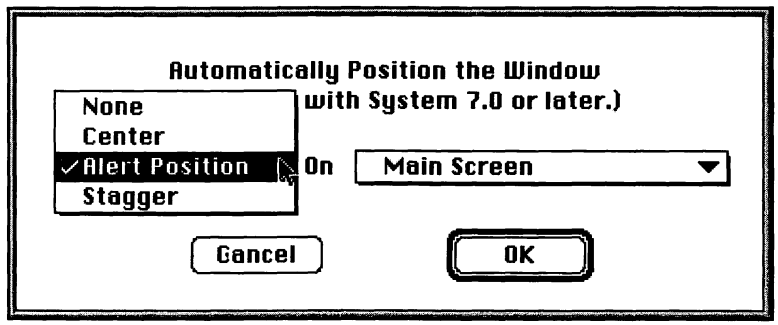


Figure 5.14 *Automatically positioning an alert or dialog.*

Color in Dialogs

If you want color, you need to select the Custom radio button in ResEdit's dialog editor. This is not only for setting a color for the background or frame, but also for making the dialog a color window, allowing the full capabilities of Color QuickDraw. This is needed for making the 3-D buttons work. You also have to specify at least one of the custom colors. Setting the Content color to a light gray matching the 3-D buttons is a good idea.

Open ScoresDemo.rsrc with ResEdit and study how its 'DLOG' looks. The Custom radio button is checked and the Content color box is not white. That is enough.

Alerts

Alerts, defined by 'ALRT' resources, are special kinds of dialogs. They can be used by a single call, which makes them very convenient for error messages and About boxes. Alerts can be centered just like dialogs, by the Auto position feature in the ALRT menu.

In Dungeon 2, I use an 'ALRT' for the About box (see figure 5.15). It is extremely simple. You surely want a fancier About box in your game, but an alert can still be a good solution. How about putting in a PICT?

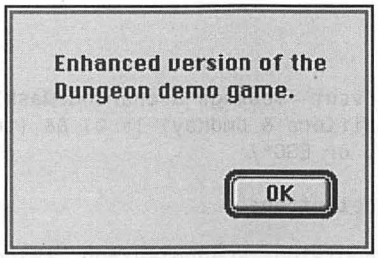


Figure 5.15 *The About Dungeon alert box in Dungeon 2.*

The alert is used as follows:

```
/* The About item was selected */

static void DoAbout()
{
   short ignore;

   ignore = Alert(128, nil);
}; /*DoAbout()*/
```

One line of code! Can you imagine an easier way to display a message?

Dialog Filter Functions

To make your dialog behave perfectly according to the interface standards, you need to include a "filter function" for your modal dialogs. This kind of function is also needed if you want to add user items.

My "standard" dialog filter function implements Command-period and ESC for Cancel, Enter for OK; it also frames the default button.

```
/* StdFilter() */

/* StdFilter() is a dialog filter function that is useful for most
modal dialogs, as long as they have an OK and a Cancel button.
It maps return and enter to OK, and Command-period and ESC to Cancel.
It also draws a frame around the OK button. */

static pascal Boolean StdFilter(DialogPtr theDialog, EventRecord *theEvent, short ➥
*itemHit)
{
   char theChar;
   short kind;
   Handle item;
   Rect box;
```

```
  switch ( theEvent->what )
    {
    case keyDown:
         theChar = (char)(theEvent->message & charCodeMask);
         if ( (((theEvent->modifiers & cmdKey) != 0) && (theChar == '.')) ||
(theChar == (char)27) ) /*Cmd-. or ESC*/
           {
             *itemHit = kCancelButton;
/*Highlight the Cancel button*/
             GetDialogItem(theDialog, kCancelButton, &kind, &item, &box);
             HiliteControl((ControlHandle)item, 1);

             return true;
           };
         if ( (theChar == (char)13) || (theChar == (char)3) )
           {
             *itemHit = kOKButton;
/*Highlight the OK button*/
             GetDialogItem(theDialog, 1, &kind, &item, &box);
             HiliteControl((ControlHandle)item, kOKButton);

             return true;
           };
         break; /*keyDown*/
    case updateEvt:
         BeginUpdate(theDialog);
         SetPort(theDialog);

         DrawDialog(theDialog);

/*Frame default button -- item 1*/
         GetDialogItem(theDialog, kOKButton, &kind, &item, &box);
         InsetRect(&box, -4, -4);
         PenSize(3, 3);
         FrameRoundRect(&box, 15, 15);

         EndUpdate(theDialog);
         break; /*update event*/
  }; /*case*/
  return false;
} /*StdFilter()*/
```

When passing a filter function to `ModalDialog()`, you pass a pointer to the function. As long as you are compiling standard Macintosh 68K code, this will work fine, both on 68K Macs and on the new Power Macs. You call `ModalDialog()` like this:

```
ModalDialog(&StdFilter, &itemHit);
```

However, if you are writing *native Power Mac* code, specifically compiled so it will run only on Power Macs, then you have to do something slightly different. You pass the address to the function to `NewRoutineDescriptor()`, which will then provide a pointer to pass to `ModalDialog()`.

```
  ProcPtr filterProc;

  filterProc = NewRoutineDescriptor(&StdFilter, uppModalFilterProcInfo, ➥
GetCurrentISA());
  ModalDialog(filterProc, &itemHit);
```

Values of Dialog Items

When getting or setting values of dialog items, you must call `GetDialogItem()`, which gives you a handle to the item, plus its type and bounding box. With that handle, you can call `SetIText()`, `GetIText()`, `GetDialogItemText()`, `SetDialogItemText()`, `GetControlValue()`, `SetControlValue()`, and other Dialog Manager and Control Manager routines.

This way of getting and setting values of dialog items is a bit clumsy, so many Mac programmers make some kind of dialog utility module for making it more straightforward. Here is an example for setting the value of a Boolean dialog item that is a checkbox or radio button:

```
void SetBooleanDItem(DialogPtr theDialog, short itemNumber, Boolean theBoolean)
{
  short itemKind;
  ControlHandle itemHandle;
  Rect itemBox;

  GetDialogItem(theDialog, itemNumber, &itemKind, (Handle *)&itemHandle, &itemBox);
  SetControlValue(itemHandle, (short)theBoolean); break;
} /*SetBooleanDItem()*/
```

This is what you will find in many code examples. The point is not to use just one line of code instead of two, but to get rid of some variables. With fewer variables, programming is quicker and more comfortable.

On the CD-ROM you'll find a variant of KeyConfig (a demo which is described in a bit) that is modified to use a set of routines that handle dialog items this way. The demo is in the 05 - User Interaction folder. DialogUtils.c holds a number of routines like the one above, though with some error checking added. The error checking will save you from crashes should you ever confuse the item numbers, e.g., treating a static text as a button.

The Dialog in ScoresDemo

ScoresDemo is a program that demonstrates how to handle a high score list, asking for the player's name when there is a new high score and saving the high scores to a preferences file. Below, I list the part of the program that is relevant for this chapter, namely the dialog management. The preference file handling is discussed in chapter 4, "File Management."

The dialog was shown as figure 5.13 a few pages back. It contains an OK button, a Cancel button, an editable text, and a static text. The actual 'DLOG' resource has resource ID of 128.

```
#define kHighDlogRes 128

/* Ask for player's name (at high score) */
static void AskHigh(Str255 name)
{
```

The `DialogPtr` dialog is used to refer to the dialog. The `GrafPtr oldPort` is used for restoring the port once the dialog is done. The short `itemHit` is used with `ModalDialog()` to return the item that was affected.

```
  DialogPtr dialog;
  GrafPtr oldPort;
  short itemHit;
```

For calling `GetDialog()`, three variables for return values are needed.

```
  Handle itemHandle;
  short itemType;
  Rect itemRect;
```

The following string is used for getting the name out of the dialog once OK is clicked on.

```
  Str255 str;
```

Save the port and create the dialog.

```
  GetPort(&oldPort);
  dialog = GetNewDialog(kHighDlogRes, nil, (WindowPtr)-1);
  ShowWindow(dialog);
  SelectWindow(dialog);
  SetPort(dialog);
```

Set the editable text field to an initial value. In the demo, I use a hard-coded string. Consider saving the last name that was used and using that here instead.

```
  GetDialogItem(dialog, 3, &itemType, &itemHandle, &itemRect);
  SetDialogItemText(itemHandle, "\pYour name here");              /*Insert string ➥
from the prefs file here*/
```

Select the name, so the player can replace it by just typing over it, without having to use the mouse or press the delete key.

```
SelectDialogItemText(dialog, 3, 0, 32767);
itemHit = -1;
```

Run `ModalDialog()` over and over until one of the buttons is hit.

```
while ( (itemHit != 1) && (itemHit != 2) ) /* 1=ok, 2=cancel */
  ModalDialog(&StdFilter, &itemHit);
```

If the dialog is cancelled, I just return an empty string.

```
if ( itemHit == 2 )
{
  name[0] = 0; /* Empty string */
};
```

If OK is hit, I get the string from the dialog and return that.

```
if ( itemHit == 1 )
{
  GetDialogItem(dialog, 3, &itemType, &itemHandle, &itemRect);
  GetDialogItemText(itemHandle, name);
};
```

Clean up and leave!

```
  SetPort(oldPort);
  DisposeDialog(dialog);
} /*AskHigh()*/
```

The KeyConfig Demo

As I said above, the two most commonly needed dialogs when making games are a dialog asking for the player name at high score, as in ScoresDemo, and a dialog for settings and key configuration. Now it's time for key configuration in the demo KeyConfig, which is on the CD-ROM. KeyConfig also demonstrates checkboxes and radio buttons, but most importantly it shows how to catch a key click in a dialog together with its key code so it can be used for `GetKeys()` later.

I use editable text fields, one for each key that is set. Please do not take this as the only way you can write a key configuration dialog. It has some drawbacks, namely that some keys are not allowed at all (Return, Enter, Escape, and modifier keys) and it doesn't display the special keys it does allow, such as Arrow keys and Backspace.

However, that is not the point. The point is to pick up the key code as well as the character, display the character, but use the key code in the game.

The dialog that is displayed by KeyConfig looks almost like one of my dialog examples above, with the addition of a static text field next to every editable text field. In the static text field, the key code is displayed (see figure 5.16).

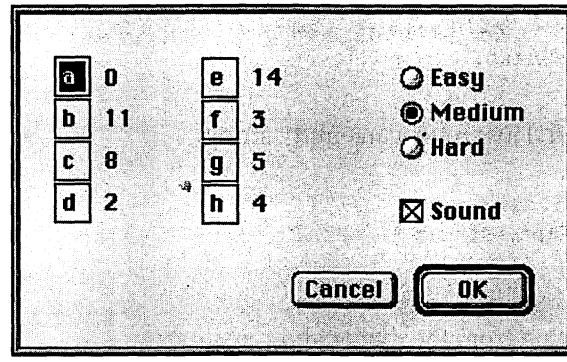


Figure 5.16 *The KeyConfig demo.*

KeyConfig is fairly long, since it saves its information in resources in the preference file (see chapter 4). It also needs more features in the filter function than ScoresDemo needed. To simplify the demo, I use the standard filter function (`StdFilter()`) much as before, but I have a new filter function (`KeysFilter()`) that passes on control to the standard one as needed but also takes care of key-downs as needed.

KeyConfig Source Code

First, I define a structure type `PrefsRec` with all the variables that are set with the dialog in KeyConfig. One such structure is saved in the preference file. From the dialog, all references to this structure are done through the global handle `gPrefs`, which points to a `PrefsRec` structure.

```
/* Prefs record */

typedef struct
{
   char keyCodes[8]; /* Key codes */
   char keyChars[8]; /* ASCII values */
   short     myRadio;
   Boolean myCheckBox;
} PrefsRec;

typedef PrefsRec **PrefsHnd;

PrefsHnd gPrefs;
```

I need a global temporary storage for key codes, since they are set in `KeysFilter()`, but the `AskKeys()` function decides whether to save them to `gPrefs` or not.

```
static char gTempKeyCodes[8]; /* Key codes, temporary storage while the dialog is ➥
open */
```

The following constants define the number of the 'DLOG' resource and the items it has. These numbers *must* be correct or the program may crash. Of course, with rigorous error checking, it won't crash. Check out the demo KeyConfig with DialogUtils to see how such error checking can be done.

```
/* The following constants must match the 'DLOG'/'DITL' resource! */
#define kHighDlogRes    128
#define kOKButton       1
#define kCancelButton   2
#define kFirstKey       3
#define kLastKey        10
#define kFirstRadio     11
#define kLastRadio      13
#define kCheckBox       14
#define kFirstCode      15
#define kLastCode       22
```

`StdFilter()` is also in the sources, but I omit it here. It is the same as above.

```
/* StdFilter() */
…
```

`KeysFilter()` now is between `ModalDialog()` and `StdFilter()`. If the event is a `keyDown` but not one that should be handled by `StdFilter()`, then `KeysFilter()` handles the event and stores the resulting key code. In other cases, control is passed over to `StdFilter()`.

```
static pascal Boolean KeysFilter(DialogPtr theDialog, EventRecord *theEvent, short
*itemHit)
{
   short    item;
   short    theKeyNumber;

   Handle   itemHandle;
   short    itemType;
   Rect     itemRect;
   Str255   str;

   if ( (theEvent->modifiers & cmdKey) != 0 )              /*Command key held down*/
      return StdFilter(theDialog, theEvent, itemHit);
   else if ( theEvent->what != keyDown )                   /*Not a key-down*/
      return StdFilter(theDialog, theEvent, itemHit);
   else if ( (theEvent->message & charCodeMask) == 3 )     /*Enter*/
      return StdFilter(theDialog, theEvent, itemHit);
```

```
  else if ( (theEvent->message & charCodeMask) == 13 )      /*Return*/
    return StdFilter(theDialog, theEvent, itemHit);
  else if ( (theEvent->message & charCodeMask) == 27 )      /*Escape*/
    return StdFilter(theDialog, theEvent, itemHit);
  else if ( (theEvent->message & charCodeMask) == 9 )       /*TAB*/
    return StdFilter(theDialog, theEvent, itemHit);
  else
  {
/* Get the key code */
    theKeyNumber = (theEvent->message & keyCodeMask) >> 8;

    if ( IsDialogEvent(theEvent) )
      if ( DialogSelect(theEvent, &theDialog, &item) )
      {
        *itemHit = item;

/* If it is one of the key boxes, put the key code in the temporary array */
        if (*itemHit >= kFirstKey && *itemHit <= kLastKey)
        {
          gTempKeyCodes[*itemHit - kFirstKey] = theKeyNumber;

/* Just for the demo, I also display the key codes. A real game wouldn't do that. */
          NumToString(theKeyNumber, str);
          GetDialogItem(theDialog, *itemHit - kFirstKey + kFirstCode, &itemType, ➥
&itemHandle, &itemRect);
          SetDialogItemText(itemHandle, str);
          return true;
        }
      }
  }
}; /* KeysFilter()*/
```

`AskKeys()` is the dialog handler. It is similar to `AskHigh()` from ScoresDemo, but bigger since it handles more items.

```
/* AskKeys() */
/* This is the procedure that handles the dialog. */

static Boolean AskKeys(PrefsHnd prefs)
{
  DialogPtr theDialog;
  GrafPtr oldPort;
  short itemHit;
  Handle itemHandle;
  short itemType;
  Rect itemRect;
  Str255 str;
  short  itemNo;
  Boolean theFlag;
  Boolean returnValue = false;
```

Set up the dialog...

```
GetPort(&oldPort);
theDialog = GetNewDialog(kHighDlogRes, nil, (WindowPtr)-1);
ShowWindow(theDialog);
SelectWindow(theDialog);
SetPort(theDialog);
```

Initialize all the dialog items...

```
/* Set all dialog items to their initial values */

/* First set all the editable text boxes… */
  for (itemNo = kFirstKey; itemNo <= kLastKey; itemNo++)
  {
    GetDialogItem(theDialog, itemNo, &itemType, &itemHandle, &itemRect);
    str[0] = 1;                                  /* Length 1 */
    str[1] = (**prefs).keyChars[itemNo - kFirstKey];       /* Put the char in the ➥
string */
    SetDialogItemText(itemHandle, str);
  }

/* …then the radio buttons… */
  for (itemNo = kFirstRadio; itemNo <= kLastRadio; itemNo++)
  {
    GetDialogItem(theDialog, itemNo, &itemType, &itemHandle, &itemRect);
    SetControlValue((ControlHandle)itemHandle, (short)(itemNo == ➥
(**prefs).myRadio));
  }

/* …and the checkbox… */
  GetDialogItem(theDialog, kCheckBox, &itemType, &itemHandle, &itemRect);
  SetControlValue((ControlHandle)itemHandle, (short)((**prefs).myCheckBox));

/* Finally, copy all the key codes to a temporary storage. */
  for (itemNo = 0; itemNo <= kLastKey - kFirstKey; itemNo++)
  {
    gTempKeyCodes[itemNo] = (**prefs).keyCodes[itemNo];

/* Just for the demo, I also display the key codes. A real game wouldn't do that. */
    NumToString((**prefs).keyCodes[itemNo], str);
    GetDialogItem(theDialog, itemNo + kFirstCode, &itemType, &itemHandle, ➥
&itemRect);
    SetDialogItemText(itemHandle, str);
  }


/* One more thing: Let the first key be selected */
  SelectDialogItemText(theDialog, kFirstKey, 0, 1);
```

Run until OK or Cancel are selected. If a radio button or the checkbox is hit, handle it.

```
  itemHit = -1;
  while ( (itemHit != 1) && (itemHit != 2) ) /* 1=ok, 2=cancel */
  {
    ModalDialog(&KeysFilter, &itemHit);

/* Handle radio buttons and checkboxes here */
    if (itemHit == kCheckBox)
    {
      GetDialogItem(theDialog, itemHit, &itemType, &itemHandle, &itemRect);
      theFlag = GetControlValue((ControlHandle)itemHandle);
      SetControlValue((ControlHandle)itemHandle, (short)!theFlag);
    }

    if (itemHit >= kFirstRadio && itemHit <= kLastRadio)
      for (itemNo = kFirstRadio; itemNo <= kLastRadio; itemNo++)
      {
        GetDialogItem(theDialog, itemNo, &itemType, &itemHandle, &itemRect);
        SetControlValue((ControlHandle)itemHandle, (short)(itemNo == itemHit));
      }
```

If an editable text field is affected, I select the entire field. That stops the user from typing more than one character in each box. It can be argued whether or not this is a proper use of a text box. If you don't think it is, there are plenty of other ways to make a key configuration box. Some games don't use text boxes at all, but rather let you click on a button and then hit a key. I find my method more comfortable and Mac-like.

```
/* If it is a key, select it */
    if (itemHit >= kFirstKey && itemHit <= kLastKey)
      SelectDialogItemText(theDialog, itemHit, 0, 32767);

  } /*while*/
```

Done! If OK is selected, copy all the results into `gPrefs`!

```
/* If not cancelled, put the results in the prefs structure. */

  if ( itemHit == 1 )
  {
/* Get values from the dialog items */

/* First get all the editable text boxes… */
    for (itemNo = kFirstKey; itemNo <= kLastKey; itemNo++)
    {
      GetDialogItem(theDialog, itemNo, &itemType, &itemHandle, &itemRect);
      GetDialogItemText(itemHandle, str);
      if (str[0] != 0)       /* If it's empty, leave it alone. */
        (**prefs).keyChars[itemNo - kFirstKey] = str[1];
    }

/* …then the radio buttons… */
```

```
    for (itemNo = kFirstRadio; itemNo <= kLastRadio; itemNo++)
    {
      GetDialogItem(theDialog, itemNo, &itemType, &itemHandle, &itemRect);
      if (GetControlValue((ControlHandle)itemHandle))
        (**prefs).myRadio = itemNo;
    }

/* …and the checkbox… */
    GetDialogItem(theDialog, kCheckBox, &itemType, &itemHandle, &itemRect);
    (**prefs).myCheckBox = GetControlValue((ControlHandle)itemHandle);

/* Finally, copy all the key codes from the temporary storage. */
    for (itemNo = 0; itemNo <= kLastKey - kFirstKey; itemNo++)
      (**prefs).keyCodes[itemNo] = gTempKeyCodes[itemNo];

    returnValue = true; /* OK */
  };
```

That's it! Clean up and return.

```
  DisposeDialog(theDialog);
  SetPort(oldPort);
  return returnValue;
} /*AskKeys()*/
```

`InitPrefs()` is essentially the same as `CopyScores()`. See chapter 4 for more details.

```
/*InitPrefs()*/
/*This procedure loads the preferences from a resource in the current resource file,*/
/*creating a new one if there is none.*/

static void InitPrefs()
{
  short i;
  OSErr ignoreErr;

  gPrefs = (PrefsHnd)Get1Resource('Pref', 0);
  if ( gPrefs == nil ) /*Didn't exist -- create it!*/
    {
    gPrefs = (PrefsHnd)NewHandle(sizeof(PrefsRec));
    if ( gPrefs == nil )
      ExitToShell();                    /*Insert error message here*/

/* We should fill it with some values here */

    AddResource((Handle)gPrefs, 'Pref', 0, "\pPreferences");
    }
  else /*Did exist - check the size!*/
    if ( GetHandleSize((Handle)gPrefs) < sizeof(PrefsRec) )
      SetHandleSize((Handle)gPrefs, sizeof(PrefsRec));
}; /*InitScores()*/
```

Standard inits are in the source, but skipped here. Finally, here is the main program. It initializes and calls `AskKeys()`.

```
/****************** Main program ******************/

void main(void)
{
   Str255 str;
   short i;
   short gAppFile, gPrefFile;

   InitToolbox();

/*Get the prefs file -- create as necessary.*/
   if ( SetPrefFile("\pKeyConfig Preferences", '????', 'pref', &gAppFile, &gPrefFile))
      ;

/*Set the current resource file to the preference file before InitScores()*/
   if ( gPrefFile != 0 )
      UseResFile(gPrefFile);
   InitPrefs();
   UseResFile(gAppFile);

   if (AskKeys(gPrefs))
   {
      ChangedResource((Handle)gPrefs);
      WriteResource((Handle)gPrefs);
   }

   CloseResFile(gPrefFile);

} /*KeyConfig()*/
```

More Reading

So, have I told you everything about user interaction, user interface, and dialog management by now? Of course not! I could write a whole book on that subject only, but there's no need—Apple has done that already in *Inside Macintosh*! Unlike the information in this book's graphics chapters, most of the interaction, interface, and dialog information can be found in *IM*—if you are willing to wade through the volumes and search for what you need!

I haven't told you how to make lists with List Manager, nor how to make pop-up menus. I didn't tell you how to write a spreadsheet application either. What I did tell you was the basics, which is all you need for 90 percent of all games. If that is not quite enough for some need you have, either check out the source code on the CD-ROM or go to *Inside Macintosh*.

Profile: Colin Boswell

Colin Boswell, with Domark Software, has been involved with a number of games including Flying Nightmares (which was the first game developed for PowerPCs), Out of the Sun, and the upcoming Absolute Zero.

Q: What are your feelings on the Mac game market? Is it evolving? Has it changed? And any sort of design hints that you could provide?

A: The Mac market has changed quite radically, particularly with the introduction of the Power Macs. Now, Apple really has a machine that can compete with the PC clones on their own ground. The PowerPC processor is really nice to work with. Also, Apple has really good development tools, and there are good third-party tools—particularly Metrowerks CodeWarrior. That's one of the best tools I've ever used.

Q: I've heard developers say that the code that the CodeWarrior compiler generates isn't as optimized as the MPW compiler output. What you're talking about is quickly writing and implementing programs, and CodeWarrior has the advantage there.

A: Yes. In fact, what we do is develop all our stuff in CodeWarrior. We get the game working, and then the last couple of builds are actually done using the MPW tools. With the MPW optimizer, we can get the best speed out of game code. I'm not a great fan of MPW. It's getting a bit long in the tooth now and I like something a bit more user-friendly like CodeWarrior. As I said, we really just use MPW to get the final builds—which was kind of awkward the first time we did it. You find that once you've done one project, you start writing the next one and you have a knowledge base of things you should do and things you shouldn't do.

Q: What do you mean by that? Do these problems relate to C language syntax or something else?

A: It's more to do with the idiosyncratic ways that the compilers work. Things are actually getting easier now because the CodeWarrior tools have the full effect of universal headers that are identical in form to the ones Apple uses. It's actually a lot easier now than when we first did Flying Nightmares. At that time, the headers were actually quite different and that caused somewhat of a nightmare. The only other problem is that on CodeWarrior, we use precompiled headers to speed up compilation time. Unfortunately, MPW doesn't have that data privacy.

Q: You talked about Flying Nightmares. Could you tell me some of the design principles you used, and maybe some of the performance optimizations that you did?

continued

A: Flying Nightmares was unusual for us in quite a few ways. It was our first Mac title, and I had never written a Mac application at all. So what we did was take the existing IBM-PC game, and I did the port to the 68K Macs and the Power Macs. In fact, we were the first native Power Mac game. We were shipping when Apple introduced the Power Macs. Looking back, there are some things I would change. We got criticized—quite rightly—by having a game that looked very much like the original PC game. You know, Mac users have very definite ideas about what stuff should look like. Because Apple's people themselves have done a lot of work to make sure that things operated in a unified manner. Everybody knows that Command-Q is the Quit command. I don't think we actually realized at the time how much work is involved in putting the user interface on a Mac program. Because we've had so much success with the Apple game, our second product was a game called Out of the Sun, which was another Mac flight simulator. It was only on the Mac; we never actually wrote it for the PC. Our current game, Absolute Zero, we're actually codeveloping on both platforms at the same time. So I think we've actually learned quite a lot from that.

Q: Along those lines, what have you learned in terms of performance tricks?

A: Probably the one thing that every game developer would change on the Mac is that there is no double buffering to the video screen. Which basically means that you either have to draw directly to the screen, which you really can't do, because you get far too much flickering in certain applications, or you've got to have an off-screen buffer. An off-screen buffer becomes a bloody big bottleneck in the program, because you're copying chunks of memory to make something visible on-screen.

We found that actually there's several different ways that you can write a screen-copy algorithm. Some of these are more or less or efficient, depending upon the processor they run on. One of the things we found is that on the processors with larger caches, you have to be very careful. Traditionally, if you wanted to really speed up a screen-copy loop, you'd actually unwind the loop. That doesn't necessarily work too well on certain processors because you end up breaking the cache. In some cases, a simple loop can actually be a lot faster than actually unwinding the loop.

I think the best advice I could give is this: Think about where the critical memory-to-memory operations are, like the screen-copying algorithms in a 3-D game. Really sit down and analyze what's going on. It's worth getting a book on PowerPC assembly code and actually looking at the disassembly of the compiled code to see what's going on. Try some different approaches to see what generates the best assembly code.

Q: Let's talk about copying data to the screen. Do you use the Toolbox routine `CopyBits()`, or did you write your own?

A: We wrote our own. The code to `CopyBits()` is brilliant because it can do a million things. But that's also its problem. It doesn't matter how good the code is; Apple engineers really can't match code that's written only for a specific application. Even though our games are similar in appearance, they usually are unique because of the way we copy a big memory buffer to the screen. We actually revise that copy routine every time, because there is no such thing as perfect piece of code. But the great thing about `CopyBits()` is that it's there. If you get problems with your own copy routine, at least you know you've got something you can fall back on for debugging purposes. It's nice to not have to write everything right from the start.

Q: So you use the original `CopyBits()` routine to get the general game architecture operational and see stuff happening on the screen. Finally, it's time to do performance tuning, and then you insert your own copy routine algorithm to speed things up. If things blow apart, you know that's your routine at fault and not your game algorithms, because you saw those work already.

A: Yeah. It's real good to have something stable that you can fall back on.

Q: You mentioned that Flying Nightmares was a game port, and then Out of the Sun was exclusively Mac. What was the reason behind that?

A: Well, Flying Nightmares was really an experiment; we were just kind of dipping our toes in the water. We were so surprised by the success, it kind of took us in a wave, and we realized that we wanted to keep our profile in the Mac community. So, we took quite an unusual step for what was basically a new platform for us, deciding to take some of our existing technology and actually put a game on top of it. We have an in-house graphics engine and our own tools, so we basically got our engine up and running on the Macintosh. Then we decided that we'd like to do a World War II flight simulator. It was a hole we saw in the market. Brian Walker—he's an ex-Desert Storm pilot, and it's really his call—basically does what he thinks the market wants. And we got a lot of feedback from Flying Nightmares saying that the market wanted a World War II simulation. That was really the driving force behind it.

It was very much a market-driven product, as are most of the things we do. A lot of people tend to think game publishers just come up an idea and that's it. That's not really true. We only produce stuff that we think people actually want. There are lots of things I'd like to do, but that doesn't mean anybody else wants to see them. So what we try to do are things that we find really interesting, but that are also obviously going to make commercial sales. So, we did that with Out of the Sun for a World War II simulation.

Then we looked around again and saw another hole, which was for a really good science fiction kind of simulation. That's where Absolute Zero came from. And that's quite an unusual feat, because we're doing it on both platforms at the same time.

continued

Q: Briefly, what is Absolute Zero?

A: Absolute Zero is set in the 23rd century, on Europa, one of Jupiter's moons. We've colonized the solar system and humans are mining Jupiter's moons to get minerals. They're actually using Europa to supply water for fusion engines. What happens is that humans discover a race of aliens on the moon, and as is usually the case with most wars, there's a lot of misread thoughts and misread intentions, and it ends up as a giant war between the aliens and the Terrans. You take the part of seven different people for the game.

We call it sci-sim: It's sort of a cross between a flight simulator and a science fiction game. We've had loads of interest in it, and I'm really excited about it. It's unusual for me to be this excited about a game this far into its development. Usually, I just want to finish it.

Q: Along that line, you said are doing simultaneous development on two platforms. Do you use an identical code base—what's called core code—that's going to be common to the application on both platforms?

A: Basically, our simulation engine sets on very low-level routines which we call primitives. These are the tools that copy stuff onto the screen, generate buffers, and handle files. Everything above that we actually abstract. Once you have these tools (or primitives) running, then you can actually put the whole rest of the engine onto the machine. We have engineers on the project that work on whichever machine they like the most. I've got one engineer working on the PC, one working on the Mac, and I'm trying to switch between the two. As we come to the end of the project, we'll concentrate on finishing one version. We're actually doing the Macintosh version first. And at that point, we stop doing common code and start doing all the code a Macintosh application needs.

Q: Do you get the core code and then build a platform-specific wrapper—usually the user interface—around that? Is that what you mean when you say, "Mac-specific," or are there things in the game internals that you also would tweak for a particular platform?

A: No, logically speaking I think Mac-specific means the user-interface/human-factor design. Its "people controls." Mac users have got quite definite ideas about what people controls should be. Also, there are some small sacrifices you have to make on the Mac to keep the operating system happy. You have to write code for that. But it's a surprisingly small amount. In fact, I suspect that even Apple itself would be surprised.

Q: How much of the game would you say is Mac-specific versus core code?

A: Probably no more than about 100 lines of C.

Q: What's the total game size?

A: Well, our source code is approaching 2 MB. That's a lot of lines of code. It's astonishingly big, and perhaps to the point where no one person can actually remember what everything does in the game. This is one of the big problems in developing games these days. Projects are getting much bigger and harder to manage. But when it comes to user interface, we actually do an awful lot of work for the Macintosh; that's one of the things we've gone through over the last year and a half. We've actually learned a lot—we've learned what Mac users expect. And quite rightly so.

Q: We'll get back to a little bit of game architecture here. You said you have an imaging engine. Is that written all in C, or are there parts written in an assembler for performance?

A: It's currently all written in C, mainly for portability and debugging. We've been to a couple of Apple's games kitchens. We'll be going to another one soon, and I'm going to be talking to some of the Apple engineers about writing some assembler for the game. I'm weary of doing that, because one of the problems I mentioned earlier is writing code that sometimes works within one processor cache and sometimes floods a different processor's cache. Specifically, I wrote a screen copy routine that was an unfolded loop written in assembly language. It worked fine on a PowerPC 601's 32K cache, but it broke in PowerPC 603's cache because the cache is organized differently and is smaller. That's awkward, because you have to be very careful that you don't do something on one machine that's going to cause problems on another one. Apple has enough different machines out there as it is without having to write different code for every different configuration.

I think also that there's a certain amount of education going on at Apple itself, people learning the right way from the wrong way of doing things. We're trying to do things that nobody ever thought of when they designed the Power Mac, which is always the way with game developers. We do horrendous things.

It's interesting that not only Apple, but also PC hardware manufacturers are actually actively courting game developers now. That's because they've come to the conclusion that we've been telling them for years: Nobody buys a faster machine for the home just because it runs Excel faster. They buy a faster machine because they play games! Hardware manufacturers are starting to learn that game designers do push the envelope. We'd like to do a lot more, and I think if they want to demonstrate fast technology, there's nothing better for it, really.

I think even Intel has recognized that. They are selling more machines to home users than to any other market segment. And there is a big explosion going on. The explosion of the Internet is really because of home users. It's not really

continued

businesses. Businesses are pushing the expansion of the Internet purely as a way to get to consumers. So, I think everybody realizes that the next two years are probably going to be the most exciting time for consumer entertainment.

Q: Now, back to something you said earlier. The programs were getting bigger and the projects were getting bigger and harder to manage. Do you see opportunities for somebody in a small outfit to make a good game? Or are we reaching the point where larger outfits are going to be the only ones to write games? Do you think there's a niche for the little guy who writes a shareware game?

A: I think it's still possible for the little guy. Let's not forget that DOOM was shareware. That was basically two guys sitting in their house writing it. It wasn't until they took it to major publishers that it was a success. The thing is, creativity is everything. We're in an industry that thrives on new ideas, and the person with the new idea is the person that's going to get ahead. From that point of view, a single person can write code and be just as creative—often more creative—than a group of people. So I think that's an advantage.

The disadvantage is that to have a really competitive product, there's a certain quality you have to achieve these days. I think the industry as a whole has lost something because they're all so desperate to put rendered introductions into their games. A lot of people have forgotten that fundamental thing, which is that the game has to be fun. At the end of the day, people play a game because they want fun, not because they want to see a ten-minute rendered intro that gets stale after the first time they watch it.

So I think that there is still an opportunity for an individual. As far as I'm concerned, a good program is a good program, no matter how many people wrote it.

Q: Suppose someone writes a game. It could be a small team, a big team, or the guy in his study at night. What would they do to play-test it? You mentioned the example of the programmer having this long introduction in the game. So he play-tests it with some friends. Or if it's shareware, he gets email really quick that says, "Get that long intro out of my face, I just want to start playing!" So how would you go about play testing? Can play testing point out some faults in the game rules?

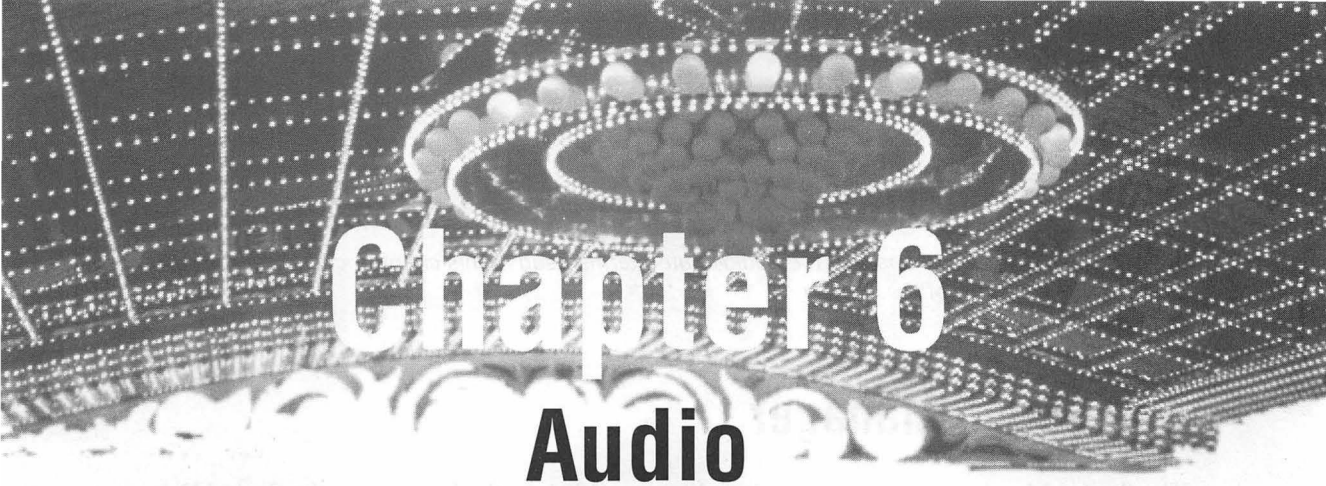
A: If you have a lot of friends, get them to test it. There are some things you have to accept if they criticize it. They criticize it because of something in the game they don't like, not because it's personal. The reason I say that is every programmer who's working on the project gets very close to it. And you get very attached to your own ideas. That's why programmers always want people to test their products. You have to accept people's judgment. If some people say they don't like it for a very good reason, then they're probably right.

We actually have our own in-house testers, and we use professional testers as well. If you think you're good enough and you want to publish it, a good publisher will actually test the game for you. And even if he's not interested, if the game is good is enough, the publisher will say, "We're not interested, but we think these elements are good, and these are bad." I think that's something you can do as well.

The other thing is, you have to be kind of humble, but at the same time have confidence in your own product. You've got to have enough confidence to say this is good enough for other people to see. But on the other hand, if you give it to 100 people and 99 of them say this is awful, then you really have to accept that you will have to do some work to change it. And it's not always easy. I've had a couple of run-ins over the last week with testers who've made comments, and I said, "Oh no, that's rubbish," but then I've gone away and thought about it and decided they were right. I've thought that this piece of code probably isn't great, and I've rewritten it and given it back to them, and they've said, "This is fine, it works better."

Q: Could you tell me what exactly that was? Was it just how the game was played or...

A: Yeah. It was really a people controls issue about the keyboard layout. I had chosen a particular combination of keys, a couple of functions that happen to suit me, and one of the things was Command-W. Of course, W is right next to Q on the keyboard, so people were accidentally quitting the game. These are really minor things, but they are the things that make a difference between a good game and a great game.



Chapter 6
Audio

A typical game has the following audio elements which bring style, excitement, and energy to your game:

- *Sound effects*—This is the simplest thing to put into a game, but very important. The sound effects define the quality of the audio experience, because they are the most present to the user. Typically, sound effects are tied directly to the player's actions. If the gunshot is a weak and wimpy sound, do you think the player is going to feel powerful in the game? I want to mention one game that I think is successful in part because of the sound effects: Flashback. This game is about total immersion into an environment. Graphics, audio, and overall feel are very important in creating the illusion. The sound effects speak to the players in such a way that they believe they have a huge gun on a distant planet.
- *Theme music and background music*—There are two kinds of music in games today: throwaway music and experience music. Throwaway music is title music, win condition music, and anything with a static screen attached. Games such as Lode Runner: The Legend Returns have some throwaway music, but since it's followed up with good background sound and consistent sound effects, the overall audio works well to create the total experience. Experience music is background music used to enhance the actual game play. Often, experience music varies with the situation or action on the screen.
- *Voice tracks*—Finally, with CD-ROM technology, there's now enough data space to include this aspect of audio. Voice characterizations with animation are all the rage now. Voice adds so much to the definition of a character. Can you imagine Disney's *Aladdin* without Robin Williams? I think not. We're going to see voice talent defining a product's overall quality more and more.

In this chapter, we are going to present the Sound Manager in a practical light for use in game programming on the Macintosh. We'll show the data formats and file formats that the Sound Manager uses. We'll show how to use the QuickTime Music architecture. We'll get deep into some of the technical problems, show solutions to some of the issues for creating sound, and finally present an alternative interface, called Hollywood, for use with the Sound Manager. Hollywood makes sound easy to understand and use.

Note
For quick definitions of some of the audio-related terms used in this chapter, check out the Glossary at the end of the chapter.

The Sound Manager

The Sound Manager has evolved quite a bit since 1984. In the beginning, the Sound Driver had limited use, but it provided something that no one had seen in the game arena. The Mac was the first personal computer to use 8-bit sampled data for playback, but surprisingly nobody used it. No wonder, though: It was difficult to use and buggy. With the introduction of Sound Manager 3.0, Apple has finally delivered upon its promise for true multimedia audio.

The Sound Manager was designed as a low-level interface between the application programmer and the Macintosh hardware. With Apple's design, as with all application programmer interfaces (APIs), there are inherent missing elements. Rightfully so, Apple leaves all the details up to the picky programmers and sticks to implementing the hard technology—something Apple does well. The Sound Manager has no memory management functions, no simple playback of sound effects asynchronously, and no sound format support. Yet it defines file formats, sound formats, and a basic API.

A simple solution is to create a layer between the application programmer and the Sound Manager. In its feature set, this new layer should have memory management, sound format support, one-shot sound effects support, and basic streaming audio support.

Features of Sound Manager 3.0

The typical game developer's response is to bypass all operating system features and go straight to the metal, or the hardware. The Sound Manager, as of version 3.0, is powerful, feature-full, and finally fast enough to use. The Sound Manager has a variety of features, but it's not entirely easy to understand. For simplicity, we'll talk only about the features that the game industry has used effectively and that you may need for your games. We will explore some of the basic and most useful aspects of the Sound Manager. The Sound Input Manager is not covered here, although we'll utilize some of its functions. Some of the basic features include setting up and disposing of channels, playing back one-shot sound effects, and controlling the sound during playback.

Sound Manager 101

The basic Sound Manager application interface (API) works quite simply. Set up a channel, issue commands to that specific channel to play samples, and process callbacks. When you're done with the channel, you dispose of it. The Sound Manager does no memory management, which causes some confusion, but it's low-level enough to do the job without getting in the way.

You need to include these header files for most of the examples here to work.

```
#include <Sound.h>
#include <SoundInput.h>
```

Most of the routines in this chapter work for any version of the Sound Manager. But some require Sound Manager 3.0 or above. One of the first things we need to determine is whether or not Sound Manager 3.0 is installed. The following code will return TRUE if Sound Manager 3.0 is installed and FALSE if the Mac is running an earlier version.

```
static Boolean HYP_IsNewSoundManagerInstalled(void)
{
   NumVersion   theVersion;
   void *  trap1;
   void *  trap2;
   Boolean installed;

   installed = FALSE;
   // SndSoundManagerVersion() Trap
   trap1 = (void *)GetToolTrapAddress(_SoundDispatch);
   // Unimplemented Trap
   trap2 = (void *)GetToolTrapAddress(_Unimplemented);
   if (trap1 != trap2)
   {
      theVersion = SndSoundManagerVersion();
      if (theVersion.majorRev >= 3)
      {
         installed = TRUE;
      }
   }
   return installed;
}
```



This code works by checking for the `SoundDispatch` trap and then using the `SndSoundManagerVersion()` function to determine the version number for the Sound Manager. If `theVersion.majorRev` is equal to or greater than 3, this routine returns TRUE, so programs that use this routine are ready for Apple's future Sound Manager revisions.

Data Types: 8-Bit, 16-Bit, Mono, and Stereo

The Sound Manager can play a variety of sound types, the most important of which are 8-bit data and 16-bit data (see figure 6.1). Positive values represent the speaker in a outward motion. Negative values represent the speaker in an inward motion, and zero is the speaker at rest.

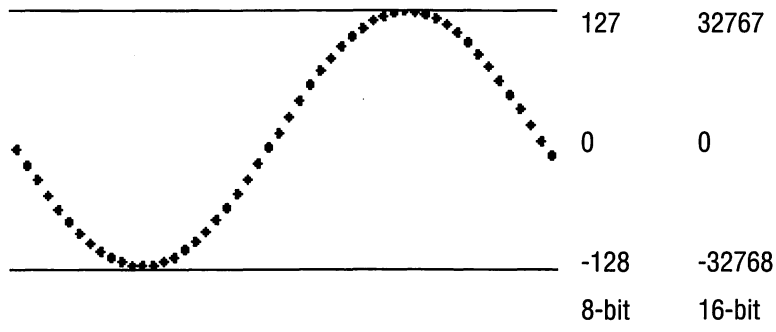


Figure 6.1 *Digital audio sine wave.*

A volume sample is one value of either 8-bit or 16-bit in size that represents the speaker position at a particular point in time. The bit size has the greatest impact on the number of volume levels or speaker positions that can be represented in the sound. For example, 8-bit data can have a volume range of only 256 levels, whereas 16-bit data can have a range of 65,536! So, 16-bit data gives better sound quality than 8-bit data, but at the cost of CPU performance.

The sample rate is the number of volume samples per second in the sound.

An audio compact disc (generally known as a CD) has data encoded in 16-bit data, with 44,100 samples per second for two channels of audio for stereo playback. So in the case of a CD, the total size in bytes for one minute of sound is:

44,100 (samples per second) * 60 (to convert to minutes) * 2 (16-bit data is 2 bytes) * 2 (two channels of data for stereo) * 1 (the number of minutes) = Total of 10,584,000 or 10 megabytes of data!

Eight-bit data will be exactly half that size, but the quality is quite a bit less. Mono sound or one-channel of data is half the size again. So in our example, 8-bit data mono is 2,646,000 or 2 megabytes of data—still large, but more manageable, and that's for only one minute of sound.

Most sound on the Macintosh is played at the rate of 22,254 samples per second and is 8-bit data, which cuts the data size in half again—or nearly so. 22,254 is not exactly half of 44,100, which raises issues we will discuss in the "Advanced Sound Manager" section later in this chapter.

Multichannel Sound

The Sound Manager can support more than one channel of sample data. The actual number depends upon the performance of the Macintosh on which your application is running. Typically it's safe to assume four channels are available.

All you need to do to have multichannel sound is allocate more than one sound channel. The only drawback to this approach is that the Sound Manager will dynamically alter the overall volume levels of all the channels currently playing. The Sound Manager will play any sound at full volume if there is only one sound playing. If there are two sound channels playing, then each sound will be at half of its respective volume level; if there are three, then one-third of its volume level, and so on. This works well for most cases, and there is no easy way to tell the Sound Manager not to do it.

For example, let's say you allocate two sound channels: one for a constant background sound track, and the other for sound effects that happen once in awhile. Suppose the sound track is playing along and there is no other sound playing. The Sound Manager will allocate all of its dynamic volume range to just the one track. Then a sound effect occurs. Now, unlike Spinal Tap's amplifiers, the Macintosh sound hardware can't go to 11; full volume is as high as it can get. If the Sound Manager simply added the sound effect to the sound track, the two sounds would add up to more than full volume, the peak sounds would get digitally clipped, and the resulting sound would be like a spike or loud pop. So the Sound Manager, being the good guy that it is, cuts the volume of the sound track in half, then plays the sound effect at half of its full volume. Then, after the sound effect is finished playing, the Sound Manager will jump the sound track volume back up to full.

Channel Setup and Disposal

In order to play sound through the Sound Manager, you have to first allocate a channel. This channel is a reference to a voice that will be mixed into the final output buffer that the Sound Manager sends to the hardware. Once you are done with a channel, you must dispose of it, which will release all of the associated memory.

`SndNewChannel()`

The Sound Manager has three basic synthesizers. When a call to `SndNewChannel()` is made, a particular synthesizer must be passed along to initialize the channel.

```
squareWaveSynth   = 1  // Square wave synthesizer
waveTableSynth    = 3  // Wave table synthesizer
sampledSynth      = 5  // Sampled sound synthesizer
```

`squareWaveSynth` and `waveTableSynth` are synthesizers that set up this channel for playback of simple square wave sound and sampled instrument playback. They're very limited, and for most uses somewhat outdated. Their main use is to play back musical notes. A much better solution is QuickTime 2.0's music synthesizer.



`sampledSynth` sets up the channel to play back sample data. This is the mode we'll use all the time, because it suits our needs as game developers.

A mode must also be passed along to `SndNewChannel()` to set up the initial feature set of this channel.

```
initChanLeft    = 0x0002  // Left stereo channel
initChanRight   = 0x0003  // Right stereo channel
initNoInterp    = 0x0004  // No linear interpolation
initNoDrop      = 0x0008  // No drop-sample conversion
initMono        = 0x0080  // Monophonic channel
initStereo      = 0x00C0  // Stereo channel
initMACE3       = 0x0300  // MACE 3:1 compression
initMACE6       = 0x0400  // MACE 6:1 compression
```

`initMono` sets the channel to handle mono data only. If stereo data is passed, it is converted into mono on the fly.

`initStereo` sets the channel to handle stereo and mono data. On most Macintoshes today, this is the default setting. Be careful, because without external speakers, the sound on the right channel may not play back at all.

`initChanLeft` sets up the channel to play all sound through this channel into the left output of the Macintosh hardware.

`initChanRight` sets up the channel to play all sound through this channel into the right output of the Macintosh hardware, and the same problem for stereo applies here. Some Macintosh hardware has only the left speaker wired.

When passed samples that need to be converted to different sample rates, the Sound Manager has two methods available to handle this problem: drop sampling and linear interpolation.

For example, with drop sampling, if you have recorded a sound effect at 22.254 KHz and play it back at 11.127 KHz, the Sound Manager will remove every other sample. If you were to play a sample that was recorded at 11.127 KHz at 22.254 KHz, then every other sample would be duplicated. This technique is very effective in converting a sample into the new sample rate, but it creates the side effect of a noisy playback. This is most apparent on the new Macintosh hardware such as the AVs and Power Macs. In these cases, low-level hardware output rates differ slightly from the natural sampling rates of most sound effects for the Macintosh.

With linear interpolation, in our example every other sample would be calculated as the point between the previous and next sample. So for most cases, this new calculated value will be closer to the original sample that needs to be replaced. But obviously this takes time.

If `initNoDrop` is set, then the Sound Manager no longer processes samples this way, but uses a method called linear interpolation that determines the in-between value between two values for the sample rate conversion process. This takes more CPU time but produces a better-quality playback.

The flags `initNoDrop` and `initNoInterp` used to control these features of the Sound Manager are mutually exclusive. Passing `initNoDrop` enables linear interpolation and disables drop sampling. Passing `initNoInterp` disables linear interpolation and enables drop sampling. If you pass both `initNoDrop` and `initNoInterp`, the Sound Manager does just basic drop sampling, but it's undefined by Apple and may change. If neither flag is passed, then linear interpolation is enabled, but this is also undefined by Apple and may change.

`initMACE3` and `initMACE6` are never used as initial values for a new sound channel that is used for playback. These are used when compressing sound data into the MACE format. You set up a channel, compress a sound with that channel, then dispose of the channel.

```
OSErr SndNewChannel(SndChannelPtr *theChannelPointer,
          short synthType,
          long initMode,
          SndCallBackUPP callbackProc);
```

`theChannelPointer` is a pointer to a `SndChannelPtr`. A valid sound channel is returned. You can pass this to all of the Sound Manager's functions.

`synthType` is the synthesizer type to be allocated for this channel.

`initMode` defines which set of modes are to be used when allocating this channel.

`callbackProc` is the callback procedure that is called when a `callBack` command is issued. The function prototype for a callback looks like this:

```
pascal void SM_CallBack(SndChannelPtr pChannel,
          SndCommand *pCmd);
```

`callbackProc` is called at interrupt time. The `pChannel` parameter contains the channel the callback was issued upon, and `pCmd` contains the command.

The setup call is quite simple.

```
SndChannelPtr  newChannel; // This needs to be global

void SetupChannel(void)
{
   theErr = SndNewChannel(&newChannel,
              sampledSynth,
              initMode,
              NULL);
}
```



The `newChannel` parameter is an allocated or static memory object of type `SndChannelPtr`. You can pass NULL, but that forces the channel to play all sound through this channel synchronously—or stop all other action and play this sound now. For games, we need asynchronous sound.

To do asynchronous sound playback, you always need to allocate a sound channel. By passing in a `SndChannelPtr` as `SndNewChannel()`'s first parameter, the Sound Manager will allocate the sound channel and place that into the pointer passed.

One of the most effective techniques is to allocate a sound channel at your application's startup time and reuse that channel for every sound that you make. The only drawback here is that you need to deallocate and reallocate it when your application needs to be put into the background. With Sound Manager 3.0, there is no longer the problem of supporting only one open sound channel at a time, but you still have a limit. It's effective memory management to disconnect yourself from the hardware when you don't need it.

The second parameter controls the type of synthesizer that we want. For our purposes, the `sampledSynth` is all we need. We always want to send sampled 8, 16, mono, or stereo samples to the sound channel.

The `initMode` parameter determines the type of mode in which we want the sound channel to work. The most effective value here is `initMono`. You can always change this later, and the Sound Manager is smart about how it plays sound. If the sound to be played does not match the channel's mode, it will be processed to match the initial channel mode. The modes are additive, hence `(initMono + initNoDrop)` would allocate a mono channel with the "no drop samples" feature turned on.

The last parameter is the callback function. This function is called only when a `callbackCmd` is issued to this sound channel. See the "Advanced Sound Manager" section for a complete example.

`SndDisposeChannel()`

Once we're done with a channel, we need to release it and free up the memory associated with it.

```
OSErr SndDisposeChannel(SndChannelPtr newChannel);
```

`newChannel` is a valid channel that was allocated with `SndNewChannel()`.

The `newChannel` parameter is our channel pointer that was passed into `SndNewChannel()`. It's always a good idea to release all channels when your application is no longer in front. With the advent of Sound Manager 3.0, releasing channels isn't absolutely necessary anymore, but it's still good practice.

Making Noise and Silence

Now that a channel has been set up, we're ready to play sound. There are a few ways to play sound with the Sound Manager. You can use the command `bufferCmd` with the functions `SndDoCommand()` or `SndDoImmediate()`, you can use `SndPlay()`, or you can use `SndStartFilePlay()`.

The Sound Manager uses a method of queuing up commands for playback. For example, to play a sound to a channel, you issue a `bufferCmd`. To change the volume of a sound currently playing, you issue a `volumeCmd`. All Sound Manager commands sent with the function `SndDoCommand()` are put into a queue to be processed when the Sound Manager is done with the last command. You use `SndDoImmediate()` to force the Sound Manager to do the command right now. `SndDoImmediate()` does not affect what is currently in the Sound Manager's queue, unless it is passing the Sound Manager a flush command.

`SndDoCommand()` and `SndDoImmediate()`

The functions `SndDoCommand()` and `SndDoImmediate()` take a `SndCommand` structure as a parameter. Each command has two parameters that are passed along with the command.

```
OSErr SndDoCommand(SndChannelPtr newChannel,
          SndCommand *pCommand,
          Boolean dontWait);

OSErr SndDoImmediate(SndChannelPtr newChannel,
          SndCommand *pCommand);
```

`newChannel` is a valid channel that was allocated with `SndNewChannel()`.

`pCommand` is the command, with two parameters, that is to be processed to this channel.

`dontWait` ensures that when queuing up commands with `SndDoCommand()`, passing TRUE will allow `SndDoCommand()` to return without making sure that the new command is put into the queue. By passing FALSE, you are guaranteed that the new command is put into the queue.

The `pCommand` variable may be static or local data. The only difference between these two functions is that `SndDoCommand()` queues up the commands in the channel for later processing and `SndDoImmediate()` completes the command immediately if possible.

```
{
  SndCommand theCmd;

  theCmd.param1 = 0;
  theCmd.param2 = 0;
  theCmd.cmd = myCommand;
}
```



There are a couple of dozen different sound commands that can be passed to the sound queue. Most commands are no longer useful, and there are better ways of doing their tasks with Sound Manager 3.0. Here is an example of a command that is useful and its parameters.

`bufferCmd` plays a buffer of sound data. The `param2` value is a pointer to a sound header structure such as `SoundHeader`, `CmpSoundHeader`, or `ExtSoundHeader`.

With Sound Manager 3.0, the `ExtSoundHeader` structure is the best one to use to gain access to all the different data types such as 8-bit, 16-bit, mono and stereo data.

If the high bit of the `bufferCmd` is enabled, `param2` will be an offset into the 'snd ' resource to the sound header structure. For example, in a resource, the sound header structure might be at offset 20 from the beginning, so `param2` would be 20 and the command would be `0x8000 + bufferCmd`. This is how 'snd ' resources are stored.

For example, here's how `bufferCmd` plays a sample using the extended sound header structure:

```
{
   ExtSoundHeader theSndBuffer;
   SndCommand   theCmd;

   theSndBuffer.samplePtr = (Ptr)pSample;
   theSndBuffer.numChannels = channelSize;
   theSndBuffer.sampleRate = sampleRate;
   theSndBuffer.encode = extSH;
   theSndBuffer.numFrames = length;
   theSndBuffer.sampleSize = dataBitSize;
   theCmd.param1 = 0;
   theCmd.param2 = (long)&theSndBuffer;
   theCmd.cmd = bufferCmd;
   theErr = SndDoCommand(newChannel, &theCmd, FALSE);
}
```

The variable `channelSize` is 1 for mono or 2 for stereo data.

The `sampleRate` is an unsigned fixed value. The value 22.254 KHz is represented as 22254.54545 in fixed point math, which would be 0x56EE8BA3.

The variable `length` is the number of frames of sample data that is pointed to by the variable `pSample`. This is not the number of bytes. For example, a mono 8-bit sample with 10,000 frames would be 10,000 bytes long. The same data in stereo would still be 10,000 frames, but its byte length would be 20,000 because we need to encode 2 bytes per frame for each channel of data.

Audio Frame	Frame Length in Bytes
8-bit mono	1
8-bit stereo	2
16-bit mono	2
16-bit stereo	4

Each audio frame contains a sample. This sample can be a different byte size depending upon if it's an 8-bit or 16-bit sample, mono or stereo. In the worst case, the 16-bit stereo sample would be 4 bytes in size, so each audio frame would be 4 bytes.

Creating Silence

To stop sound, you must issue a quiet command and a flush command to stop the sound and clean up the queue.

`quietCmd` and `flushCmd` quiet the current sound playing in the channel and flush out the queue for the channel. Send these commands with the `SndDoImmediate()` command. Both parameters are unused.

```
{
   SndCommand   theCmd;

   theCmd.param1 = 0;
   theCmd.param2 = 0;
   theCmd.cmd = quietCmd;
   SndDoImmediate(newChannel, &theCmd);
   theCmd.cmd = flushCmd;
   SndDoImmediate(newChannel, &theCmd);
}
```

The order is important here. First, we want to shut down any sound that is playing, and then clear the queue of commands that the Sound Manager may be processing. You need to perform both of these commands to silence the channel.

`SndPlay()`

`SndPlay()` is the simplest way to play a sound. However, it has some limitations.

```
OSErr SndPlay(SndChannelPtr newChannel,
       Handle soundHandle,
       Boolean doAsync);
```



`newChannel` is a valid channel that was allocated with `SndNewChannel()`.

`soundHandle` is a valid 'snd ' resource of either type 1 or type 2 format.

If `doAsync` is TRUE and `newChannel` is not NULL, then `SndPlay()` will return immediately and the sound will play in the background.

For `SndPlay()` to be most effective, you must allocate a sound channel. You can pass NULL for the variable `newChannel`, but the function does not return until the sound is done playing. The variable `soundHandle` is a 'snd ' resource or handle that can be any of the standard types of sound objects: 8-bit, 16-bit, mono or stereo, compressed or not. When using `soundHandle` with a sound channel, you must pass TRUE for the variable `doAsync`. If `newChannel` is NULL, then `doAsync` is ignored.

```
{
  Handle  theSnd;

  theSnd = GetResource('snd ', 128);
  if (theSnd)
  {
    HLock(theSnd);
    SndPlay(newChannel, theSnd, TRUE);
  }
}
```

Given a sound resource, these lines of code will play the sound asynchronously. If you release the resource before it's done playing, you might hear static. You need to lock the resource down; the Sound Manager will not do that for you. In this example, if you were to unlock the resource before it was done playing and the memory manager moved that data, you might hear static.

Is the Sound Finished Playing?

The simplest way to determine if there is any sound playing from a particular channel is to ask the Sound Manager for status on that channel.

```
OSErr SndChannelStatus(SndChannelPtr newChannel,
            short theLength,
            SCStatusPtr pStatus)
```

`newChannel` is a valid channel that was allocated with `SndNewChannel()`.

`theLength` is the length of a `SCStatus` structure. Passing `sizeof(SCStatus)` is expected.

`pStatus` is a pointer to a `SCStatus` structure. It is filled with information about the current channel. Depending on what the channel is doing at that moment, certain parameters in the structure will be valid. Not all will be valid at the time of the call.

This method will return to you whether or not that channel is currently playing a sound:

```
{
   SCStatus  status;
   OSErr   theErr;
   Boolean busy;

   busy = FALSE;
   theErr = SndChannelStatus(newChannel, sizeof(SCStatus),
                  &status);
   if (theErr == noErr)
   {
      busy = status.scChannelBusy;
   }
   if (busy)
   {
      // then there is still sound playing in the channel
   }
   return busy;
}
```

You can poll the Sound Manager with `SndChannelStatus()` at interrupt time, or at `GetNextEvent()` time to determine when a sound is finished.

`SndStartFilePlay()`, `SndPauseFilePlay()`, and `SndStopFilePlay()`

These calls provide the simplest way to play back large sounds through the Sound Manager. You can play either files or large 'snd ' resources.

```
OSErr SndStartFilePlay(SndChannelPtr newChannel,
          short fileRefNum,
          short resourceID,
          long bufferSize,
          void *pBuffer,
          AudioSelectionPtr theSelection,
          FilePlayCompletionUPP pCallbackProc,
          Boolean doAsync)
```

`newChannel` is a valid channel that was allocated with `SndNewChannel()`.

`fileRefNum` is a valid file reference number obtained from `FSOpen`, `HOpen`, `OpenDF`, `HOpenDF`, or `FSpOpenDF`, identifying an AIFF- or AIFF-C-formatted file. You need to pass this parameter only when playing a file; otherwise, pass 0 to play from a resource.

`resourceID` is a resource ID number of a 'snd ' resource in the open resource file chain. If you're playing from a file, then pass 0 for the resource ID.



For `bufferSize`, a minimum of 20,480 is recommended to handle to CPU load on all Macintoshes. If you pass 0, the default size will be allocated if `pBuffer` is NULL.

`pBuffer` should be a pointer allocated with `NewPtr()` of `bufferSize` bytes. If you pass NULL, then the Sound Manager will allocate its own internal buffer for use during playback.

`theSelection` is a pointer to an `AudioSelection` structure that allows partial playback of the sound. If you wish to play the entire sound, pass NULL.

When the sound is finished playing or is stopped by `SndStopFilePlay()`, the function passed through the variable `pCallbackProc` is called. It's only used if `doAsync` is TRUE. The callback function is defined as:

```
pascal void S_FileDone(SndChannelPtr pChannel);
```

This callback function will be called at interrupt time. The `pChannel` parameter contains the channel on which the file play was started.

If `doAsync` is TRUE and `newChannel` is a valid sound channel, then `SndStartFilePlay()` starts the playback in the background and returns. If FALSE, then `SndStartFilePlay()` plays the sound and doesn't return until the sound is finished playing.

You must first open the data fork or resource fork of a file. If you are playing a file, then the file reference number is passed as `fileRefNum` and `resourceID` parameter is set to 0. If you are playing from a resource, then the `fileRefNum` is set to 0 and `resourceID` is set to the 'snd ' resource number that you want to play.

If you pass NULL with the `pBuffer` parameter, `SndStartFilePlay()` allocates a block of memory that is `bufferSize` in length. `SndStartFilePlay()` uses `bufferSize` to prebuffer when reading data into memory. This is helpful on slow devices such as CD-ROM drives.

If you're using files, they must be formatted in either AIFF or AIFF-C. This is a standard sound file format defined by Apple, and most applications that work with sound can save in this format.

If you're using 'snd ' resources, they must be the standard type 1 or 2 sound resources. The Sound Manager handles all the memory management and playback of audio. All you have to do is open and close the file or resource file associated with the sound you want to play.

Be careful how you use `SndStartFilePlay()`. The Sound Manager can get very confused if you were to start playing a resource from a particular open resource file and try to write to the same open resource file at the same time. Likewise, playing the same resource from the same open resource file, even with a different sound channel, will cause the Sound Manager to lock up and crash sometimes. It would be better to convert the resources that you want to stream through `SndStartFilePlay()` to AIFF files.

You can start a sound playing with `SndStartFilePlay()` from a point inside of the file by specifying an Audio Selection. You can specify only seconds, but the selection points are in fixed point. For example, 1.5 seconds from start would be 0x00018000. An even simpler method, but one that's slower, is the fixed point math Toolbox routines.

```
#define unitTypeSeconds  0x0000

struct AudioSelection
{
   long      unitType;
   UnsignedFixed  selStart;
   UnsignedFixed  selEnd;
};
```

The following code snippet calculates 1.5 seconds from the start and 4.75 seconds from the end:

```
{
   AudioSelection theSelection;

   theSelection.unitType = unitTypeSeconds;
   // Convert 1.5 into fixed point
   theSelection.selStart = X2Fix(1.5);
   // Convert 4.75 into fixed point
   theSelection.selEnd = X2Fix(4.75);
}
```

Once the playback starts, you can pause and resume the audio with this call:

```
OSErr SndPauseFilePlay(SndChannelPtr newChannel);
```

`newChannel` is a valid channel that was allocated with `SndNewChannel()`.

Once audio is paused, you must call `SndPauseFilePlay()` again to resume playback. You can determine if the channel is paused already by checking the status of the channel:

```
Boolean IsSoundChannelPaused(SndChannelPtr newChannel)
{
   SCStatus  status;
   OSErr   theErr;
   Boolean paused;

   paused = FALSE;
   theErr = SndChannelStatus(newChannel, sizeof(SCStatus),
                  &status);
   if (theErr == noErr)
   {
      paused = status.scChannelPaused;
   }
   return paused;
}
```



If you need to stop audio all together, you can call `SndStopFilePlay()`:

```
OSErr SndStopFilePlay(SndChannelPtr newChannel,
            Boolean quietNow);
```

`newChannel` is a valid channel that was allocated with `SndNewChannel()`.

The `quietNow`, if TRUE, stops all audio playback and returns immediately. If FALSE, `SndStopFilePlay()` does not return until the audio has finished playback naturally.

If a callback procedure is passed to `SndStartFilePlay()`, it is called by `SndStopFilePlay()`.

If you allow the Sound Manager to allocate its own buffers when calling `SndStartFilePlay()`, the buffers are released after calling `SndStopFilePlay()`.

Advanced Sound Manager

Now we've reached the fun part. The Sound Manager is not hard to use if all you want to do is play a simple beep sound once in a while. But as with most things Macintosh, you always want to do something "extra." Here's where you'll find out how to determine which sample rates you need, set up an interrupt task, use sound callbacks, change the volume and rate of a currently playing sound, and get information about sound formats.

Sample Rates: 11,025/22,050 vs. 11,127/22,254

There are two standards in the world of sampling rates: the Macintosh and the rest of the world. The slogan "The Computer for the rest of us" has a different meaning now.

The natural sampling rate is the rate at which the hardware can easily read data and output it to the speaker. Processing at 22,254 times a second, the Macintosh hardware gets samples and inputs them to the speaker. In 1984, the standard rate of 22,254 KHz was historically linked to the way the original Macintosh hardware processed its audio and floppy disk data. One thing about hardware design is that you commit to the design early. Apple most likely decided upon the final technology in 1981 or 1982.

But then the audio compact disc changed the world and inexpensive digital audio converters (DACs) were available, and the master sampling rate was 44,100 KHz. The values 22,050 and 11,025 are divisors of 44,100, and the IBM PC clone sound board market adapted the inexpensive DAC chips. Apple continued to use and develop custom audio chips to support its sampling rate. But the race to be competitive in price put Apple in a new direction—to the better.

Apple changed its hardware design to the new frequency of the audio compact disc, which is 44,100 KHz, with the introduction of the 660AV and 840AV machines and every PowerPC-equipped Macintosh (and all the clones). Every machine built after these will use the new and inexpensive chips for audio output.

The only side effect of all of this change is that all the sound sampled at the old Macintosh natural frequency is flat or sounds "off" on the new hardware. That's because the Sound Manager has to resample data that was passed to it by just a small percentage—just over one percent difference. But to some human ears, that's enough to cause panic.

To remedy the problem, you can resample all of your data to the new natural frequency or let the Sound Manager resample it. If you let the Sound Manager resample, it will take CPU time, but the Sound Manager does a fair job because it's geared for speed rather than quality.

Compression and Expansion with MACE

Audio compression is not like your normal everyday data compression. It's almost an art form. There are two types of audio compression: lossless and lossy.

Lossless means that the audio data that is compressed is exactly the same when uncompressed, bit for bit. This is the purest form of compression—no change in the data. Playback is exactly the same.

Lossy means that the audio data will lose some information during compression. This typically causes the playback of the data to sound scratchy or distorted. Sometimes, though, when you need the space, a lossy compression is the way to go.

Apple has provided lossy compression with the Macintosh Audio Compression and Expansion (MACE) algorithms. MACE is always present with Sound Manager 3.0 or better. You can encode MACE sounds as 'snd ' resources and the Sound Manager will automatically expand them during playback, or if needed, you can expand them yourself.

Most audio authoring tools provide a way to save MACE sounds, but sometimes you just need to do it yourself.

To compress audio with MACE, you first need the raw audio data. MACE will compress multichannel sound, but it's not a straightforward process.

```
void Comp3to1(void *inBuffer, void *outBuffer,
        unsigned long length,
        void *inState, void *outState,
        unsigned long numChannels,
        unsigned long whichChannel);

void Comp6to1(void *inBuffer, void *outBuffer,
        unsigned long length,
        void *inState, void *outState,
        unsigned long numChannels,
        unsigned long whichChannel);

void Exp1to3(void *inBuffer, void *outBuffer,
        unsigned long length,
        void *inState, void *outState,
        unsigned long numChannels,
        unsigned long whichChannel);

void Exp1to6(void *inBuffer, void *outBuffer,
        unsigned long length,
        void *inState, void *outState,
        unsigned long numChannels,
        unsigned long whichChannel);
```



`inBuffer` is the sample data to compress or expand. It may be 8- or 16-bit data, mono or stereo. You may pass NULL for either the `inBuffer` or the `outBuffer` to have the MACE tools allocate the state buffer for you.

`outBuffer` is the compressed or expanded stream of data.

`length` is the length of the data to compress or its current length if expanding.

`inState` is a pointer to a 128 byte buffer. You can pass NULL to tell MACE to allocate the buffer itself.

`outState` is a pointer to a 128 byte buffer. You can pass NULL to tell MACE to allocate the buffer itself.

`numChannels` is the number of channels for the data to compress or expand.

`whichChannel` tells which channel of data MACE is compressing.

MACE does not compress multichannel data. It compresses only one channel at a time. To compress multichannel data, you need to compress each channel separately, then reinterleave the sample data for playback.

Callbacks

A callback is an action that occurs when the Sound Manager is done playing a sound and it calls you to inform you of that fact. When you allocate a sound channel with `SndNewChannel()`, one of the parameters is a callback function. An example of this function for 68K and PowerPC-based Macintoshes looks like this:

```
pascal void HandleSoundDoneCallBack(SndChannelPtr pChannel,
                    SndCommand *pCmd)
{
   register long            saveA5;

   if (pCmd->param1 == kSoundManagerID)
   {
#if GENERATING68K == 1
      // restore previous A5 for 68K-based CPUs
      saveA5 = SetA5(pChannel->userInfo);
#else
      // don't do anything for PowerPC
      saveA5;
#endif

      // Do something interesting during a sound interrupt
      // Set flags, start more sound, and twiddle bits

#if GENERATING68K == 1
      // restore previous A5 for 68K-based CPUs
```

```
    SetA5(saveA5);
#endif
  }
}
```

This function is called at interrupt time. You can't allocate or dispose of memory, but you can use the `BlockMove()` function, set flags, and start new sounds.

This function is not invoked automatically. When you have started your sound with a `SndDoCommand()` or a `SndPlay()` command, you then must put into the sound queue a `callBackCmd`. You do this by calling `SndDoCommand()` with the command `callBackCmd`.

```
OSErr PostCallbackFunction(SndChannelPtr pChannel, long userData)
{
  SndCommand   theCmd;
  OSErr   theErr;

  if (pChannel)
  {
    theCmd.cmd = callBackCmd;
    theCmd.param1 = kSoundManagerID;

    // used for ID. Bug in SM that a callback is
    // sometimes called at the beginning of a sample
    // with the wrong information.
    // Use this to make sure that it is our callback.
    theCmd.param2 = userData;
    pChannel->userInfo = globalA5;
    theErr = SndDoCommand(pChannel, &theCmd, FALSE);
  }
  else
  {
    theErr = badChannel;
  }
  return theErr;
}
```

When you issue a `callBackCmd`, the first parameter in the `SndCommand` should be a defined constant. There is a bug in some versions of the Sound Manager that calls your callback before the sound is finished playing if the first parameter of the command is set to 0. To avoid this problem, use any value you wish other than 0.



The other important aspect of using callbacks is working with global data. The best strategy is to store the global data register (A5 on the 68K) in a safe place to retrieve once the callback function is called. The sound channel structure has a longword defined that is perfect for this.

```
long globalA5; // global register A5
```

```
void SetupGlobalA5(void)
{
#if GENERATING68K == 1
   // we do this here, so we be can be called via interrupt
   globalA5 = (long)SetCurrentA5();
#else
   // Nothing on the PowerPC
   globalA5 = 0;
#endif
}
```

This function should be called once or at the same time you allocate your sound channels. This preserves the global register for later use. The PowerPC is simpler because it isn't necessary to preserve the global data registers for interrupts, so the variable is just set to 0.

A callback can be useful for a variety of things, the most important of which is to start another sound. You can call `SndDoCommand()`, `SndDoImmediate()`, and `SndPlay()` at interrupt time. Doing this will create a smooth sound transition.

Volume Control

There are two kinds of volume control available with the Sound Manager: overall hardware control and individual volume control of each sample as it plays.

In all forms of volume control with the Sound Manager, the format is the same (see figure 6.2).

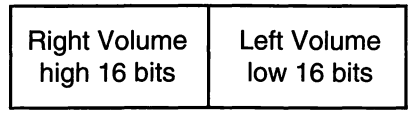


Figure 6.2 *32-bit longword.*

Volume level is stored as a 32-bit unsigned value in which the low 16 bits are used for the left channel and the high 16 bits are used for the right channel volume. 0 is silence and 256 is full or natural volume. With Sound Manager 3.0, you can overdrive the volume. For example, if you want to double the volume overall, you would pass 512 for each channel.

The expression `(512 << 16L | 512L` represents an easy way to conceptualize both volume levels together into one longword. Passing this value would double the overall volume.

```
OSErr GetDefaultOutputVolume(long *pLevel);
```

`pLevel` is a pointer to a 32-bit value. The Sound Manager will fill this longword with the current overall hardware volume.

```
OSErr SetDefaultOutputVolume(long level);
```

`level` is a 32-bit value. Each 16-bit word represents the volume level for the left and right stereo placement for all sound output.

```
OSErr GetSysBeepVolume(long *pLevel);
```

`pLevel` is a pointer to a 32-bit value. The Sound Manager will fill this longword with the current alert volume.

```
OSErr SetSysBeepVolume(long level);
```

`level` is a 32-bit value. Each 16-bit word represents the volume level for the left and right stereo placement for all alerts that play sound.

Most of the time, you just want to set the overall volume and leave it alone. What's most important is the ability to set individual volume over sound that you play on demand. By performing `SndDoCommand()` or `SndDoImmediate()`, you can dictate the volume and stereo placement of sound that is currently playing or about to play.

`volumeCmd` passes a 32-bit value into `param2`. Each 16-bit word represents the volume level for the left and right stereo placement for sound currently playing and for all subsequent sound in the channel in which this command was performed.

`getVolumeCmd` is a pointer to a 32-bit value that is placed in `param2`. This is the current volume level for this channel.

If you overamplify the volume, you risk distorting the audio playback. The sign that distortion is happening is a quick crackling. The other side effect of overdriving is sudden volume level changes. If two channels are playing with doubled volume and one finishes before the other, the Sound Manager will overdrive the volume of the other channel, causing a sudden jump in volume.

Panning

Given the following definitions...

```
#define kFullLeft -kFullVolume
#define kFullRight  kFullVolume
```

...stereo placement becomes a simple extension of left and right volume placement. The constant `kFullVolume` is defined by Apple as 256.



Using this simple formula, full left is -256, full right is 256, and dead center is 0. If you want to position a sound at 20 percent right from center, you pass the value 51. So, the values passed to the Sound Manager would be 205 for the left and 307 for the right, giving you the audio illusion of being about 20 percent right from the center.

```
void SetStereoPlacement(SndChannelPtr newChannel,
                        short positionMask)
```

```
{
   short theVolumeLeft, theVolumeRight;

   theVolumeLeft = kFullVolume - positionMask;
   theVolumeRight = kFullVolume + positionMask;

   // by breaking down the volumes here, we become a little
   // more flexible to compilers
   theCmd.param1 = 0;
   theCmd.param2 = (long)((long)theVolumeRight << 16L |
                     theVolumeLeft);
   theCmd.cmd = volumeCmd;

   // change volume now for this channel
   SndDoImmediate(newChannel, &theCmd);
}
```

Thus with this function, you can set the stereo position of a sample by walking through values -512 through 512. You can, for instance, call in your main loop and change the stereo placement at every value to have the sound position change from left to right.

Rate Playback Control

When a sound has been started, you can control aspects of the sound during playback. By modifying the playback rate, you can slow down or speed up a sound. This can be very effective in setting a mood or style.

Sending the `rateCmd` command alters the playback rate and duration of the current sound playing in the channel. It's relative to 22 KHz. For example, to play back a sound at 11 KHz, you would send 0x00008000 (0.5 in fixed point) for half the current playback rate.

`getRateCmd` returns the current playback rate relative to 22 KHz. If the sample is playing at 44 KHz, then the value 0x00020000 (2.0 in fixed point) would be returned.

An easier way to modify the rate is to use the actual rate of the sample with which you started.

```
UnsignedFixed SetRate(SndChannelPtr theChannel,
              UnsignedFixed newRate)
{
   UnsignedFixed  oldRate;
   SndCommand   theCmd;

   // get the current rate for this channel
   theCmd.param1 = 0;
   theCmd.param2 = (long)&oldRate;
   theCmd.cmd = getRateCmd;
   SndDoImmediate(theChannel, &theCmd);
```

```
    // Use this cool Toolbox routine to determine
    // the actual sample rate
    oldRate = UnsignedFixedMulDiv(rate22khz, oldRate, 0x10000);

    // Convert the passed rate into a relative sample multiplier
    newRate = UnsignedFixedMulDiv(newRate, 0x10000, rate22khz);

    // change rate now for this channel
    theCmd.param1 = 0;
    theCmd.param2 = (long)newRate;
    theCmd.cmd = rateCmd;
    SndDoImmediate(theChannel, &theCmd);
    return oldRate;
}
```

This function, given a valid sound channel and a new sample rate in fixed point, will modify the currently playing sound and return the old sample rate.

By changing the sample rate slowly, perhaps a fraction at a time, you can create cool effects.

Setting Up a Callback and a VBL Task

Creating a Vertical Blanking task, or VBL for short, is not essential, but it can become very useful for setting up an easy way to have a solid 1/60th of a second interrupt to process sound logic and other things. The VBL task has a long and torrid history. It's named VBL because on the old classic Macintosh hardware it actually was called during the blanking time of the built-in monitor. The particular VBL task that we use here does not actually interrupt during blanking time for the new Macintosh hardware, but at a constant rate. All we're interested in is the fact that it's a great 1/60th of a second timer that is consistent on all Macintosh platforms. There are new functions to get back to the real blank time of the current monitor, but they're not important for our use.

The basic theory for 68K- and PowerPC-based Macintoshes is primarily the same. The only difference is that there's more busy work for the 68K. You allocate a VBL structure, set up the interval, and for the 68K save the current A5 register, then start the task with the `VInstall()` Toolbox function.

The most important thing to remember about VBL tasks is that if you create the task inside of your application heap, it can get switched out and become prone to system level events.

The way to solve this problem is to allocate the task from the system heap and create a jump into your real procedure function.

Here are some typical globals for both 68K and PowerPC.

```
static long    * taskPtr;
static VBLTask * theSoundVBPtr;
```



The following function sets up a VBL task process for 68K-based Macintoshes. As you can see in the example, you allocate a block of memory in the system heap with one extra longword. This longword is used to save the current A5 global register.

Since we create a code level jump into our real procedure, we are breaking an important rule here: We are creating self-modifying code. It only happens once, but it happens, and in CPUs with an instruction cache, such as the 68030/68040, this will cause a cache fault. To fix this problem, as soon as we have built our new jump data, we flush the cache. This has no effect on earlier 68K CPUs, but logically fixes the problem in a very safe way.

```
// 68K-based VBL task setup and cleanup
static OSErr HYP_SetupVBLTask(void)
{
  OSErr theErr;
  struct Jump
  {
    short  bsr_inderect;  // 0
    void * address;       // 4
    long movea;           // 8
    short  jmp_a1;        // 12
  };
  struct Jump *heapJumpPtr;

/* This is the actual code we are building in the system heap.
    0x6104,     bsr.s  6(pc)
    0x0000, 0x0000,  dc.l 0
    0x225F         movea.l (a7)+, a1
    0x2251         move (a1), a1
    0x4Ed1         jmp  (a1)
*/

/* Set up Vertical Blank Interrupt */
  taskPtr = (long *)
       NewPtrClear((long)sizeof(VBLTask) + sizeof(long));
  heapJumpPtr = (struct Jump *)
         NewPtrSys((long)sizeof(struct Jump));
  if ( (taskPtr) && (heapJumpPtr) )
  {
    theSoundVBPtr = (VBLTask *)
         ( ((Byte *)taskPtr) + sizeof(long) );
    taskPtr[0] = SetCurrentA5();
    heapJumpPtr->bsr_inderect = 0x6104;
    heapJumpPtr->address = HYP_PreProcessVBLTask;
    heapJumpPtr->movea = 0x225F2251L;
    heapJumpPtr->jmp_a1 = 0x4ED1;

    theSoundVBPtr->vblAddr = (VBLUPP)heapJumpPtr;
    theSoundVBPtr->vblCount = 1;  // Every 1/60th of a second
```

```
        theSoundVBPtr->qType = vType;
        theSoundVBPtr->qLink = NULL;
        theSoundVBPtr->vblPhase = 0;

// Okay, flush the code/data cache
#ifndef __MIXEDMODE__
        if (*((char *)CPUFlag) >= 2)
#else
        if (LMGetCPUFlag() >= 2)    // is it an '020 or better?
#endif
        {
          FlushInstructionCache();
          FlushDataCache();
        }

        theErr = VInstall((QElemPtr)theSoundVBPtr);
    }
    else
    {
        if (heapJumpPtr)
        {
          DisposePtr((Ptr)heapJumpPtr);
        }
        if (taskPtr)
        {
          DisposePtr ((Ptr)taskPtr);
          taskPtr = NULL;
          theSoundVBPtr = NULL;
        }
    }
    return theErr;
}
```

The following function shows how to clean up after setting up the VBL task for a 68K-based Macintosh.

```
static OSErr HYP_CleanupVBLTask(void)
{
    if (theSoundVBPtr)
    {
        if (theSoundVBPtr->vblAddr)
        {
          DisposePtr ((Ptr)theSoundVBPtr->vblAddr);
        }
        VRemove((QElemPtr)theSoundVBPtr);
        theSoundVBPtr = NULL;
    }
```



```
  if (taskPtr)
  {
    DisposPtr((Ptr)taskPtr);
  }
  return noErr;
}
```

The following is the actual call to handle the VBL interrupt for 68K Macintoshes. As you can see from the example, we have to retrieve the A0 register from within our function. The register A0 points to our VBL task structure on entry, and the value -4 from A0 will be the A5 register that we need in order to access our globals. We set this up in our setup code. The `#pragma` is a funny way to define special inline code for most Macintosh compilers.

```
// 68K-based VBL task function

// This defines a 68K-based instruction that can work
// with any compiler. It generates the instruction:
//
//   MOVE.L  -4(A0),D0
//
// which is to retrieve the global A5 register that we saved.

#pragma parameter __D0 GetA0toVariable
pascal long GetA0toVariable(void) = {0x2028, 0xFFFC};

static pascal void HYP_PreProcessVBLTask(void)
{
  long saveA5, newA5;

  newA5 = GetA0toVariable();     // Globals register points the
                                 // VBL task structure

  saveA5 = SetA5(newA5);         // set current A5
  theSoundVBPtr->vblCount = 1;   // tell it to continue

  HYP_ProcessVBLTask();          // Okay, globals active, do something
                                 // interesting

  SetA5(saveA5);                 // restore previous A5
}
```

The PowerPC equivalents are simpler and cleaner overall. Same theory: Define the actual structure into the system heap. But none of the extra setup work is required:

```
// PowerPC-based VBL task setup and cleanup
static OSErr HYP_SetupVBLTask(void)
{
  OSErr  theErr;

// Set up Vertical Blank Interrupt
  theErr = noErr;
```

```
    taskPtr = NULL;
    theSoundVBPtr = (VBLTask *)NewPtrSysClear((long)sizeof(VBLTask));
    if (theSoundVBPtr)
    {
      theSoundVBPtr->vblAddr =
            NewVBLProc(HYP_PreProcessVBLTask);
      theSoundVBPtr->vblCount = 1 // Every 1/60th of a second
      theSoundVBPtr->qType = vType;
      theSoundVBPtr->qLink = NULL;
      theSoundVBPtr->vblPhase = 0;

      theErr = VInstall((QElemPtr)theSoundVBPtr);
    }
    return theErr;
}

static OSErr HYP_CleanupVBLTask(void)
{
    if (theSoundVBPtr)
    {
      if (theSoundVBPtr->vblAddr)
      {
        DisposeRoutineDescriptor(
              (VBLUPP)theSoundVBPtr->vblAddr);
        theSoundVBPtr->vblAddr = NULL;
      }
      VRemove((QElemPtr)theSoundVBPtr);
      DisposPtr((Ptr)theSoundVBPtr);
      theSoundVBPtr = NULL;
    }
    return noErr;
}
```

The VBL interrupt handler is simple and much easier to understand. Since there is no global register to worry about, you just set up your counter and process your logic:

```
// PowerPC-based VBL task function
static pascal void HYP_PreProcessVBLTask(VBLTaskPtr theVBLTask)
{
    theSoundVBPtr->vblCount = 1;     /* tell it to continue */

    HYP_ProcessVBLTask();
}
```



Now that you have a VBL task running every 1/60th of a second, what can you do with it? Well, you could change volume levels, control sounds, and process queues, or order take-out food every 1/60th of a second.

File Formats

The Sound Manager supports two basic file formats: special 'snd ' resources and AIFF files. Both formats are versatile, but both have their drawbacks. The most important drawback of using 'snd ' resources is the Resource Manager's size limit. Resources have a maximum size limit of 16 megabytes. Now, how could you possibly exceed that, you might ask? Well if you're streaming five minutes of music or voice, it will happen. The AIFF format does not have this limit, so it's the choice for large streamed audio files.

Creating 'snd ' Resources

Building 'snd ' resources inside of your own application has always been complex until now. The function `SetupSndHeader()` does most of the work for you. All you need to do is allocate a handle, call this function, then block move your sample data into the handle.

```
OSErr SetupSndHeader(SndListHandle sndHandle, short numChannels,
          UnsignedFixed sampleRate,
          short sampleSize, OSType compressionType,
          short baseFreq, unsigned long numBytes,
          short *headerLen);
```

`sndHandle` is a handle that is at least 200 bytes. This will contain the header information.

`numChannels` is the number of channels for the sample. Mono is 1, and stereo is 2.

`sampleRate`, the sample rate, is a fixed point number.

`sampleSize` is the sample bit size. Typically this value is 8 or 16 for 8- or 16-bit samples.

`compressionType` is a constant defined in the include file <AIFF.h>.

`NoneType` is for no compression.

`ACE2Type` and `ACE8Type` are for MACE compression 2-to-1 and 8-to-3.

`MACE3Type` and `MACE6Type` are for 3-to-1 and 6-to-1 compression.

If you use one of these compression values, you'll need to compress the data yourself before adding onto the handle created with this function.

`baseFreq` is the MIDI playback value in the range of 0 to 127. The constant `kMiddleC` is defined as 60.

`numBytes` is the actual byte length of the sample data. This may not be the actual number of sample frames.

`headerLen` is returned. It's the total number of bytes for the header.

The following function creates and returns a handle from a pointer of sample data. First, you create a handle that contains the sample header, increase its size, and move the real sample data into the handle; then, you finally call `SetupSndHeader()` again with the real information.

```
Handle HY_CreateSndResourceFromPtr(Ptr pSample, long dataLength,
                UnsignedFixed sampleRate,
                short bitSize,
                short numChannels,
                short int baseFreq)
{
   OSErr   theErr;
   Handle  theSoundHeader;
   Handle  theSound;
   short   headerLength;

   theSound = NULL;
   // first allocate enough of a handle to set up
   // the header information
   theSoundHeader = NewHandleClear(200L);
   if (theSoundHeader)
   {
      theErr = SetupSndHeader((SndListHandle)theSoundHeader,
                numChannels, sampleRate,
                bitSize, 'NONE',
                baseFreq, 0, &headerLength);
      if (theErr == noErr)
      {
         // okay, now we know how large header info is, so
         // we can block move the sample data right after
         theSound = NewHandle(headerLength + dataLength);
         if (theSound)
         {
            HLock(theSound);
            BlockMove(*theSoundHeader, *theSound,
                   headerLength);
            BlockMove(pSample, ((Byte *)*theSound) +
                   headerLength, dataLength);
            HUnlock(theSound);
            DisposeHandle(theSoundHeader);

            // okay, now set the new length of the data
            theErr = SetupSndHeader((SndListHandle)theSound,
                        numChannels,
                        sampleRate,
                        bitSize, 'NONE',
                        baseFreq,
                        dataLength,
                        &headerLength);
            if (theErr)
            {
               DisposeHandle (theSound);
               theSound = NULL;
            }
         }
      }
      else
```



```
    {
      DisposeHandle (theSoundHeader);
    }
  }
  return theSound;
}
```

Creating AIFF Files

The following function simplifies the process of creating AIFF files, the other Apple sound format standard. All you need to do is create and open a file with the type AIFF or the constant AIFFID, pass the file reference number to this function with the information about the sample data, and block write the date; then close the file.

```
OSErr SetupAIFFHeader(short fRefNum, short numChannels,
          UnsignedFixed sampleRate,
          short sampleSize, OSType compressionType,
          unsigned long numBytes,
          unsigned long numFrames);
```

`SetupAIFFHeader()`'s parameters are much like `HY_CreateSndResourceFromPtr()`'s, but there are enough differences that we'll go through these parameters too.

`fRefNum` is a file reference number from an open file.

`numChannels` is the number of channels for the sample. Mono is 1, and stereo is 2.

`sampleRate`, the sample rate, is a fixed point number.

`sampleSize` is the sample bit size. Typically this value is 8 or 16 for 8- or 16-bit samples.

`compressionType` is a constant defined in the include file <AIFF.h>.

`NoneType` is for no compression.

`ACE2Type` and `ACE8Type` are for MACE compression 2-to-1 and 8-to-3.

`MACE3Type` and `MACE6Type` are for 3-to-1 and 6-to-1 compression.

If you use one of these compression values, you'll need to compress the data yourself before adding onto the handle created with this function.

`numBytes` is the actual byte length of the sample data. This may not be the actual number of sample frames.

`numFrames` is the number of sample frames. For example, a mono 16-bit sample that is 10,000 bytes in length will have 5,000 sample frames.

The use of this function is similar to `SndSetupHeader`. Once the file is open, you call `SetupAIFFHeader()` once at the beginning to determine the header size, write out your sample data, and then call it again to fill in the details.

One advantage AIFF files have over 'snd ' resources is that they can span megabytes in length. It would be very easy to write AIFF blocks of data, for example, in an audio processing application. The function below shows an example of writing an AIFF file from a pointer in memory.

```
OSErr HY_CreateAIFFFileFromPtr(FSSpec *pFile, Ptr pSample,
              long dataLength,
              UnsignedFixed sampleRate,
              short bitSize, short numChannels)
{
  OSErr  theErr;
  short int fileRef;
  long   length;

  theErr = badFormat;
  if ( (bitSize == 8) || (bitSize == 16) ||
       (numChannels == 1) || (numChannels == 2) )
  {
    theErr = FSpCreate(pFile, 'hlly', AIFFID , 0);
    if (theErr == noErr)
    {
      theErr = FSpOpenDF(pFile, fsRdWrPerm, &fileRef);
      if (theErr == noErr)
      {
        SetFPos(fileRef, fsFromStart, 0L);
        // Call first to set up the basic info
        theErr = SetupAIFFHeader(fileRef, numChannels,
               sampleRate, bitSize, NoneType,
                      0L, dataLength);
        if (theErr == noErr)
        {
          length = dataLength * numChannels *
                (bitSize / 8);
          // Write out our sample buffer
          theErr = FSWrite(fileRef, &length,
                        pSample);
          if (theErr == noErr)
          {
            // Reposition back to the start and
            // call again to fill in the details
            SetFPos(fileRef, fsFromStart, 0L);
            theErr = SetupAIFFHeader(fileRef,
                   numChannels,
                   sampleRate,
                   bitSize, NoneType,
                   length, dataLength);
          }
          FSClose(fileRef);
        }
      }
```



```
    }
  }
  return theErr;
}
```

The Future of the Sound Manager

If someone at Apple is reading this, here are a few suggestions for features I'd like to see in the New Improved Sound Manager.

With Apple technologies such as QuickDraw 3D, I'd like to see 3-D placement of sound with the Sound Manager. You can do it now, primarily because of stereo support, but it's hard.

I'd also want a way to get positional information from the sound that is currently playing. This is useful for timing. Audio is becoming the driving force behind a lot of games, and this small but very useful feature would allow animation in sync with audio to be much easier.

The last very simple improvement would be for the automatic allocation of an asynchronous sound channel for `SndPlay()`. This, in its simplest form, would allow you to call `SndPlay()` with a 'snd ' resource handle and nothing else.

Hollywood Sound API

Hollywood Sound API is a simple layer over the Sound Manager that allows you to play 8-bit, 16-bit, and stereo sound effects. There are many reasons to create a layer, but the most compelling one is that a layer has a unifying effect on all of the features of a particular set of function calls. You can always go too far and smother the function calls beneath an overly complex API, so you walk this fine line between useful and not so useful.

Setup and Cleanup

To set up the audio system, you would do something like this:

```
HY_Setup(kNormal);  // Set up audio for 8-bit mono
```

This would set up the audio for 8-bit and 16-bit mono output. For Macintoshes without 16-bit hardware, Sound Manager 3.0 will convert the data into 8-bit samples. It's not desirable to play 16-bit samples on a Macintosh without 16-bit output hardware for the primary reason that it takes too much CPU time to process the samples.

To do something more sophisticated, such as stereo or 16-bit audio, you might do something like this:

```
HY_Setup(kUseStereo + kMaxQuality);
```

This would set up the audio for best quality (interpolation and sample rate conversion). Stereo has the side effect that even though there may be hardware support, the right channel

of the data goes through only a back panel connector. Typically, for ultimate stereo support, you must warn users that they may lose the right channel. But if their Macintosh is hooked up to external speakers, then there's no problem.

The following two functions do all the work for setting up the channels and cleaning up after you're all finished.

```
OSErr     HY_Setup( short int featureMask,
               short int maxVoices);
```

Parameter `featureMask` passes in how you want to set up the audio system. Pass `kNormal` for normal quality and mono. Pass `kUseStereo` to allow for stereo, and `kMaxQuality` for the best quality. You can add these values together.

`maxVoices` is how many voices to allocate for playback. If you allocate only one voice and try to play a sound on any voice other than 0, you will get an error. Typically, most color Macintoshes with Sound Manager 3.0 can allocate four voices. You'll get an error if you try to allocate too many.

The `HY_Cleanup()` function deallocates all memory associated with the channels that were allocated with `HY_Setup()`.

```
OSErr     HY_Cleanup(void);
```

Memory Management

There are two ways to play samples with Hollywood. The easiest way for the Hollywood API is to provide a pointer to the sample data that you want to play, and you handle all the management of that memory. The simplest way for you is to create all your sound data into 'snd ' resources. Then you can use the resource manager and Hollywood to keep track of all the memory.

All resources or allocated handles of data must be registered with Hollywood before using.

```
SndReference HY_GetSoundResource(short int resourceID);
```

`resourceID` uses the resource manager to get a 'snd ' resource with the `resourceID`. NULL is returned if there is an error. This handle will then become registered with Hollywood, so there is no need to call `HY_RegisterThisSound()`.

```
void    HY_RegisterThisSound(SndReference theSound);
```

`theSound` is a resource handle or an allocated memory handle. This call associates this block of memory with Hollywood. The handle becomes locked. Do not unlock this memory until after calling `HY_UnregisterSoundResound()` or `HY_UnregisterAllSoundResources()`. Doing so may result in static or the wrong sound being played when you need it.

```
void    HY_UnregisterSoundResound(SndReference theSound);
void    HY_UnregisterAllSoundResources(void);
```



theSound is a resource handle or an allocated memory handle. This call removes the association of this block of memory with Hollywood. Hollywood won't free or release the memory. If you try to play a handle that is no longer registered, Hollywood returns an error.

The following code gets a 'snd ' resource, registers it, locks it, and returns a SndReference back to you.

```
{
   SndReference theSound;

   theSound = HY_GetSoundResource(128);
   if (theSound)
   {
   // Play, or do something else interesting

      HY_UnregisterSoundResound(theSound);
   }
}
```

It's important not to unregister a sound while it's playing. If you do, Hollywood might play back static or some other undesirable sound.

Given just a data handle, you can register it with Hollywood for playback. Registration locks the handle, but nothing else.

```
{
   Handle  theSound;

   theSound = GetResource('snd ', 128);
   if (theSound)
   {
      HY_RegisterThisSound(theSound);
      // Play back
      HY_UnregisterSoundResound(theSound);
   }
}
```

Playing Sound

Hollywood has three ways to play sound: handle-based, pointer-based, and file-based.

```
OSErr     HY_PlaySoundHandle(short int voiceNumber,
               SndReference theSound,
CustomCallbackProc callbackProc,
               long userData,
               Boolean killSound);
```

voiceNumber is the voice that you want this sound to play through. If there is a sound already playing, an error will be returned if killSound is FALSE. voiceNumber should be in the range

of 1 to the maximum number of voices allocated with `HY_Setup()`. You can pass the constant `kAnyVoice` to automatically allocate a voice.

`theSound` is a memory handle that has been registered with `HY_RegisterThisSound()` or a handle that has been retrieved with `HY_GetSoundResource()`.

`callbackProc` is a callback function that is called after this sound has finished playing. See "Hollywood Callbacks" for more details.

If you are using callbacks, you can pass a user-defined value. Typically, `userData` is used to identify the sound being played after it is finished.

If `killSound` is TRUE, then the previous sound is forced to stop in favor of this new sound. If it's FALSE and there's a sound playing, an error is returned.

You can easily start a sound playing. The following code will register and load a 'snd ' resource number 128 and start playing it on any open voice with no callback.

```
{
   SndReference theSound;

   theSound = HY_GetSoundResource(128);
   if (theSound)
   {
      HY_PlaySoundHandle(kAnyVoice, theSound, NULL, 0L, TRUE);
   }
}
```

The other variance is if you have all the data yourself and want to play a sample:

```
OSErr     HY_PlaySample( short int voiceNumber,
               void *pSample,
               long length, UnsignedFxied rate,
               short dataBitSize,
               short channelSize,
CustomCallbackProc callbackProc,
               long userData,
               Boolean killSound);
```

As with `HY_PlaySoundHandle()`, `voiceNumber` is the voice that you want this sound to play through. An error is returned if `killSound` is FALSE and there is a sound already playing, and you can pass the constant `kAnyVoice` to automatically allocate a voice. All the features you've come to expect from Hollywood API!



`pSample` is a pointer to a sample. This pointer can be either 8-bit, 16-bit, mono or stereo data. It follows the same data format as the Sound Manager.

`length` defines the number of audio frames, not necessarily the number of bytes of the sample.

`rate` is the sample rate to play back this sample. It's in fixed point. The constants `rate44khz`, `rate22050hz`, `rate11025hz`, `rate22khz`, and `rate11khz` define the more popular sample rates.

`dataBitSize` is either 8 or 16, depending on the type of data. If 8, then the pointer `pSample` is one audio frame per byte. If 16, then it's one audio frame per word.

`channelSize` tells the number of channels: either 1 for mono or 2 for stereo data. If 1, then each audio frame is mono data. If 2, then each audio frame is interleaved with left and right sample data.

`callbackProc`, `userData`, and `killSound` are the same as they were in `HY_PlaySoundHandle()`.

```
OSErr     HY_StartFilePlay(short int voiceNumber,
               FSSpec *pFile,
               CustomCallbackProc customCallback,
               long userData,
               long bufferSize,
               Boolean killSound);
```

`voiceNumber` and `killSound` do what they've done in the previous routines.

`pFile` is a valid `FSSpec`. It points to a close file. `HY_StartFilePlay()` will open and close the file for processing.

`customCallback` does the same thing as `callbackProc` did in the earlier functions. `userData` is used to identify the sound being played after it is finished.

`bufferSize` tells how much data to allocate for reading the file. If 0 is passed, then `HY_StartFilePlay()` allocates a default amount of 20,480.

If `killSound` is TRUE, then the new sound replaces any previous sound. If `killSound` is FALSE and there is a sound playing, an error is returned.

Once sound is playing, you can use `HY_StopSample()` to stop the sound. This works for any of Hollywood's sample playback functions.

```
void    HY_StopSample(short int voiceNumber);
```

`voiceNumber` defines which voice you want to be silenced if there is a callback defined. It's called with the constant `kSoundDoneForcedStop`. See the next section for more details.

Hollywood Callbacks

All audio callbacks in Hollywood are the same. When you call a Hollywood API function that requires a callback, you pass in your callback function and special user-defined data. When the sound is finished playing, Hollywood, using the Sound Manager, calls your callback function with the special user-defined data. Your callback function may be called up to two times depending on the action required.

A Hollywood callback looks like this:

```
void HandleHollywoodCallback(short int voiceNumber,
              short int what, long userData);
```

`voiceNumber` is the voice on which the sound is finishing. This should be in the range of 1 to the maximum number of voices allocated with `HY_Setup()`.

`what` is one of three constants:

- `kSoundDoneNormalStop` is passed at interrupt time. It's important not to allocate or deallocate memory at this time. You're guaranteed that this message is right before the sound has finished.
- `kSoundDoneNoInterrupt` is passed during noninterrupt time. It's safe to allocate and deallocate memory. You're guaranteed that this message occurs sometime after the sound has finished.
- `kSoundDoneForcedStop` is passed if this sound is stopped or killed. This may or may not be called at interrupt time. You won't receive any other messages after this one.

`userData` is the same value that was passed into `HY_PlaySoundHandle()`, `HY_PlaySound()`, or `HY_StartFilePlay()`.

Determining Features

Sometimes it's necessary to determine features of the current Macintosh platform before you set up or initialize anything. The following functions help you out. You can call these before calling `HY_Setup()`.

```
Boolean   HY_IsStereoAvailable(void);
```

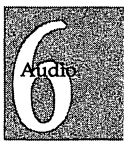
The function above returns TRUE if stereo hardware is available. It does not, however, tell you if the user has external speakers attached. Some Macintosh systems can play back stereo but users won't be able to hear it unless there are external speakers.

```
Boolean   HY_Is16BitAvailable(void);
```

This function returns TRUE if this Macintosh is capable of playing 16-bit data. Another valuable use for this function is that it can determine if this Mac is a PowerPC, 660AV, or 840AV system. These systems can play back 16-bit audio.

```
Boolean   HY_SoundActive(void);
```

This function returns TRUE if Hollywood has been set up correctly. If it's FALSE, then calling any Hollywood function has no effect.



Getting Information and Changing Sounds That Are Playing

This group of functions enables you to query Hollywood about itself and the sounds it is playing, and to change the current sounds that are being played.

```
Boolean   HY_IsVoiceEmpty(short int voiceNumber);
```

`voiceNumber` defines which voice you want to query about. This should be in the range of one to the maximum number of voices allocated with `HY_Setup()`.

`HY_IsVoiceEmpty()` returns TRUE if there is no sound being played on the requested voice, and FALSE if there is something playing.

Changing Sample Rates

```
void    HY_SetRate(short int voiceNumber,
            UnsignedFixed newRate);
```

`voiceNumber` defines which voice you change. There is no error checking here, so be sure you are requesting a voice that has been allocated with `HY_Setup()`.

`newRate` is the new sample rate for the currently playing sample. This affects the sample immediately.

```
UnsignedFixed  HY_GetRate(short int voiceNumber);
```

`voiceNumber` is the voice you want to query about.

`HY_GetRate()` returns a fixed point value of the sample rate of the currently playing sample. If there is no sample being played, the rate of 22 KHz is returned.

Changing Volume Levels

```
void    HY_SetVolume(short int voiceNumber,
              SoundVolume newVolume);
```

`voiceNumber` defines which voice you change.

`newVolume` is the new volume level. The structure `SoundVolume` contains a left and a right element. The ranges are from 0 to `kFullVolume`. You can overdrive the samples by passing values larger than `kFullVolume`, but you should be careful because this can create distortion.

```
SoundVolume  HY_GetVolume(short int voiceNumber);
```

`voiceNumber` is the voice you want to query about.

`HY_GetVolume()` returns a `SoundVolume` structure which contains a left and a right element. Each element is in the range of 0 to `kFullVolume`. Values larger than `kFullVolume` are allowed and mean that the voice is overdriving that particular sample.

Changing Stereo Positions

```
void    HY_SetStereoPosition(short int voiceNumber,
            short positionMask);
```

`voiceNumber` defines which voice you change.

`positionMask` is the stereo positioning value for this voice. The range is from `kFullLeft` (-256) to `kMiddle` (0) to `kFullRight` (256).

```
short int HY_GetStereoPosition(short int voiceNumber);
```

`voiceNumber` is the voice you want to query about.

This function returns the current stereo position for this voice, from `kFullLeft` (-256) to `kMiddle` (0) to `kFullRight` (256).

Other Mundane but Important Functions

There are three other functions that are important, but once you put them into your application you can forget about them.

```
void    HY_ServiceTasks(void);
```

This function handles all of Hollywood's memory management and callback processing. Without this call, Hollywood would not process callbacks or handle memory-related tasks. A good place to put this is right after `WaitNextEvent()` or `GetNextEvent()` in your main event handler.

```
OSErr     HY_ResumeHardware(void);
OSErr     HY_PauseHardware(void);
```

These two functions are useful when your application is about to be suspended and resumed. Don't call these at interrupt time; you can call them during event time to pause and resume Hollywood.

VBL Tasks

Hollywood makes creating VBL tasks easy. All you need to do is call the following function with a function pointer of the code you want executed every 1/60th of a second. Global variables will be set up and ready to go. You can't allocate memory or deallocate memory, but you can use `BlockMove()` and start sounds.

```
void    HY_SetSoundVBCallBack(void (*theProc)(void));
```



MACE Compression

Hollywood makes using MACE a breeze. All MACE-compressed 'snd ' resources are handled automatically by `HY_PlaySample()`, `HY_PlaySoundHandle()`, and `HY_StartFilePlay()`. But what about creating MACE resources from a block of sample data? Here's how to do that:

```
Handle    HY_CreateMACESndResourceFromPtr(short maceType,
                    Ptr pSample,
```

```
                    long dataLength,
                    UnsignedFixed sampleRate,
                    short bitSize,
                    short numChannels,
                    short int baseFreq);
```

`maceType` uses the Sound Manager constants `threeToOne` and `sixToOne` to determine what kind of compression to use for this data.

`pSample` is a pointer to sample data.

`dataLength`, the length of data, may not be actual bytes.

`sampleRate` is the fixed point sample rate.

`bitSize` tells the sample bit size. This function supports only 8-bit data, but it's here as a stub for expansion.

`numChannels` is the number of channels. Mono is 1 and stereo is 2. This function supports only mono data, but it's here as a stub for expansion.

`baseFreq` is the MIDI playback frequency. The constant `kMiddleC` is 60.

Hollywood forces you to think about bit size and number of channels, but supports only 8-bit mono data. MACE can compress other kinds of data, but the actual details are left up to you. To create stereo compressed data, you need to build buffers, one for each channel. Compress each channel into that buffer by calling the Sound Manager's `Comp3to1()` or `Comp6to1()` functions with the last parameter set to the channel you are compressing. Then, to rebuild the final data, you need to interleave the two buffers into one buffer for the final output.

Creating and Obtaining Information about 'snd ' Resources

Here are two functions to simplify the process of creating valid 'snd ' resources.

```
OSErr     HY_GetSoundResourceInformation(    Handle theSnd,
               long *pLoopStart,
               long *pLoopEnd,
               long *pSampleOffsetStart,
               long *pTotalSize,
               short *pBaseKey,
               short int *pNumChannels,
               short int *pBitSize,
               UnsignedFixed *pRate,
               short int *pCompressionType);
```

`theSnd` is a handle to a 'snd ' resource obtained from the Resource Manager.

`pLoopStart` and `pLoopEnd` are pointers to a longword. These return the loop point starting and ending points.

`pSampleOffsetStart` is a pointer to a longword. It returns the offset from the beginning of the handle at which the actual sample data begins.

`pTotalSize` is a pointer to a longword. This is the total size in bytes of the sample data.

`pBaseKey` is a pointer to a long short. This is the MIDI playback frequency, and is in the range of 0-127. The constant `kMidddleC` is 60.

`pNumChannels`, a pointer to a short, is the number of channels. Mono is 1, and stereo is 2.

`pBitSize` is a pointer to a short. This is the sample bit size and is typically 8 or 16.

`pRate` is a pointer to an `UnsignedFixed`. This is the sample rate in fixed point.

`pCompressionType` is a pointer to a long short. The Sound Manager constants `threeToOne` and `sixToOne` determine what kind of compression to use for this data.

Now that you can get information about a 'snd ' resource, how about creating them? The following function, given the basic information, creates a 'snd ' resource which you can pass to the Sound Manager.

```
Handle    HY_CreateSndResourceFromPtr(void * pSample,
                  long dataLength,
                  UnsignedFixed sampleRate,
                  short bitSize,
                  short numChannels,
                  short int baseFreq);
```

`pSample` is a pointer to sample data. This can point to 8-bit, 16-bit, mono, or stereo data, and depends upon `dataLength`, `bitSize`, and `numChannels` to interpret the data.

`dataLength` is the number of sample frames. This may not be the actual number of bytes if stereo or 16-bit data.

`sampleRate` is the sample rate in fixed point.

`bitSize` is the sample bit size—typically 8 or 16.

`numChannels` equals the number of channels. Mono is 1 and stereo is 2.

`baseFreq` is the MIDI playback frequency, which is in the range of 0-127. The constant `kMidddleC` is 60.

`HY_CreateSndResourceFromPtr()` returns NULL if it cannot create a valid 'snd ' resource.



QuickTime

QuickTime is Apple's System software that provides multimedia capabilities. QuickTime manages time-based data such as video, audio sound effects, and music. It also manages the low-level details of delivering data in a timely manner from disk or CD-ROM, compressing

or decompressing video and audio data, and synthesizing or delivering music to MIDI devices. QuickTime also supports text, still pictures, and other media types.

The performance and feature set of QuickTime have come a long way since version 1.0. Apple has added new video compression algorithms, such as Cinepak, which can deliver video at 320 x 240 resolution at 30 frames per second on most Macs now shipping. Support for text and audio are improved as well.

QuickTime's usage of RAM is much lower with versions 1.5 and later than with the original release. The Component Manager, an object-oriented code storage system, provides QuickTime with the ability to load only the portions of it that are needed for playback or recording at the time they're needed.

QuickTime is available for PC compatibles with Windows. Your QuickTime content will play on PCs, easing your crossplatform efforts.

QuickTime Movies

Movies are files that can contain one or more "tracks," each with its own type of media. For example, movies commonly contain both a video and an audio track. QuickTime keeps the tracks in synchronization with each other as they play. You may also, for example, consider the use of multiple audio or text tracks and control which is being played depending on where your product has been localized.

You can create and edit movies with simple utilities such as MoviePlayer to get your feet wet, then use more sophisticated editing tools, available from many sources, to make content with a more professional feel.

Once you've created a movie, it's a simple matter to use it in your application by calling the Movie Toolbox. There are even HyperCard XCMDs available to play back QuickTime movies from your stacks. Later, I'll discuss how to open and play a movie from within a C program.

QuickTime Music Architecture

One of the most significant new additions in version 2.0 of QuickTime is the Music media type. QuickTime Music is based on the General MIDI model for device-independent instrumentation, but also supports custom instruments. General MIDI is used as a least common denominator and is available if the user's setup doesn't include the MIDI or software-based instrument needed to play the desired sound.

The software-based music synthesizer included with QuickTime 2.0 is not General MIDI-compliant, but provides a reasonable substitution for many of the instruments found in General MIDI. Apple included only a subset of the instruments found in General MIDI due to space and memory limitations. However, the software synthesizer will include better expansion capabilities in future versions of QuickTime, and may in the future become fully compliant with the General MIDI standard.

QuickTime's software-only solution for music performance is designed to be scalable. If your application is running on a Mac LC, you'll have fewer simultaneous voices available to play with and the sound quality will be lower. On the other hand, on a Power Mac 8100/110, a full 24-voice sequence playing with reverb may tax the CPU about the same amount. QuickTime's software synthesizer measures the CPU's ability to synthesize music and adapts its performance strategy to take a comparable amount of CPU time no matter what Mac hardware it's being played on.

Just as with traditional audio/video movies, it's easy to incorporate music content into your application through QuickTime. The Movie Toolbox gives your application complete control over playback of movies, including music movies. But first, you'll have to get your songs into movie form. Fortunately, QuickTime makes this easy.

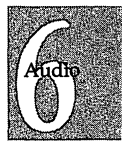
QuickTime can import Standard MIDI Files (SMF) and convert them to QuickTime music tracks. SMF is the universally adopted file format for storing MIDI sequences on disk, so you'll have ready access to a vast array of premade content, as well as be able to compose and edit your own music with any sequencing package.

QuickTime doesn't actually store music information in standard MIDI form internally. Instead, it uses a proprietary format that is a superset of regular MIDI functionality. For example, MIDI has traditionally supported only 16 channels per device. QuickTime abandons the "MIDI channel number" concept to enable the performance of as many simultaneous instruments as a device can handle. Knobs and controller values are precisely defined, guaranteeing accurate results when, for example, bending the pitch on an arbitrary device. Likewise, pitch can be specified with much higher precision than the typical diatonic scale used by MIDI. While these may seem to be minor differences, and because SMFs are the easiest way to import music data into QuickTime, these capabilities will be more useful in the future as the computers and music devices become more powerful.

Creating Content

If you're not the musically creative type, hire a musician or consider buying some canned music sequences. Several companies are in the business of supplying General MIDI orchestrated SMFs, and some have fine-tuned their songs to sound best with the instruments built into QuickTime's software synthesizer.

Keep the copyrights in mind when you find content you want to use, especially on a bulletin board or handed to you by a friend: Many SMFs floating around are not authorized by the original artists.



If you're musically skilled, buy a music sequencing package. The music marketplace is very competitive, so you'll be able to find entry-level music sequencing software at reasonable prices. There are many different styles of sequencers, so it's highly recommended that you spend some time at a music software dealer or with friends, trying out the packages to find the one that best fits your style.

You'll also want a MIDI interface and a General MIDI-compatible synthesizer such as Roland's Sound Canvas. Prices and features vary substantially for General MIDI synthesizers. While QuickTime will support a few non-General MIDI synthesizers, it's best to have General MIDI since it's the least common denominator that QuickTime needs.

Once you've written or acquired your masterpiece, save it as a Standard MIDI File. If you already own a non-General MIDI synthesizer, now is the best time to reorchestrate the file for General MIDI. Some sequencing packages help you choose the appropriate equivalent instrument. If you're unsure, don't worry. You'll have another chance to reorchestrate when your file is imported.

Importing Content into a QuickTime Movie

Launch MoviePlayer 2.0 or any later version. (Early versions of MoviePlayer don't allow you to reorchestrate your music or adjust volume levels after the file has been imported.)

From the File menu, choose the Import menu item. A Standard File dialog box appears. Select a Standard MIDI File and click on the Convert button (see figure 6.3).

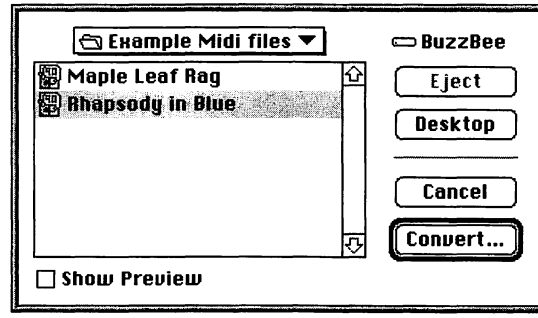


Figure 6.3 *Standard Open dialog.*

Give a name for the movie you'll be converting the file to, then click on the Options button to change the default instrument selection (see figure 6.4).

Click on the Save button to save the movie.

Click on the name of any instrument on the list. Click on the Instrument button. Click on the OK button when you're done reorchestrating (see figure 6.5).

Select an instrument and/or synthesizer from the Instrument Picker dialog box (figure 6.6).

You can play the song and listen to it with the new orchestration while the Options dialog is still active.

Click on the Cancel button if you were happier with the original selection, or click on OK if you found a better instrument for the current part.

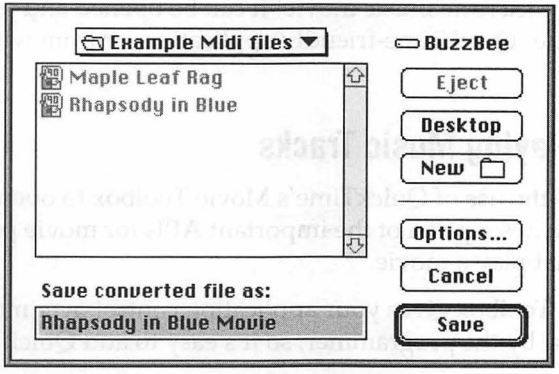


Figure 6.4 *Save dialog.*

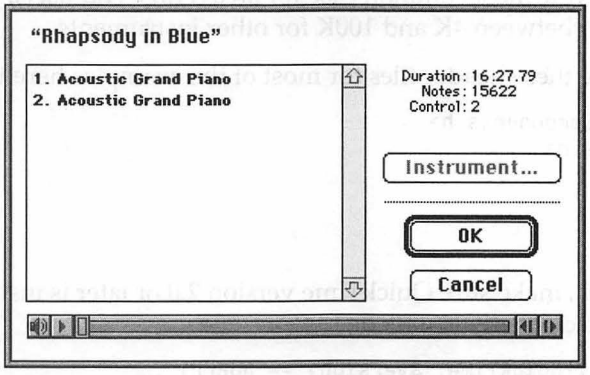


Figure 6.5 *Instrument Track dialog.*

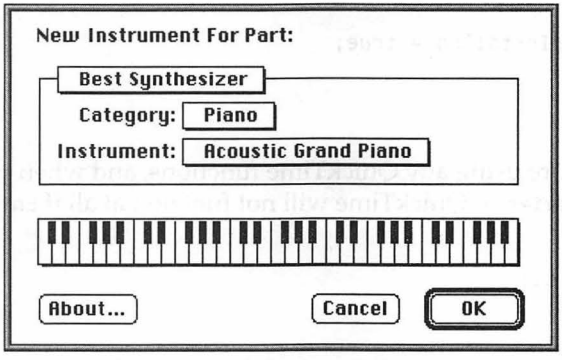


Figure 6.6 *Instrument Picker dialog.*

You've just created a QuickTime music movie! It can be opened and played from MoviePlayer or any other QuickTime-friendly application, or from within your own program.

Embedding and Playing Music Tracks

This section deals with the use of QuickTime's Movie Toolbox to open and play movie sequences. It includes a description of the important APIs for movie playback and example code fragments in C that play a movie.

The QuickTime Movie Toolbox gives your application control over movie playback. It's designed for ease of use by the programmer, so it's easy to add QuickTime content to your application.

Your own application should have at least 300K bytes of free RAM set aside to use the Software Synthesizer (the most common case for an average end user), plus about 120K of RAM for the piano, or between 4K and 100K for other instruments.

You'll need to include these header files for most of the examples here to work:

```
#include <QuickTimeComponents.h>
#include <Components.h>
```

Setup

To start playing music, make sure QuickTime version 2.0 or later is installed by checking the version of QuickTime currently installed:

```
if (Gestalt(gestaltQuickTime, &version) == noErr)
{
   version >>= 16L;
   if (version >= 0x200L)
   {
      quickTimeMusicInstalled = true;
   }
}
```

Use `EnterMovies()` before using any QuickTime functions, and when you are finished with QuickTime, call `ExitMovies()`. QuickTime will not function at all if `EnterMovies()` has not been called.

```
OSErr EnterMovies(void);
void ExitMovies(void);
```

Opening and Closing Movie Files

These functions open, close, create, and dispose of QuickTime movies:

```
OSErr OpenMovieFile(FSSpec *pFile, short *resRefNum,
         SInt8 permission);
```

`pFile` is a standard file specification returned from `StandardGetFile()`. It points to a valid QuickTime movie file.

`resRefNum` is a pointer to a file reference number created when the file is opened. Pass this to `CloseMovieFile()` to close up this file.

`permission` defines the standard file sharing parameters. The constants defined are `fsCurPerm`, `fsRdPerm`, `fsWrPerm`, `fsRdWrPerm`, and `fsRdWrShPerm` for current permissions, read-only permission, write-only permission, read/write permission, and read/write shared permission.

Use this to close a movie file:

```
OSErr CloseMovieFile(short resRefNum)
```

`resRefNum` is the reference number returned from `OpenMovieFile()`.

Once the movie file is open, you need to create a movie in RAM that can play the movie file. You need to call `NewMovieFromFile()` before you can play the movie. This function creates a Movie from a file that was opened with `OpenMovieFile()`.

```
OSErr NewMovieFromFile(Movie *theMovie, short resRefNum,
            short *resId, StringPtr resName,
            short newMovieFlags,
            Boolean *dataRefWasChanged)
```

`theMovie` is the new Movie created. Pass a pointer to a Movie structure.

`resRefNum` is the reference number returned from `OpenMovieFile()`.

Pass a pointer to a short to get the resource ID from the newly created Movie, otherwise pass NULL. The pointer should be set to NULL to load the first resource in the Movie. `resId` would be used if multiple movie resources are stored in a resource file.

Pass a pointer to a string to get the resource name from the newly created Movie, otherwise pass NULL. `resName` is the name of the resource actually loaded if loading a Movie resource.

`newMovieFlags` determines how data references should be resolved and whether the Movie should be active. Set to the value `newMovieActive` to make the Movie active.



`dataRefWasChanged` is a pointer to a Boolean. It is TRUE if the Movie Toolbox has to change data references as they're being resolved. Set this pointer to NULL to ignore the information.

The following function disposes of all memory associated with the Movie passed:

```
void DisposeMovie(Movie theMovie)
```

`theMovie` is the valid Movie pointer.

To convert MIDI files, use the following function. Given the MIDI file as the input and a new file from specification from `StandardPutFile()`, this function creates a new Movie from the MIDI data.

```
OSErr ConvertFileToMovieFile(FSSpec *inputFile, FSSpec *outputFile,
              OSType creator, ScriptCode scriptTag,
              short *resID,
              long flags,
              ComponentInstance userComp,
              MovieProgressUPP pProc, long refCon);
```

`inputFile` is a valid file specification. For our examples, it should point to a MIDI file.

`outputFile` is a valid file specification for a new file. This is a standard QuickTime movie file.

`creator` refers to the application creator. MoviePlayer 2.0's creator type is 'TVOD'.

Pass `smSystemScript` to `scriptTag` for the standard system or `smCurrentScript` for the current script system the user has selected.

Pass `resID` a pointer to a short to get the resource ID from the newly created Movie; otherwise pass NULL.

If `showUserSettingsDialog` is passed to `flags`, then the standard instrument dialog will be presented to the user to customize the final music movie file.

Pass NULL for `userComp` in our example. This is used when you have a particular component that you want to use for data conversion.

Pass NULL for `pProc` also in our example. This is used to create a custom function that is called during the conversion process. You could, for example, create your own progress bar dialog box.

The value of `refCon` is passed to reference your custom data. Passing zero should be the default.

Basic Movie Functions

Once you have a valid movie, you can use these functions to control the movie playback. They all take the same parameter, a valid Movie pointer.

Use `StartMovie()` to start a movie playing. If there were a video track, we'd have to open a window and tell QuickTime to play in it. See the example code below. It checks whether the movie contains video and opens an appropriately sized window if necessary.

```
void StartMovie(Movie theMovie)
```

To stop the current movie playing, do this:

```
void StopMovie(Movie theMovie)
```

This sets the current playback position to the beginning:

```
void GoToBeginningOfMovie(Movie theMovie)
```

This sets the current playback position to the end:

```
void GoToEndOfMovie(Movie theMovie)
```

You may want to poll the movie to see if it's done playing, particularly for music movies, when you may want to queue up another music movie right away. The following call returns TRUE if a movie is still playing and FALSE if the movie is finished.

```
Boolean IsMovieDone(Movie theMovie)
```

Other Useful Functions

Here is a small collection of QuickTime functions that are useful in our example, which we'll get to in a moment.

The following function is called during the event loop of your application. QuickTime needs to be called regularly with the `MoviesTask()` function to service the movie once it's playing. With music tracks, as much as several seconds may elapse safely between calls, but it's good practice to call `MoviesTask()` as often as you'd call `WaitNextEvent()` for your event servicing. If your movie contains video, call `MoviesTask()` as frequently as you can.

```
void MoviesTask(Movie theMovie, long maxMilliSecToUse)
```

`theMovie` is a valid Movie pointer.

`maxMilliSecToUse` is the limit of how long your application wants QuickTime to spend servicing the movie. Set this value to 0 to let QuickTime decide the best servicing rate for the movie.

The next step is important: QuickTime preloads movie data to ensure a smooth start when you play it.

```
OSErr PrerollMovie(Movie theMovie, TimeValue time, Fixed Rate)
```

As usual, `theMovie` is a valid Movie pointer.

`time` is the start time within the movie we're playing.



`Rate` is a fixed point value specifying the playback rate (relative to the rate at which the movie was recorded). A fixed value of 1.0 (0x10000) would be used for normal speed forward playback. You can also specify negative values for backwards playing.

```
void SetMovieActive(Movie theMovie, Boolean active)
```

Guess what `theMovie` is? That's right, a valid Movie pointer.

`active` is TRUE to activate this movie, FALSE to deactivate this Movie.

Code Example

This code example assumes the QuickTime movie has no video associated with it. It converts a MIDI file into a movie file, then plays it.

```
void StartQuickTimeMovie(void)
{
   SFTypeList        theTypeList;
   StandardFileReply theReply;
   short int       myRef;
   Rect          wRect;
   FSSpec        theMidiFile;
   OSErr         theErr;

   /* Let the user select a movie */
   theTypeList[0] = MovieFileType;
   theTypeList[1] = 'Midi';
   StandardGetFile(NULL, 2, theTypeList, &theReply);
   if (theReply.sfGood)
   {
      /* initialize Movie Toolbox */
      theErr = EnterMovies();
      if (theErr == noErr)
      {
         if (theReply.sfType == 'Midi')
         { // Then we are converting a MIDI
            // file into a QuickTime movie

            theMidiFile = theReply.sfFile;
            StandardPutFile(NULL, theReply.sfFile.name,
                     &theReply);
            if (theReply.sfGood)
            {
               theErr = ConvertFileToMovieFile(
                     &theMidiFile,
                     &theReply.sfFile,
                     'TVOD', 0,
                     NULL,
showUserSettingsDialog,
                     0L, NULL, 0L);
```

```
        }
        else
        {
          theErr = badFormat;
        }
      }
      /* open the movie file, get the movie from it, then close it */
      if ( (theErr == noErr) &&
        (OpenMovieFile(&theReply.sfFile, &myRef,
                  fsRdPerm) == noErr) )
      {
        if (NewMovieFromFile(&theMovie, myRef, NULL,
                  NULL, newMovieActive,
                       NULL) == noErr)
        {
          /* Find out the size of the movie. */
          GetMovieBox (theMovie, &wRect);


          /* Tell QuickTime we're playing from the beginning. */
          GoToBeginningOfMovie(theMovie);

          /*
          PrerollMovie() helps QuickTime preload
            parts of the movie to give smoother
            playback.
          Second parameter is the time we expect
            to start playing at.
          Third parameter is the playback rate we
            expect to be using.
            1.0 in fixed rate. */
          PrerollMovie(theMovie, 0, 0x10000);

          /* Start playing forward. */
          StartMovie(theMovie);
        }
        CloseMovieFile(myRef);
      }
    }
  }
}
```



Requesting Instruments and Playing Notes

This section deals with how to control a musical instrument in QuickTime. It includes API descriptions of the relevant part of QuickTime used for allocation of musical instruments, and a simple example program that plays a musical scale with several different instruments.

QuickTime Music enables applications to allocate instruments from the pool of available musical devices. The basic data structure used for choosing an instrument is called a `ToneDescription` record. Its format is as follows:

```
struct ToneDescription
{
   OSType synthesizerType;  /* synthesizer type */
   Str31 synthesizerName;   /* name of instantiation of synth */
   Str31 instrumentName;    /* preferred name for human use */
   long instrumentNumber;   /* inst number used if synth name matches */
   long gmNumber;           /* Best matching General MIDI number */
};
typedef struct ToneDescription ToneDescription;
```

To open an instrument and make it available for your use, you want to fill out a `NoteRequest` record, which incorporates the `ToneDescription` record:

```
struct NoteRequest
{
   UInt8 flags;             /*1: don't accept GM match, 2: don't accept same-synth-➥
type match */
   UInt8 reserved;          /* must be zero */
   short polyphony;         /* Maximum number of voices */
   Fixed typicalPolyphony;  /* Hint for level mixing */
   ToneDescription tone;
};
typedef struct NoteRequest NoteRequest;
```

You can fill out as many or as few of these fields as needed to describe your instrument to QuickTime. For example, you may want to force QuickTime to use the Software Synthesizer for the note channel. To do this, fill out the `synthesizerType` record as follows:

```
myNoteRequest.tone.synthesizerType = 'ss  ';      /*select Software Synth */
```

Note

Constants for the `synthesizerType` field are not defined explicitly in Apple's headers and are unique to the particular synthesizers supported. Normally, leave this field zeroed out by passing `0L` (zero long) and let QuickTime choose the best synthesizer. In this example, we are passing the 'ss' type because we want to force it to select the software synth.

Note

When setting a `ToneDescription` record, set the `polyphony` field to 1 if you're allocating instruments for the Software Synthesizer, and plan to load many instruments at the same time; otherwise, the Software Synthesizer may run out of voices. In QuickTime 2.0, the Software Synthesizer is limited to a total polyphony of 24, but allocates available voices dynamically, so the polyphony field isn't relevant to how instruments are actually played.

Once you've filled out the fields, you're ready to allocate a note channel:

```
ComponentResult NANewNoteChannel(NoteAllocator na,
            NoteRequest *noteRequest,
            NoteChannel *outChannel);
```

`na` is the `NoteAllocator` component we're opening.

`noteRequest` is a valid note request structure for this new channel.

`outChannel` is a pointer to the new channel allocated.

Later, when you're done, call this:

```
ComponentResult NADisposeNoteChannel(NoteAllocator na,
            NoteChannel noteChannel);
```

`na` is the `noteAllocator`, and `noteChannel` is the pointer to the channel, which we opened with `NANewNoteChannel()`.

Once the note channel has been created, make the following call to make sure all the needed resources have been brought into RAM:

```
ComponentResult NAPrerollNoteChannel(NoteAllocator na,
                  NoteChannel noteChannel);
```

Playing Notes with Your Note Channel

Now that the note channel is ready, we can play musical notes and events through it. The most important function is note playing, obviously. Here's the call to play a note:




```
ComponentResult NAPlayNote(NoteAllocator na,
            NoteChannel noteChannel,
            long pitch,long velocity)
```

`na` is the `NoteAllocator` component we've opened.

`noteChannel` is the previously opened note channel, and `na` is its `NoteAllocator`.

`pitch` is the pitch at which we want to play the note. 0-127 are traditional MIDI note pitches. 0x100 to 0x7F00 specify a note pitch in fixed point, with 8 bits of whole note and 8 bits of fractional note pitch precision. This resolution enables very accurate control over the actual note pitch.

`velocity` stands for the velocity with which the note is struck, from 1 to 127.

To stop a playing note, issue another `NAPlayNote()` command with the same pitch and a velocity of 0.

Controllers

QuickTime Music supports controllers for various capabilities. In QuickTime version 2.0, the software synthesizer supports only the pitch bend and sustain controllers. Later versions are expected to support many different controller types.

The call to set a controller value is as follows:

```
ComponentResult NASetController(NoteAllocator na,
            NoteChannel noteChannel,
            long controllerNumber,
            long controllerValue);
```

`na` and `noteChannel` are as usual—the note won't play if these have not been opened.

`controllerNumber` specifies which controller we wish to change. The controller numbers are roughly the same as the MIDI controller numbers.

The control numbers include:

- `kControllerModulationWheel`—To add pitch modulation to the currently playing notes. The lower 8 bits specify 256ths of a semitone and the upper 8 bits represent a semitone of modulation to swing the pitch.
- `kControllerBreath`—Breath controller. How much breath effect applied. Negative values are air in and positive values are air out.
- `kControllerFoot`—On or off to sustain the current notes.
- `kControllerPortamentoTime`
- `kControllerVolume`—To change the volume of the current notes.
- `kControllerBalance`—To change the left and right balance. Negative values for left and positive values for right.
- `kControllerPan`—To change the stereo panning position. Passing a 0 is the default position, -128.0 is full left, and 127.7 is full right.

- `kControllerPitchBend`—Pitch bend the current notes playing. The lower 8 bits specify 256ths of a semitone and the upper 8 bits represent a semitone of bend.
- `kControllerSustain`—A Boolean value, positive values for on and 0 or negative for off.
- `kControllerPortamento`—A Boolean value, positive values for on and 0 or negative for off.
- `kControllerSostenuto`—A Boolean value, positive for on and 0 or negative for off.
- `kControllerSoftPedal`—A Boolean value, positive for on and 0 or negative for off.
- `kControllerReverb`—Controls reverb. Ranges from -128.0 to 127.7. The larger the number, the more depth the reverb effect has on the playing notes.
- `kControllerTremolo`—Controls note tremolo. Ranges from -128.0 to 127.7. Larger numbers represent a faster tremolo or vibration of the notes.
- `kControllerChorus`—Controls chorusing effects.
- `kControllerCeleste`—Controls the celeste effect for the current notes being played.
- `kControllerPhaser`—Controls the amount of phase effect applied the current notes.

`controllerValue` is the new value for the controller. The controller value's range is -128.0 to 127.7 signed fixed point. All controllers default to 0 except volume. The constants `kControllerMaximum` and `kControllerMinimum` are defined for the maximum and minimum values possible for a controller.

Code Example Using the Note Allocator

This example code uses all the functions described to open the note allocator and play a list of instruments through a sliding scale.

```
void PlayQuickTimeNotes(void)
{
   NoteAllocator theNoteAllocator;  /* The music note allocator component */
   NoteChannel myNoteChannel;       /* A channel to play notes through */
   NoteRequest myNoteRequest;       /* A structure describing the instrument we want ➥
to play */
   short int i, velocity, instrument_index;
   unsigned long current_tick_count;
   /* This is a list of General MIDI instrument numbers we want to play. */
   static int instrument_list[6] = {1, 5, 7, 8, 12, 13};

   /* Open the note allocator component */
   theNoteAllocator = OpenDefaultComponent(kNoteAllocatorType, 0);

   /* Walk through our list of available instruments. */
   for (instrument_index = 0;
```



```
          instrument_index < 6; instrument_index++)
   {
      /* Set up a NoteRequest structure with a request for a piano with polyphony ➥
of 1. */
      myNoteRequest.polyphony = 1;
      myNoteRequest.typicalPolyphony = 1<<16;

      /* We can request a specific instrument by name. If it's not found, the ➥
gmNumber field provides the General MIDI fallback instrument. */
      myNoteRequest.tone.synthesizerType = 0;
      myNoteRequest.tone.synthesizerName[0] = 0;
      myNoteRequest.tone.instrumentName[0] = 0;
      myNoteRequest.tone.instrumentNumber = 0;
      myNoteRequest.tone.gmNumber =
             instrument_list[instrument_index];

      /* Allocate a note channel with our desired instrument as filled out in the ➥
NoteRequest */
      NANewNoteChannel(theNoteAllocator, &myNoteRequest,
               &myNoteChannel);

      /* Load the resources needed by the instrument */
      NAPrerollNoteChannel(theNoteAllocator, myNoteChannel);

      /* Perform a loop, playing notes at increasing pitch and velocity, using ➥
TickCount()as a timebase. You can play notes at interrupt time, so you can ➥
schedule note events with the Time Manager, or through your own schemes. */
      current_tick_count = TickCount() + 6;
      for (i = 32; i < 90; i += 3)
      {
         velocity = i + 10;
         /* Play the note */
         NAPlayNote(theNoteAllocator, myNoteChannel,
                  i, velocity);

         /* Wait a while */
         while (current_tick_count > TickCount()) {};
         /* Prepare to do this again in another 1/10th of a second. */
         current_tick_count += 6;

         /* Stop playing the note. */
         NAPlayNote(theNoteAllocator, myNoteChannel, i, 0);
      }

      /* Wait for the last note to decay away to silence. */
      current_tick_count += 20;
      while (current_tick_count > TickCount()) {};
```

```
        /* Dispose of the note channel when we're done with it. */
        NADisposeNoteChannel(theNoteAllocator, myNoteChannel);
    }
    CloseComponent(theNoteAllocator);
}
```

QuickTime Software Synthesizer Instruments

Below are listed the instruments supported by the software synthesizer. Please note that the software synthesizer is not General MIDI-compliant, but that the sound of the instruments, whose samples were supplied to Apple by Roland, is intended to approximate General MIDI for most purposes.

Program	Instrument
1	Acoustic Piano
5	Rhodes Piano
7	Harpsichord
8	Clavichord
12	Vibraphone
13	Marimba
17	Drabar Organ
21	Reed Organ
25	Acoustic Nylon Guitar
31	Distortion Guitar
34	Electric Finger Bass
37	Slap Bass
41	Violin
48	Timpani
49	String Ensemble
53	Choir Aahs
56	Orchestra Hit
57	Trumpet
61	French Horn
66	Alto Sax
69	Oboe

continued



72	Clarinet
74	Flute
76	Pan Flute
79	Whistle
82	Lead 2 (Sawtooth)
87	Lead 7 (Fifths)
89	Pad 1 (New Age)
90	Pad 2 (Warm)
91	Pad 3 (PolySynth)
105	Sitar
106	Banjo
114	Agogo
115	Steel Drums
118	Melodic Tom
120	Reverse Cymbal
123	Seashore
124	Bird Tweet
125	Telephone Ring
126	Helicopter
128	Gunshot
16384	Standard Rhythm Set
16385	Room Rhythm Set

Example Code Using the Sound Manager, QuickTime, and Hollywood

The best test of an API is in a product. The stress of shipping is omnipresent. Parts of Hollywood have been in products, so it should hold up under the pressure. The test application on the CD-ROM, called Hollywood API Tester, uses Hollywood for all access to the Sound Manager, and the QuickTime examples use just QuickTime.

The examples on the CD-ROM have all the source code to Hollywood, plus an example application that plays with various parts. It shows memory management, playback of streamed audio, multichannel sound, and 8-bit and 16-bit mono and stereo playback all at once.

The application Hollywood API Tester has all the basic Macintosh application stuff in it, but the functions to check out are:

- `PlayFile()`—Plays an AIFF file of any size. There are a few AIFF files included on the CD-ROM.
- `SaveAIFFFile()`—Loads in a custom sample and saves it as an AIFF file.
- `PlaySoundEffect1()` and `PlaySoundEffect2()`—Shows multichannel playback.
- `PlaySoundEffect3()`—Loads and plays a custom sample, then MACE-compresses and plays it.
- `PlaySoundEffect4()`—Also loads in a custom sample, creates a 'snd ' resource, and plays that.
- `PlayQuickTimeNotes()`—Plays just notes with the QuickTime Music Architecture.
- `StartQuickTimeMovie()`—Plays a QuickTime movie, either video or musical.

So from all of that we have Hollywood. See you on the Walk of Fame!

Debugging Audio

Debugging audio is difficult at best. But there are a few tricks that can help you figure out what is going wrong. Most of the useful Sound Manager code and callbacks require interrupt-level code, which is impossible to debug with high-level source debuggers. There are a few exceptions, but the best way currently is to become familiar with MacsBug, a trusty low-level debugger from Apple.

Typically, if you don't hear sound when you should, check the sample rate. The Sound Manager expects an unsigned fixed value that is a 32-bit number, but the lower 16 are to the right of the decimal point. Hence, 22050.0 KHz would be represented as `22050 << 16L`.

If you hear static when you play a sound, a few things could have gone wrong. If the sound object is MACE-compressed and you didn't use `SndPlay()` to play back the sound, then you must decompress it yourself. The `bufferCmd` does not do this for you.

The other possible error is trying to play a 16-bit sample without setting the appropriate extended header information for 16-bit output. Static occurs if you try to play 16-bit data through an 8-bit setup or 8-bit data through a 16-bit setup.



Passing the wrong length of sample data to sample playback functions sometimes creates surprising results. Make sure that the length is correct.

Working with callbacks can create havoc if you're on a 68K-based Macintosh and access global data without setting up your A5 world. If this is not done, you may be setting a memory location in some other system space; if it doesn't crash now, it will very soon.

Where to Go from Here

Sound creation is an art form, just as graphics and pure programming style have become art forms. To bring an exciting experience to the game, you must decide early on to devote some time and energy into creating the audio experience. The better quality games built today devote up to one-third of the total time and cost to the game's audio.

Already it's beginning. Games are getting larger and more lavish, and the hardware is better. Talented musicians, sound effects people, and audio editors from Hollywood are going to be involved in games more and more. With that infusion of talent into games, the player is going to get a better quality experience that is more exciting. With the tools and techniques presented in this chapter, your games can be a part of the overall advancement of game sound quality. I can't wait.

Glossary

AIFF—Audio Interchange File Format. This is a file format defined by Apple to exchange audio data in a platform-friendly way.

AIFF-C—Same as an AIFF but with compression.

Asynchronous sound—Sound that starts and plays without stopping the CPU.

Bit depth—How many bits per sample.

Channel—How many channels of audio. Mono is one, stereo is two.

Chorusing effects—A sound effect process that combines two sounds that are exactly the same but start at slightly different times.

DAC—Digital Audio Converter. This is a chip to convert digital amplitude signals into analog signals to drive a speaker.

Digital clipping—What occurs when values exceed the bit depth of a sample. They sometimes mathematically wrap, causing clicks or pops.

Instrument—A special data structure that defines how sample data should work together to form an actual musical instrument for QuickTime Music.

Polyphony—The number of different voices at once.

Reverb—A sound effect process that sounds like a large hallway.

Sample—One-time slice of an analog volume level.

Sample rate—The number of times per second that a digital volume level is played back or recorded.

Stereo placement—Spatial position of a sound between two speakers.

Synchronous sound—Sound that starts and stops all other CPU actions.

Tremolo—The sound of the pitch of a sound moving slightly up and down. Violins are especially identified by their tremolo traits.





Chapter 7
Game Mechanics: Environments and Entities

This book has a lot to say about graphics, sounds, music, and input/output. However, all that gives you is a pile of building blocks. To turn that into a game, a lot of code is required to link all the parts together and to describe when and how each of the building blocks is used. That code is the code for the *game mechanics*.

The game world description and the game entities are essential parts of the game code. How complex should the game world be? How should it be structured? How should the game entities be described, and how should they be stored? How should the game entities interact?

There are no rules written in stone on this subject. There are many ways to solve each problem, and which one to use depends on the game. For example, there are big differences between the representations needed by a Reversi game, a Lunar Lander, and an adventure game.

In this chapter, I suggest common ways to implement game worlds and game entities, as well as ways to handle them. My suggestions may be useful for your game, but there surely are games for which my suggestions just don't apply. I propose solutions that are common in existing games, and some that are not so common but that may be useful—but it is up to you to adapt them to your own problems and pick the right solution for your game.

A Game's "Virtual Reality"

Every game has some kind of "world," the virtual reality it describes, in which entities bound by the rules of this virtual reality interact. No, I'm not just referring to realistic 3-D games and adventure games. Even though the term "virtual reality" may seem ambitious, I would consider even Tetris to have one. Tetris is a world of blocks in a discrete world, in which each block is bound to the grid. There is a kind of gravity that forces the pieces downward. You manipulate the blocks to achieve a goal, and you are limited by the laws of "nature" in that world.

While the Tetris world is abstract and has no similarity to reality, many games present much more realistic worlds. Adventure games may not look real (though they often do), but the events taking place in it should be reasonable. Another "virtual reality" occurs in a card game. In this case, the game is actually a virtual representation of a subset of the real world—the card table! Since the game is simulating a part of the real world, the laws of the game world should make it as realistic as possible. In games without that demand, the laws may be totally different.

The collection of laws of the game world is what I call game mechanics. Game mechanics includes how we represent the world, how we represent its inhabitants (be they monsters, vehicles, or abstract objects), and how they move and interact.

World Representation

There are many ways to represent the game world, and which one to use obviously depends on the game. A simple game needs only a simple representation; a complex game, one that simulates a bigger subset of reality (or nonreality), needs a detailed representation.

The world representation can be used for two tasks:

- Presenting the world graphically
- Interacting with game entities (e.g., sprites)

Some representations are used only for the latter of these two, for interacting with game entities. For example, the network world in an adventure game is rarely used for drawing graphics.

It is also conceivable to make games in which some background image is built from a description that is a kind of world description, but not used as such. As an example, take the mountains in the background of Moon Patrol. These mountains, drawn as polygons, move relative to each other to enhance the feeling of movement. However, I would consider that merely an image description, not a world representation. A world representation is used for telling where game entities may or may not go, what the move's costs are, where entities can do certain things, and often how game entities can find each other. The mountains in Moon Patrol do not affect the game mechanics.

Next, I want to distinguish between unstructured and structured world representations. Many simple games can use unstructured world representations, in which there are no structures that really describe the world. If you want a more complex game, you should probably use some structured representation.

Unstructured World Representations

Many game worlds are not at all described in any structured form. The world is in the program code rather than on some map.

Open Space

The simplest world you can have is "open space." By this I don't necessarily mean space games, but games in which there are no obstacles. All moving game entities can move anywhere, except perhaps onto each other. The world is an open continuum. A classic game with an open space world is Asteroids. The whimsical Cheese Toast game appears on the CD-ROM with accompanying code (in the C/C++ folder within the More Source folder), and though you pilot a tea creamer rather than a spaceship, it is otherwise a "classic" open space game.

Stationary Objects

A game using open space can be transformed into a game with a far more complex world simply by making some game entities work as stationary obstacles. Space Invaders is perhaps the simplest game with a stationary object-based world: The world is an open space except for the shelters.

You can build a world based on stationary objects that is fairly complex. For example, you can make a platform game in which every platform is a stationary object. You can also put in walls, slopes, and movement-constraining areas.

Custom Movement Constraints

Another way to turn open space into a more complex world is to constrain movement for game entities. Again, Space Invaders is the simplest example. While most sprites can move just about anywhere (i.e., the missiles), the player ship/gun/whatever is constrained to move sideways along the bottom of the screen. This gives us the impression that the player sprite moves on some kind of floor. Even though there is no easily accessible description of this world, the world representation is still there, in the program code.

Obviously, you can make a simple game based on custom movement constraints, but once it gets just a bit more complex than Space Invaders, the game gets unbearable. Then you need to replace the *custom* constraints by *general* constraints, through a structured world description. You could again turn to stationary objects for this, but although that makes for a fairly workable representation, it's hard to grasp the total layout.

Structured World Representations

The unstructured world representations are just fine for simple worlds, but games with complex worlds need a structured representation—that is, some structure to which all objects can turn to see whether or not they can move to a certain position or take some specific action.

The Grid-Based World

The most common world representation, which you find in a great many games, is the grid-based world. Most board games and strategy games use it, as do many arcade games.

In a grid-based world, the environment is represented by an array, typically a two-dimensional array. Each space in the grid corresponds to an item in the array. Each item may be 1 byte (e.g., char), a word (`int`), or even a whole data structure. You may also use several arrays of the same size, as I did in Dungeon.

Dungeon is obviously a grid-based game, using a grid of 15 x 12 tiles, each drawn as 32 x 32 pixels (see figure 7.1). That makes the total game area (32 x 15) x (32 x 12) pixels = 480 x 384 pixels. Thus, I don't use the entire screen, although I could if I want to. A game using a full 640 x 480-pixel screen could fit a 20 x 15 grid with the same tile size. Dungeon 1 would fit on a Color Classic screen if the window were positioned for it.

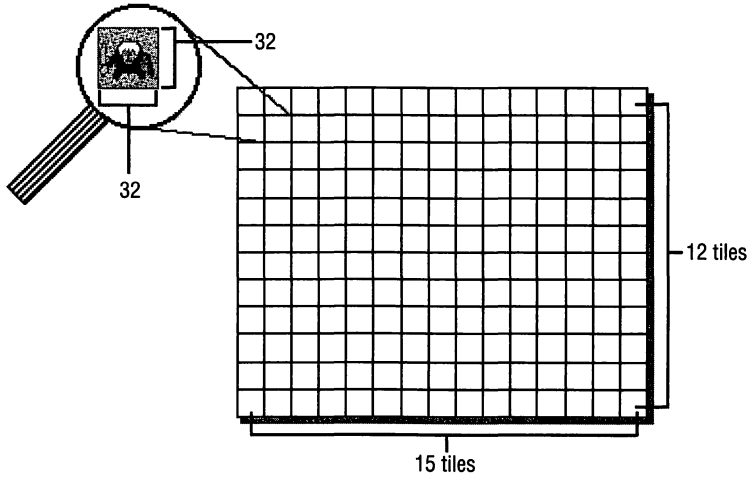


Figure 7.1 *The 15 x 12 grid that represents the game world in Dungeon.*

The grid is represented by two arrays in memory, the array `tileKnown` and the array `tileArray`. They are defined by the following code:

```
/*Size of the array*/
#define kArraySizeH 15
#define kArraySizeV 12

/*Size of the tiles*/
#define kTileSizeH 32
#define kTileSizeV 32

/* What tiles have we seen? */
Boolean tileKnown[kArraySizeH][kArraySizeV];
/* What does each tile contain? */
TileState tileArray[kArraySizeH][kArraySizeV];
```

Note

I use constants for both the size of the array and the size of the tiles. You should always design your programs so that they are easy to modify.

An exception to this is an array of strings, which is very practical when editing new levels. You can also use readable characters for all valid values. In that case, a level can be stored on disk, perhaps in a 'TEXT' resource editable in ResEdit. That is what I do in MacSokoban, my own implementation of the Sokoban puzzle game. The first Sokoban level looks like figure 7.2.

```
    #####
    #   #
    #$  #
  ###  $##
  #  $ $ #
### # ## #   ######
#   # ## #####  ..#
# $  $          ..#
##### ### #@##  ..#
    #     #########
    #######
```

Figure 7.2 *The text representation of the first level in the grid-based game Sokoban.*

When someone plays the game, the level is drawn using an icon corresponding to each character (see figure 7.3).

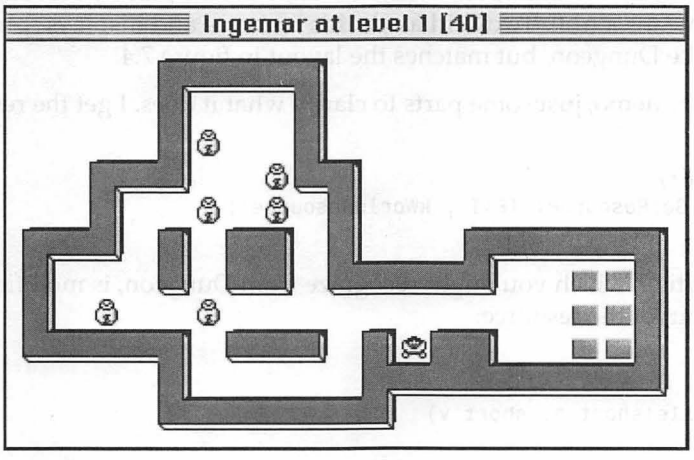


Figure 7.3 *The first level as it appears in MacSokoban.*

In a commercial game, you should not demand that users edit levels in a representation like the one in figure 7.2, but it is perfectly fine for a freeware game like MacSokoban or for your own editing.

In the latter case, when you edit the levels yourself and users are not supposed to edit them, you can even add some constraints; for example, you can demand that each line in the text resource is exactly, say, 32 characters including return. This will make your programming simpler (at least in the short term—if you someday decide to change the limits, you may find that you should have made the program more flexible from the start).

The demo TextGrid (in the 07 - Mechanics: Environments folder on the CD-ROM) demonstrates how you can do this. It draws the graphics from Dungeon in a layout that is determined from 'TEXT' resource number 128. In the resource, the levels look almost like they do in Sokoban (see figure 7.4).

```
###############
#@  ###########
#   ### $$#####
### ###   #####
### ###   #####
### ### #######
### ### #######
###     ### X##
### ## ####$ ##
##  ## #### ###
##* ##    * ###
###############
```

Figure 7.4 *The level layout from the TextGrid demo.*

Much like Sokoban, space means open, `#` means wall, `@` means player, `$` means treasure, `X` means exit, and `*` means a troll. TextGrid loads this resource and displays something that looks very much like Dungeon, but matches the layout in figure 7.4.

I won't list the entire demo, just some parts to clarify what it does. I get the resource with `GetResource()`:

```
/*Get the resource*/
  gWorldHandle = GetResource('TEXT', kWorldResource);
```

The `DrawTile()` routine, which you might recognize from Dungeon, is modified to get its information right out of the resource:

```
/* Draw a tile */

static void DrawTile(short h, short v)
```

```
{
   Rect tileRectangle;
   char theTile;

   SetRect(&tileRectangle, h * kTileSizeH, v * kTileSizeV, (h + 1) * kTileSizeH, ➥
(v + 1) * kTileSizeV);
```

Here is the important change: I calculate an offset into the resource and get the character at that position. The `+1` to `kArraySizeH` is a compensation for the return character at the end of each row. Obviously, this requires all rows to have exactly the same length!

```
   theTile = *(*gWorldHandle + h + v * (kArraySizeH+1));
```

Then, I just use `theTile`, the character I got out of the resource, for determining what picture to display.

```
   switch (theTile)
   {
      case ' ':
         DrawPicture(floorTile, &tileRectangle); break;
      case '#':
         DrawPicture(wallTile, &tileRectangle); break;
      case '@':
         DrawPicture(playerTile, &tileRectangle); break;
      case '*':
         DrawPicture(enemyTile, &tileRectangle); break;
      case '$':
         DrawPicture(goldTile, &tileRectangle); break;
      case 'X':
         DrawPicture(exitTile, &tileRectangle); break;
      default:
         PaintRect(&tileRectangle);
   }
} /*DrawTile()*/
```

Are you ready for a little programming assignment? Plug the level handling in TextGrid into any of the Dungeon incarnations (for example, Dungeon 2), replacing the random level generation with a number of pregenerated levels.

Dungeon 1, like many other grid-based games, restricts movement to grid positions only. This is not, however, a necessary restriction. Some games (e.g., Civilization, Pac Man, MacSokoban) make smooth animation between the grid positions, but you still must move from one grid position to the next. Other games, such as Oxyd, have underlying grids that restrict motion in some positions and allow free movement in others.

The Network World

A completely different world representation is the network world description. In such a representation, the world is a set of nodes, and each node has a number of pointers to other nodes. These pointers may or may not go both ways between the nodes.

The most obvious use for this kind of representation is in adventure games—we may represent each room by one node, and the pointers to other nodes (rooms) are the possible exits from the room. Figure 7.5 shows a simple example.

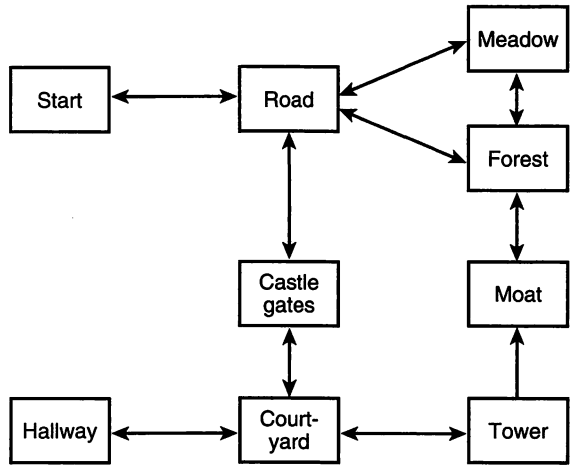


Figure 7.5 *Adventure games often use network world descriptions.*

Note that most rooms have arrows going both ways, while there is one one-way arrow from the tower to the moat. That makes sense, because you can't go directly back up into the tower after jumping from it.

Adventure games generally use this kind of world description. However, graphical adventure games tend to combine it with some other description *within* each room. Take the Sierra games, such as SpaceQuest, as an example. There's definitely a network description of the connections from room to room, but there's also another description of each room, possibly a grid-based description.

The network descriptions are not just for adventure games, though! You most definitely can use network descriptions for arcade games. You can replace a grid-based description with a network description in which each network node holds a rectangle or polygon and pointers to neighboring nodes.

In such a description, each node can describe either an obstacle or an open space. If nodes describe open spaces and everything else is an obstacle, the representation can be used for a platform game (see figure 7.6)

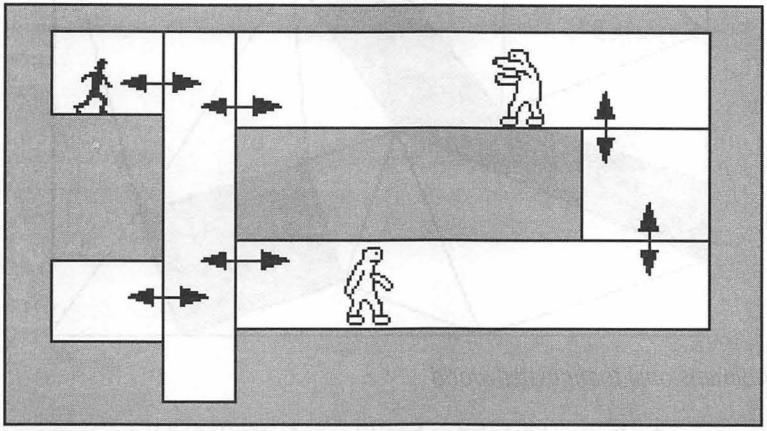


Figure 7.6 *A set of rectangles linked to a network, which can be used for representing the open spaces in a platform game.*

In this example, each game entity keeps track of which node it's currently in. As long as its moves keep it within the rectangle of that node, the moves are accepted. If the entity moves outside the rectangle, it checks all neighboring nodes (the ones its current node has pointers to) to see if it is inside one of them. If it is, the entity sets that node as the current node. If not, the entity moves so it's inside the rectangle of the original node again.

As with everything else in this chapter, I don't claim that this is the only way or the best way to solve the problem—in this case to make a platform game. A platform game can just as well be made using a stationary object world, in which case including both moving and stationary platforms in the game comes naturally. A platform game could also use a grid-based world, which has the clear advantage of making it is easier to design new levels.

The Polygon World

Yet another representation is the polygon world. Your world can be represented by a set of lines. These lines are often connected in some way, so you can easily find a line's neighbors. A not-too-uncommon representation is to break down the world into triangles. Then your representation can be a set of nodes, with each node pointing out all the nodes to which it has a branch.

Figure 7.7 suggests one way to build scenery in a polygon world.

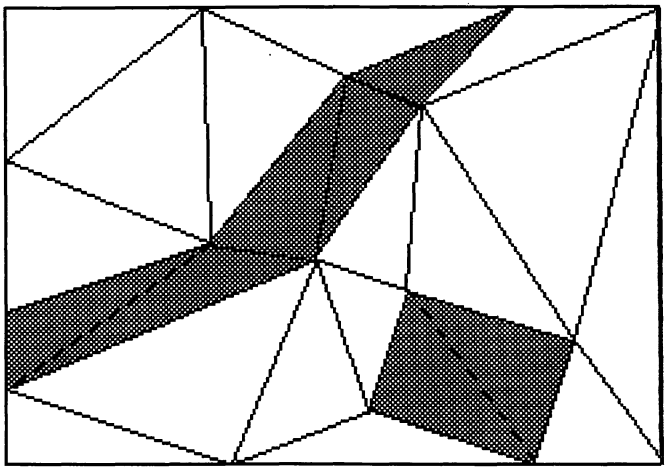


Figure 7.7 *A two-dimensional triangulated world.*

In this figure, a road and a house have been built from triangles. All areas around it are also filled with triangles in order to connect all parts of the world. If every vertice has a height, a 3-D image can be rendered from this model. This kind of representation, in one form or another, was common in the 3-D vector graphics games of a few years ago, such as car racing games (e.g., Vette).

What's the point? Isn't a grid easier?

Yes, usually it is, but the good thing about a polygon-based world is that it isn't sensitive to the orientation of the X and Y axis. The geometry of a grid-based game is often distorted by the grid. In a polygon-based world, you are free to place anything anywhere, in any rotation.

For an example from the recent 3-D games, let's compare Wolfenstein 3D with Marathon. Wolfenstein 3D is a grid-based game, and that is noticeable. Everything is square; the whole world is built from cubes. Marathon is polygon-based, at least to some extent. I have no insights as to how it was implemented, but when you look at the map, doesn't it strike you that it looks very much like polygons?

Distance in a Game World

The grid-based world is the most common world, and from both the programmer's and player's point of view, also the easiest to use. However, the grid creates some limitations and problems of which you should be aware.

When it comes to display, every computer game is grid-based—every pixel has a horizontal and vertical component—so to one extent or another, every game must compromise "ideal" position for what is possible. Even an "open space" game needs to calculate `newPosition.h` and `newPosition.v` for every object in motion. Grid-based worlds are merely the simplest example of a universal game programming concern.

In this section, I discuss how distances work in grid-based worlds. There are two problems to consider. The first is how to compensate for or reduce the distortion of distance when moving game entities in a grid. The second is how to find the shortest path in a grid.

The Chessboard and City Block Metrics

If your game world is grid-based, the concept of distance often gets distorted. Let's take a board game—chess, for example. For the king, which can move one space per move in any direction, the number of moves it takes to go along one side of the board is the same as along a diagonal. This is what is often referred to as Chessboard distance (see figure 7.8).

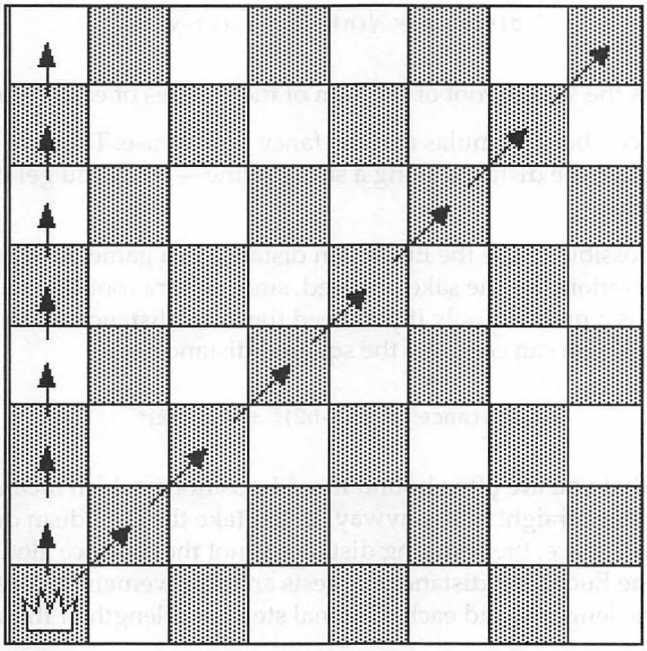


Figure 7.8 *For the king in a chess game, the two distances are the same, while for us, the diagonal move is over 40 percent longer!*

Chess players accept Chessboard distance as part of the rules, but for a strategy game with some ambition of being realistic, it is a clear weakness. A king in the lower left square of a chessboard can reach the top or right edge of the board in seven moves by moving horizontally or vertically, but it can reach both the top and right edges of the board, the top right square, in the same number of moves by moving diagonally. In the real world, there is no speed advantage to moving diagonally, while in Chessboard distance, diagonal speed is approximately 1.4 times (the square root of 2, to be exact) faster than vertical or horizontal speed.

City Block distance is just as unrealistic, only in City Block distance, diagonal moves are penalized instead of rewarded. If you are standing at the corner of 1st Avenue and A Street and you want to go to the corner of 2nd and B, you're only one diagonal move away from your goal—but to get there, you need one vertical move up 1st Avenue to B Street, then one horizontal move along B Street to 2nd Avenue.

Pseudo-Euclidean and Euclidean Metrics

The distance we perceive in the real world is called Euclidean distance. To get the distance between two points (h1, v1) and (h2, v2) in Euclidean distance, you calculate it like this:

$$\text{distance} = \sqrt{(h1-h2)^2 + (v1-v2)^2}$$

That is, the distance is the square root of the sum of the squares of each component.

Now, don't get worried about formulas and the fancy terms I use. They are merely a mathematical way to calculate the distance along a straight line—what you get if you measure the distance with a ruler.

It's often hard or impossible to use the Euclidean distance in a game. First, you will want to avoid square root operations for the sake of speed, since square root calculations demand a lot of CPU time. This is a problem only if you need the exact distance. If you just want to compare two distances, you can compare the squared distance:

$$\text{distance}^2 = (h1-h2)^2 + (v1-v2)^2$$

A bigger problem is that you are often bound to grid positions, which means that the game entities can't travel along straight lines anyway. If you take the Euclidean distance along a path, grid space to grid space, the resulting distance is not the distance along a straight line. However, knowing the Euclidean distance suggests an improvement. Every vertical or horizontal step has the length 1, and each diagonal step has a length of the square root of 2, or approximately 1.4.

This is called a pseudo-Euclidean distance, which uses different distances for different directions. However, the distance 1.4 is not practical for a game, since it's a floating point value. Integers are much better, at least as long as 68K Macs are supported. On Power Macs, using integers or floats makes much less difference. See further chapter 13, "Optimizing Game Code for PowerPC."

For example, you can use the movement cost of 2 for horizontal and vertical moves, and 3 for diagonal moves. Let's return to the chessboard for a moment and see what happens (see figure 7.9).

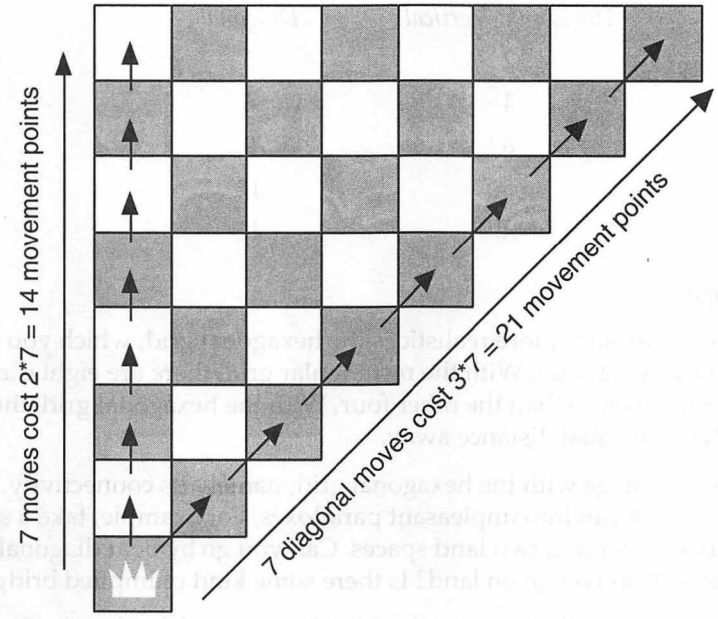


Figure 7.9 *With different movement costs for horizontal or vertical directions and diagonal directions, the distance is much closer to the real distance.*

This is a lot better! For the 14 movement points that the 7 vertical moves cost, the king can go almost 5 diagonal moves. That agrees pretty well with reality.

The conclusion: If it doesn't cost 1 to move in each direction, but rather 2 movement points in horizontal and vertical directions and 3 in the diagonal direction, you will get much closer to real distances than when using Chessboard or City Block distance.

This solution is also useful in certain arcade games. Suppose that you make a game in which you allow movement in eight directions: up, down, right, left, and all diagonals. If all moves up, down, right, and left are made three pixels at a time, and diagonal moves are made two pixels at a time, the speed will be almost the same in all directions.

Now, let's bring this one step further. I just proposed a movement in which the cost depends on the direction. Imagine a strategy game in which the movement cost depends on the terrain! That's a pretty common feature. You find it in games such as Civilization and Warlords II.

You can easily combine these two techniques and compensate for both direction and terrain. The number of movement points must be higher, of course. You can, as an example, use a table like this:

Terrain	*Horizontal/Vertical*	*Diagonal*
Road	2	3
Grassland	4	6
Plain	6	9
Hills	8	12
Mountains	10	15

Hexagonal Distance

Another way to make distance more realistic is the hexagonal grid, which you find on some board games and strategy games. With the rectangular grid, there are eight directions, four of which are closer in distance than the other four. With the hexagonal grid, there are six directions, all of them an equal distance away.

There is one more advantage with the hexagonal grid, namely its connectivity. With the rectangular grid, you can run into unpleasant paradoxes. For example, take a strategy game with four spaces: two water and two land spaces. Can you go by boat diagonally between the two water spaces? Can you go on land? Is there some kind of implied bridge there?

This problem doesn't exist in the hexagonal grid. It has none of the drawbacks of the rectangular grid. However, it has other drawbacks instead. It's a bit more tedious to put together a hex grid-based game.

Hexagonal grids will also not fit as well in a rectangular window, as seen in a screen shot of Xconq—a strategy game in the Empire family that uses the hexagonal grid (see figure 7.10).

Strategy games are games in which the hexagonal grid, as well as the pseudo-Euclidean distances I described in the previous section, are most called for. Hexagonal grids are also fun to use in other kinds of games. Some games are explicitly made for the hex grid, such as the game Susan by Stephen Linhart. Other games can be adapted to the hex grid with interesting results. Some that have already been made are Tetris (Bikaka, written by me), Mines (Hexmines, ditto), and Snake (Hexnake, by Folke Söderström).

The demo HexGrid (in the CD-ROM's 07 - Mechanics: Environments folder) demonstrates how to draw a hexagonal grid. The trick is to displace every other row (or column, if you rotate the grid differently) by half a hex.

```
/* Define constants that match our hexes */
#define kHorizontalSpacing  32
#define kVerticalSpacing 26

/* Draw a tile */

static void DrawHexTile(short h, short v)
{
   Rect tileRectangle;
```

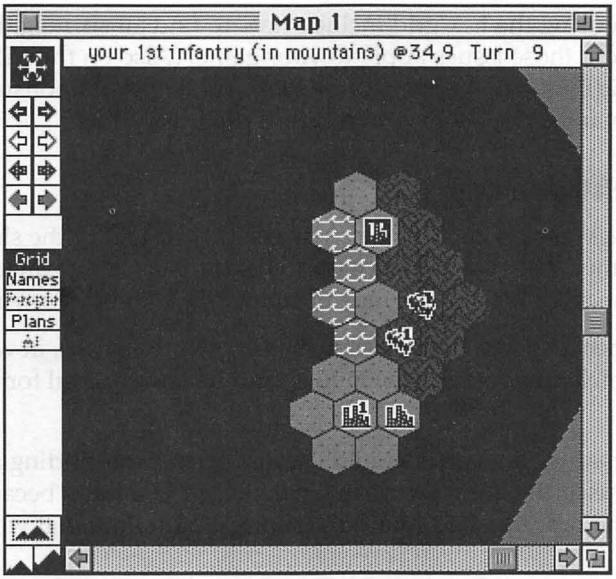


Figure 7.10 *The strategy game Xconq uses the hexagonal grid in order to make distances more realistic.*

I use the `picFrame` field of the picture, which is the bounding box.

```
  tileRectangle = (**hexTile).picFrame;
```

Just in case the PICT is not positioned right, I offset it so that it locates its top left corner at (0, 0).

```
  OffsetRect(&tileRectangle, -tileRectangle.left,  -tileRectangle.top);
```

The `tileRectangle` is now offset to the proper position. If `v` (the row number) is even, I just offset it as if it were a rectangular grid. If `v` is odd, I offset it by half a hex extra in the horizontal direction.

```
/* Offset to the proper position */
/* For every other line, offset a bit extra */
  if ((v & 1) == 0)
    OffsetRect(&tileRectangle, kHorizontalSpacing*h, kVerticalSpacing*v);
  else
    OffsetRect(&tileRectangle, kHorizontalSpacing*h + kHorizontalSpacing / 2, ➥
kVerticalSpacing*v);
```

Finally, I draw the tile in the resulting position.

```
  DrawPicture(hexTile, &tileRectangle);
} /*DrawHexTile()*/
```

Other problems when using the hex grid include how to find all six neighbors and how to handle a click in a hex, if those issues apply to your game. (Also on the CD-ROM, you can find the demo HexGridPlus, a bigger version of HexGrid, which demonstrates these problems as well.) But in many cases, a hex grid is worth the extra effort.

Finding the Shortest Path on a Grid

In many games, strategy games in particular, you often need to find the shortest path from one grid space to another. The player may wish to have a piece move unattended from one place to another, or the computer player may need the operation for moving pieces.

This problem also appears in some puzzle games (e.g., MacSokoban), in which shortest path analysis is used for quick moves using the mouse, and it can be useful for other kinds of games as well.

In fact, I can think of many games that should use a shortest path-finding algorithm that don't. Have you ever used the goto feature in Civilization? Did it fail because the piece got stuck in a detour? If the program had found the shortest path first, that wouldn't have happened.

Distance Transforms

One solution that is quick and painless is to make a so-called *distance transform* of the grid. Given an array with a zero at your destination, -1 at all obstacles, and some high value (999, in Maze—my example coming up) in all other spaces, the distance transform will fill the empty spaces with distance values to the destination. Once that is done, the game entity can walk to the destination by always picking the lowest-numbered grid in the neighborhood.

The demo Maze (also on the CD-ROM) demonstrates how a distance transform can be implemented.

The algorithm I use is a scanning algorithm. It scans through the entire array and, for each space it visits, checks if any of the neighbors has a value two or more lower than its own. If that is the case, that value plus one is copied into the space.

A value of zero denotes the destination—or destinations, if there are multiple goals. When a space next to a destination is scanned, it finds a neighbor (the destination) with a small value (zero), so the scanned space is set to that value plus one, for a total of one. That one will be found when the next space over is scanned, which will then be set to two.

This way, the value will propagate over all free spaces. The walls, by the way, are not allowed to change values.

The algorithm runs two scans at a time, one from top to bottom and another from bottom to top. Since a change in one space can carry over to the next in any number of steps, values will spread downward in the first scan and upward in the second.

Those two scans are often enough to create a complete map, but there are cases in which they're not, so the scans must be repeated until no changes occur.

Maze Source Code

Here is the source code to Maze.

The array dimensions and tile size must be defined.

```
/*Size of the array*/
#define kArraySizeH 15
#define kArraySizeV 12

/*Size of the tiles*/
#define kTileSizeH 24
#define kTileSizeV 24
```

The array is filled by values. A zero is a destination, 999 is an empty space, and a -1 is a wall.

```
/* What does each tile contain? The destination is zero
(more than one is permitted!), and the outermost entries
must always be walls (-1). Edit this to try other layouts! */

short tileArray[kArraySizeH][kArraySizeV] = {
   {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
   {-1,999,999,999, -1, -1, -1, -1, -1, -1,999, -1},
   {-1,999,  0,999, -1, -1, -1, -1, -1, -1,999, -1},
   {-1,999,999,999,999,999,999,999,999,999,999, -1},
   {-1,999, -1,999, -1, -1,999,999,999, -1, -1, -1},
   {-1, -1, -1,999, -1, -1,999, -1, -1, -1, -1, -1},
   {-1, -1, -1,999, -1, -1,999, -1, -1, -1, -1, -1},
   {-1, -1,999,999,999,999,999,999,999,999,999, -1},
   {-1, -1,999,999, -1, -1, -1, -1, -1, -1,999, -1},
   {-1, -1,999,999, -1, -1, -1, -1, -1, -1,999, -1},
   {-1, -1, -1,999,999,999,999,999, -1, -1,999, -1},
   {-1, -1, -1,999,999,999,999,999, -1, -1,999, -1},
   {-1, -1, -1, -1, -1,999,999,999,999,999,999, -1},
   {-1, -1, -1, -1, -1,999,999,999, -1, -1, -1, -1},
   {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}
   };
```

I need an array of directions:

```
/*All eight directions as vectors*/
Point directionTable[8] =
{
   {-1, 1 },
   {-1, 0 },
   {-1,-1 },
   { 0,-1 },
   { 1,-1 },
   { 1, 0 },
   { 1, 1 },
   { 0, 1 }
};
```

There's only a single picture here.

```
/* Pictures*/
PicHandle wallTile;
```

A flag changeFlag is used for the transform to signal whether there are any changes during a run.

```
/* A Boolean for signaling if a change is made */
Boolean changeFlag;
```

This time, DrawTile() draws a wall picture for a negative number, and the number itself in other cases.

```
/* Draw a tile */

static void DrawTile(short h, short v)
{
  Rect tileRectangle;
  Str255 tempString;

  SetRect(&tileRectangle, h * kTileSizeH, v * kTileSizeV, (h + 1) * kTileSizeH, ➥
(v + 1) * kTileSizeV);
  if (tileArray[h][v] < 0)
    DrawPicture(wallTile, &tileRectangle);
  else
  {
    EraseRect(&tileRectangle);
    MoveTo(h * kTileSizeH + 5, v * kTileSizeV + 18);
    NumToString(tileArray[h][v], tempString);
    DrawString(tempString);
  }
} /*DrawTile()*/
```

DrawAllTiles() draws all tiles.

```
void DrawAllTiles()
{
  short h, v;

  for ( h = 0 ; h < kArraySizeH ; h++)
    for ( v = 0 ; v < kArraySizeV ; v++)
      DrawTile(h, v);
}
```

InitToolbox() is omitted as usual.

`DistanceTransformForward()` is the first half of the algorithm. It scans the entire array, starting from the top left corner.

```
void DistanceTransformForward()
{
   short   h, v;
   short   direction;
   Point   neighbor;

   for ( v = 0 ; v < kArraySizeV ; v++)
      for ( h = 0 ; h < kArraySizeH ; h++)
      {
```

If the current tile is not a wall...

```
         if (tileArray[h][v] > 0)
         {
```

...go through all its neighbors. Actually, I only go through four of the eight neighbors here, namely the ones that have been passed by the scan before. This little optimization makes it even faster. The other four neighbors will be dealt with in the backwards scan.

```
            for (direction = 0; direction < 4; direction++)
            {
```

The point `neighbor` is set to point to the chosen neighbor tile.

```
               neighbor.h = h + directionTable[direction].h;
               neighbor.v = v + directionTable[direction].v;
```

If that neighbor is not a wall...

```
               if (tileArray[neighbor.h][neighbor.v] >= 0)
```

...and its value plus one is smaller than the value of the current tile...

```
                  if (tileArray[neighbor.h][neighbor.v]+1 < tileArray[h][v])
```

...then change the value of the current tile. Also, set the `changeFlag`.

```
                  {
                     tileArray[h][v] = tileArray[neighbor.h][neighbor.v]+1;
                     changeFlag = true;
                  }
            } // for directions 0..3
         } // if >0
      } // for all spaces
} /*DistanceTransformForward()*/
```

`DistanceTransformBackward()` is the second half of the algorithm. It works the same way as `DistanceTransformForward()`, but scans the array from the bottom right corner.

```
void DistanceTransformBackward()
{
   short   h, v;
   short   direction;
   Point   neighbor;

   for ( v = kArraySizeV ; v >= 0 ; v--)
      for ( h = kArraySizeH-1 ; h >= 0 ; h--)
      {
         if (tileArray[h][v] > 0)
         {
```

There is a small difference here: This scan, like the previous one, doesn't use all eight directions. It uses just four of the eight—the very four directions that the previous scan did not handle.

```
            for (direction = 4; direction < 8; direction++)
            {
               neighbor.h = h + directionTable[direction].h;
               neighbor.v = v + directionTable[direction].v;
               if (tileArray[neighbor.h][neighbor.v] >= 0)
                  if (tileArray[neighbor.h][neighbor.v]+1 < tileArray[h][v])
                  {
                     tileArray[h][v] = tileArray[neighbor.h][neighbor.v]+1;
                     changeFlag = true;
                  }
            } // for directions 0..3
         } // if >0
      } // for all spaces
} /*DistanceTransformBackward()*/
```

The rest is history, so to speak—or rather the main program.

```
/***************** Main program *****************/

void main(void) {
   WindowPtr myWindow;
   Rect windowRectangle;
   short   h,v;

   InitToolbox();
```

```
/*Set up the window*/
  SetRect(&windowRectangle, 50, 50, 50 + kArraySizeH * kTileSizeH, 50 + kArraySizeV ➥
* kTileSizeV);
  myWindow = NewCWindow(0L, &windowRectangle, "\pMaze demo", true, 0, (WindowPtr)- ➥
1L, false, 0);
  SetPort(myWindow);
  TextSize(9);

/* Load the picture*/
  wallTile = GetPicture(128);      /*'PICT' resource #128.*/

/* Redraw */
  DrawAllTiles();
```

The two scans are made alternatingly, as many times as it takes. If there are two consecutive scans with no changes, that is, if `changeFlag` stays FALSE, the distance transform is finished.

```
/* Repeat the distance transform until no more changes occur! */
  do
  {
    changeFlag = false;

/* Forward scan */
    DistanceTransformForward();

    DrawAllTiles();
/* Backward scan */
    DistanceTransformBackward();

    DrawAllTiles();
  }
  while (changeFlag);

/* Signal completion */
  SysBeep(1);

  while ( ! Button () )
    ;
} /*Maze*/
```

This operation is usually very fast. Scanning a game grid takes very little time. What takes time in Maze is to redraw the grid, not to scan it.

The result is the distance map shown in figure 7.11.

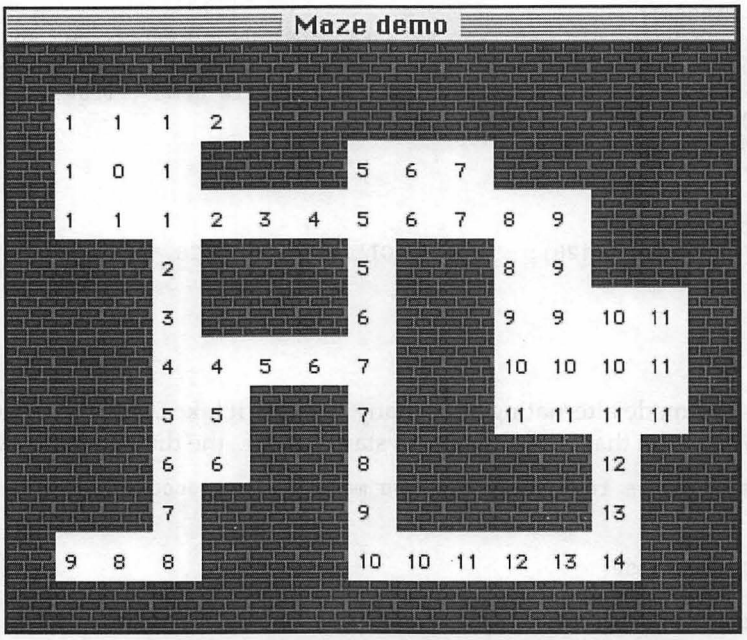


Figure 7.11 *A distance map, which is used for finding the shortest path.*

This algorithm assumes that diagonal moves are allowed. You can modify it to allow only horizontal and vertical moves by not checking diagonal neighbors. Remove the diagonal directions from the `directionTable` and modify the direction `for` loop.

So, how can this be used for finding the shortest path? That is a very simple "algorithm": Pick a space anywhere in the maze. Then move to the neighbor with the smallest number. Repeat that until you reach the destination.

This algorithm also gives an answer to the question, "Can I move to the space h, v at all?" If it isn't possible to move from one starting point to the destination, the starting point will have the value 999 (the initial value) after the algorithm has completed.

Game Entities

A game entity is an object in the game world, something that is not just part of the terrain or background. It is an object that might be moved, that can interact with other entities, and that may have other properties such as monetary value or hit points. An entity does not have to be a graphical object. Thus, it is not necessarily a sprite, but it can be displayed on the screen using a sprite. A sprite is a graphical concept, while a game entity is a game mechanics concept.

In Dungeon 1, the only thing in the game that is more than a single value in the `tileArray` is the *player object*, the hero, which has hit points. From that point of view, the hero is our only game entity. However, from the player's point of view, the trolls, and perhaps the treasures as well, are game entities too.

Dungeon 1 is really a kind of simple board game. Once the game is improved to a higher level of complexity, the simple representations used are no longer adequate. I will now suggest an alternative way to represent the game entities in Dungeon, building upon the Dungeon 2 variation of the game.

Once a game entity needs more characteristics than what the `tileArray` allows (`tileArray` tells us only that there is an object of a certain kind in the space—that entity's only "characteristic" is that it exists), the entity is best represented with a data structure.

Data Structures for Game Entities

Dungeon E is yet another variation of the Dungeon demo. This time I implement data structures for game entities. Each game entity is given its own copy of a game entity data structure, and the structures are organized in a linked list.

From the player's point of view, Dungeon E (E stands for Entities) works almost the same as Dungeon 2 (Dungeon 1 plus random level generation and menus). There are some subtle differences, however. First, the trolls can move over the exit position. A second addition, one that has less to do with the game entities, is the score line at the bottom.

I use the following data structure for representing a game entity:

```
/* All the possible kinds of game entities */
typedef enum {playerEntity, enemyEntity, goldEntity} EntityType;

/*Variables describing a game entity*/
typedef struct GameEntityRecord {
   EntityType     kind;       /* What kind of entity? */
   Point          position;   /* Where in the grid? */
   short          value;      /* Hit points or point value */
   struct GameEntityRecord  *previous, *next; /* Next and previous entity in the ➥
list */
} GameEntityRecord;

typedef GameEntityRecord *GameEntityPtr;            /* Define a handle type to it */
```

There is very little information about each entity, since Dungeon is such a simple game. Each entity has a position, a kind, and a value. The value is a multipurpose field used for the hit points of the trolls and for the monetary value of the treasures.

A real game would have much more information for each game entity. Fields can give much more detail to each individual entity. Entities can have individual names, they can have some personality variables, and they can have some memory of what has happened to them.

Imagine, for example, a multiplayer game in which one player attacks some monster. That monster can have a different attitude toward that player than toward another player. To have that different bent, the monster must keep some information about the event.

For the moment, however, let's stick to this simple representation. I will go into richer structures in the next chapter, which discusses game entity behaviors.

These simple data structures are arranged in a *linked list*. In `GameEntityRecord()` there are two fields named `next` and `previous`. They are pointers to the next and previous item in the linked list of game entities. A pointer to the very first item in this list is stored in the global `gEntityList`, which is what I would call the list root, the place from where I start each time I search through the list.

This is a double linked list, having pointers in both ways (see figure 7.12). I prefer this kind of list because it's much easier to modify than a list that has pointers only in one direction (i.e., a `next` but no `previous` field).

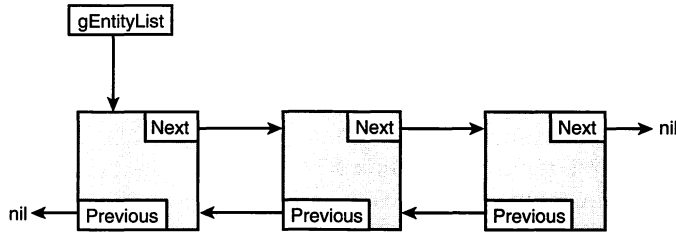


Figure 7.12 *The entity list is a double linked list—each item in the list has a reference to the previous item as well as to the next.*

Using a linked list for the game entities is extremely convenient. It can grow and shrink dynamically, limited only by the amount of memory available. Entities can be added anywhere and removed without leaving any kind of holes in the list.

I have written two routines for manipulating this list, `NewEntity()` and `DisposeEntity()`. `NewEntity()` allocates space for a new `GameEntityRecord`, adds it to the list, and returns a pointer to it, a `GameEntityPtr`. `DisposeEntity()` takes a `GameEntityPtr`, unlinks it from the list, and disposes of the memory.

```
/* NewEntity() allocates space for a new entity and puts it in the entity list */
static GameEntityPtr NewEntity()
{
  GameEntityPtr  newEntity;
```

First of all, I allocate the space for the new entity using `NewPtr()` and type-casting the pointer to a `GameEntityPtr`.

```
  newEntity = (GameEntityPtr) NewPtr(sizeof(GameEntityRecord));
```

If `NewPtr()` failed, return NIL.

```
  if (newEntity == nil) return nil;
```

The new item will now be linked into the list as its new first item. If the entity list is not empty (that is, if `gEntityList` is not NIL), then the first item in the current list gets the new item as its `previous` item.

```
  if (gEntityList != nil)
    gEntityList->previous = newEntity;
```

The new item's `next` item is set to the first item in the current list.

```
  newEntity->next = gEntityList;
```

The new item is the first, so there is no `previous` to it.

```
  newEntity->previous = nil;
```

Finally, make `gEntity` point to the new item, and return it.

```
  gEntityList = newEntity;
  return newEntity;
} /*NewEntity()*/
```

Removing an item from the list is equally simple, if not simpler. The routine `DisposeEntity()` takes a `GameEntityPtr` as parameter named `doomedEntity`, which is the entity that will be disposed of.

```
/* DisposeEntity() removes an entity from the list and disposes of it. */
static void DisposeEntity(GameEntityPtr doomedEntity)
{
```

If the program sends nothing to the routine, it does nothing. Makes sense, doesn't it?

```
  if (doomedEntity== nil) return;
```

The `next` and `previous` items are set to point to each other instead of to `doomedEntity`—if they exist at all.

```
  if (doomedEntity->next != nil)
    doomedEntity->next->previous = doomedEntity->previous;
  if (doomedEntity->previous != nil)
    doomedEntity->previous->next = doomedEntity->next;
```

A special case is when the first item is disposed of. Then, the `gEntityList` pointer must be changed to point to the next item.

```
  if (doomedEntity == gEntityList)
    gEntityList = doomedEntity->next;
```

Finally, doomedEntity is unlinked and can be disposed of.

```
    DisposePtr((Ptr)doomedEntity);
} /*DisposeEntity()*/
```

The routines above handle nothing but the linked list, linking and unlinking, and allocating and disposing of memory. The good news is that it's easy to reuse the routines for other programs. However, they are not very practical to call directly, since they don't take care of game-specific things like filling in the fields and making sure each entity is being drawn.

Thus, on top of these routines, I have a second layer consisting of the routines MakeEntity() and KillEntity(). MakeEntity() calls NewEntity(), fills in the fields, puts a reference to the new entity in entityArray, and draws it. KillEntity() removes an entity from entityArray, disposes of it, and redraws the position where it used to be.

```
/* MakeEntity() builds an entity with the given values and puts it into the ➥
entityArray */
static GameEntityPtr MakeEntity(EntityType eType, short hPos, short vPos, short ➥
value)
{
    GameEntityPtr newEntity;
```

First of all, I call NewEntity() to create the entity. If NIL is returned, return. Nothing more can be done in that case.

```
    newEntity = NewEntity();
    if (newEntity == nil) return nil;
```

Next, I fill in all the fields.

```
    newEntity->position.h = hPos;
    newEntity->position.v = vPos;
    newEntity->kind = eType;
    newEntity->value = value;
```

The pointer is written into entityArray.

```
/* Border checking here would be good for safety */
    entityArray[newEntity->position.h][newEntity->position.v] = newEntity;
```

DrawTile() is called to draw the new entity.

```
    DrawTile(newEntity->position.h, newEntity->position.v);
    return newEntity;
} /*MakeEntity()*/
```

`KillEntity()` is even more straightforward than `MakeEntity()`.

```
/* KillEntity() disposes of an entity, removes it from EntityArray, and erases it */
static void KillEntity(GameEntityPtr doomedEntity)
{
```

First, never do anything with a NIL pointer.

```
  if (doomedEntity == nil) return;
```

Set the corresponding `entityArray` entry to NIL—but only if it refers to `doomedEntity`. If it doesn't, something has gone wrong, but there's probably some other entity on the same spot.

```
  if (entityArray[doomedEntity->position.h][doomedEntity->position.v] ==➥
doomedEntity)
    entityArray[doomedEntity->position.h][doomedEntity->position.v] = nil;
```

Redraw the space and call `DisposeEntity()`.

```
  DrawTile(doomedEntity->position.h, doomedEntity->position.v);
  DisposeEntity(doomedEntity);
} /*KillEntity()*/
```

So, what is the point of this representation? Above everything else, it gives the possibility to expand the entities' personalities as much as we like. Each and every entity has its own private memory.

The organization in a linked list is also, in many ways, a nicer method for storing the information. One point at which Dungeon E really has gotten better is at the end of `MovePlayer()`, where `MoveEnemy()` is called.

```
/* Monsters are allowed to move now!*/

  theEntity = gEntityList;
  while (theEntity != nil)
  {
    if (theEntity->kind == enemyEntity)
      MoveEnemy(theEntity);
    theEntity = theEntity->next;
  }
```

What happens here is that the program steps through the list, item by item, and for each item that is an `enemyEntity`, `MoveEnemy()` is called. This continues until the list ends. The kludge with scanning the whole array twice is gone.

In order to have at least a little use for the information in each entity, I give each enemy two hit points instead of one (making the game considerably harder) and set a value of each treasure pile upon creating it.

Do you feel uncomfortable with a linked list? You can just as well use an array, but that means that you'll have to set a fixed maximum number of game entities, and there will be many idle entities in the array all the time. The array method works just as well as long as you don't underestimate how many entities can be needed in the game.

There is one drawback with linked lists compared to arrays. The frequent calls to `NewPtr()` and `DisposePtr()` can take some time. It is possible to speed that up a bit by not disposing of old memory blocks but rather collecting them in another list, from which you can take memory blocks as needed rather than calling `NewPtr()`.

Game Entities and Sprites

So far I've been talking about game entities, which is a rather broad concept. Now I'll talk more specifically about sprites rather than game entities. In order to not make the terms too confusing, I'd like to put a few things straight with some definitions. These definitions are a preparation for the next section, in which I describe the SpriteEngine demo, a foundation for some demos that will come later.

Game Entity

As previously described, a game entity is an object in the game world, something that is not just part of the terrain or background. It is an object that can be moved and that can interact with other entities. A game entity does not have to be a graphical object. A character in a text adventure is also a game entity.

Face

A face is the appearance of a sprite, or a picture/icon. A sprite can change its face at any time. The size and shape of a sprite are defined by its current face (possibly in conjunction with a separate rectangle for collision detection, but that is not supported by SpriteEngine).

Frame

A frame is one step of animation. If we generate ten steps of animation per second, the frame rate is 10 fps. In some texts, you may find the term "frame" for what I call face, but in my opinion, that can mess up the concept of a frame rate. We don't have to change the face to make a new frame, and changing the face of one sprite is not one new frame.

Sprite

A sprite is what you get if you take a game entity and present it graphically, animated on the screen. One could say that the sprite is the graphical part of a game entity. Such a game entity includes information the position of its sprite, its sprite's current face, the kind of sprite it is, plus game-dependent information that defines it as a game entity.

SpriteEngine

In chapter 3, "Advanced Graphics," you learned about sprite animation. In the following section, the demos use a set of routines named SpriteEngine which is, as the name says, a simple sprite engine (it's also, of course, on the CD-ROM—in the Libraries folder within the More Source folder). SpriteEngine is built explicitly as easy to use for the demos below. A sprite engine is relatively complex, so I am reusing the same engine for several demos.

Note that SpriteEngine is *not* primarily a graphics demo! It is far from the fastest you can find.

SpriteEngine consists of two header files, SpriteTools.h and SpriteHandlers.h, and three source files, SpriteEngine.c, SpriteHandlers.c, and SpriteTools.c.

The demo is written as a shell in which the reusable code is separated from the code that changes from demo to demo. SpriteEngine.c, SpriteTools.c, and SpriteTools.h hold all reusable code. Only SpriteHandlers.c and SpriteHandlers.h are changed.

The program has a strict hierarchical organization, with SpriteEngine.c being the main unit calling any of the others, but no routine in it is ever called from other units. SpriteHandlers.c is in between, using routines in SpriteTools.c as necessary.

SpriteEngine.c

In SpriteEngine.c you find the following routines:

`InitSpriteEngine()` initializes the two GWorlds `gOffScreen` and `gBackScreen`.

`RunSpriteEngine()` runs one frame of animation. It goes through the following sequence:

1. Erase sprites from `gOffScreen`.
2. Call `MoveSprite()` for each sprite.
3. Check for collisions using the bounding rectangles and call `HitSprite()` when a collision occurs.
4. Draw sprites in `gOffScreen`.
5. Copy changed areas from `gOffScreen` to the screen.

`MoveSprite()` and `HitSprite()` should be in SpriteHandlers.c.

`InitToolbox()` is the same old routine—again!

`InitStuff()` sets up a window and seeds the random number generator.

The `main` program calls all initialization procedures and calls `RunSpriteEngine()` until the user clicks the mouse. No, that's right, no event processing. That's up to you.

SpriteTools.c

SpriteTools.c holds utility routines for the other units and calls no other units itself. Many of the routines make little sense at this point.

`MyNewGWorld()` is a function that creates a GWorld and locks it in one single call.

`LoadFaceFromCicn()` creates a "face" the way SpriteEngine wants it: a GWorld for each. This wastes lots of memory, but it's easy to use. The routine creates a GWorld and draws the 'cicn' in it. This preloads the 'cicn' so it can be `CopyBits()`ed, which is much faster than `PlotCIcon()`.

`LoadFaceFromCicn()` also creates a mask region which is stored in the `clipRgn` of the GWorld. This assumes that the face is never drawn into after that change, so the `clipRgn` will not be used by QuickDraw.

`PlotFace()` draws a "face" by doing a `CopyBits()` from the "face" GWorld to some other port. This is done with the mask region mentioned above.

`NewSprite()` allocates space for a new sprite and links it into the sprite list.

`DisposeSprite()` takes out a sprite from the sprite list and disposes of the memory it occupied.

`KeepOnScreen()` checks if a sprite is located within the window borders and moves it if the sprite is outside.

`KeepOnScreenFixed()` is the same as `KeepOnScreen()`, but works when the sprites have a `fixedPointPosition` field.

`RectSeparate()` separates two sprites so that their bounding boxes no longer overlap.

`Rand()` is the same as in Dungeon: It generates a random value from 0 to `range-1`.

`RegionHit()` checks if the mask regions of two sprites overlap. It's used for advanced collision detection.

`SplitVector()` splits a vector into two components. It will be described in the collision handling section in the next chapter.

SpriteTools.h

In SpriteTools.h you find prototypes of all the routines in SpriteTools.c, plus routines for the mandatory functions in SpriteHandlers.c. You also find a number of globals and definitions.

SpriteHandlers.c

SpriteHandlers.c is different for every SpriteEngine-using demo. But it always holds the following routines:

`MoveSprite()` is called once for every sprite in the sprite list when it should be moved.

`HitSprite()` is called when two sprites have overlapping bounding rectangles.

In `InitSprites()`, faces should be loaded and sprites created.

SpriteHandlers.h

SpriteHandlers.h holds definitions that are changed from demo to demo, namely the `SpriteRecord`. I add new fields in it as necessary.

```
typedef struct SpriteRecord {
/*Game entity data -- edit as desired*/
  Point       speed;            /*  */
/*Sprite data -- don't remove*/
  Point       position;              /* Integer screen coordinates! */
  GrafPtr     face;                  /* Appearance of the sprite */
  Rect        drawingRect;           /* Where is it? */
/*List pointers -- don't remove*/
  struct SpriteRecord *prev, *next;  /* Next entity in the list */
} SpriteRecord;
```

Pretty similar to the entity record a while back, don't you think? It's no wonder—the sprites are the game entities in sprite-based games.

With SpriteEngine, I have the foundation needed to dig deeper into game entities. The next step is how to handle movement; coming up you'll find the first demos using SpriteEngine.

Position, Speed, and Acceleration

When you're making games with smooth movement, such as sprite-based games, a little physics will help make it realistic and pleasing. There will be a lot more physics in the next chapter when I discuss collision handling, but in this section I start with the basics.

The movement of an object is defined by its position, its velocity (speed), and its acceleration.

Position

The position is best measured in screen coordinates. In two-dimensional games, position is conveniently stored in a `Point` structure. It is split into a horizontal and a vertical component (e.g., `position.h` and `position.v`) as shown in figure 7.13.

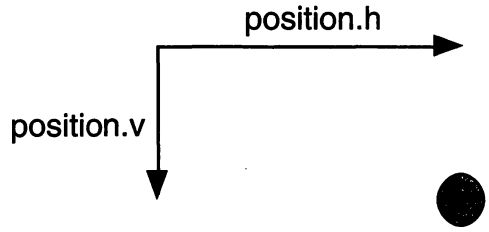


Figure 7.13 *A position is a two-component structure.*

Speed

The speed is usually measured in meters per second or miles per hour, but in a game it is best measured in the change in position per *frame*. The speed, just like the position, can be stored in the standard `Point` structure. In this case, the `Point` is no longer a "point," but a relative measure that can be applied to other points—I'll call it a *vector*, since it holds the data for speed and direction.

Since the speed denotes how much the position should change for every frame, the following operations should be made for every frame:

```
position.h = position.h + speed.h

position.v = position.v + speed.v
```

The result is that the object moves one step along the path defined by `speed` for every frame, as illustrated in figure 7.14.

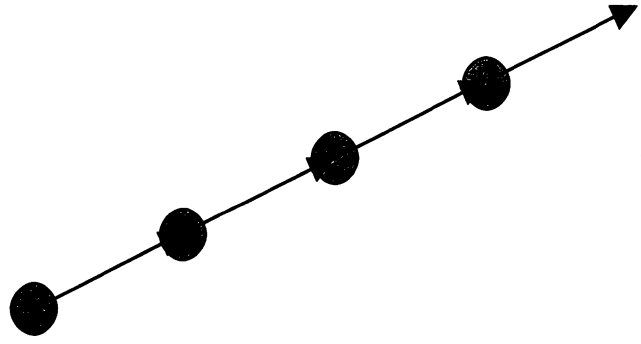


Figure 7.14 *Movement occurs when the speed is added to the position for every frame.*

Now it's time to get back to SpriteEngine and have a look at the first demo, SE Simple.

SE Simple

SE Simple (short for SpriteEngine Simple) is a very simple sprite demo. It creates three sprites and animates them with constant speeds. The only file you need to look in is SpriteHandlers.c, and within that, only `MoveSprite()` is of any real interest.

`MoveSprite()` defines the movement you see. It adds the speed to the position and calls `KeepOnScreen()` to make the sprites stay visible.

```
void MoveSprite(SpritePtr theSprite)
{
   theSprite->position.h += theSprite->speed.h;
   theSprite->position.v += theSprite->speed.v;
   KeepOnScreen(theSprite);
} /*MoveSprite()*/
```

The rest of the file is an empty stub for `HitSprite()` (collisions aren't handled here) and `InitSprites()`, which just loads the faces and creates sprites.

Acceleration

The acceleration is the next step. It is the change in speed per time unit, which in a game translates into how much the speed should change for every frame.

The acceleration can be stored as a `Point`, just like the speed. In such a case, the speed should be changed according to the acceleration:

```
speed.h = speed.h + acceleration.h

speed.v = speed.v + acceleration.v
```

However, the acceleration is usually constant. The most common acceleration goes straight down and is constant to simulate the case of gravity. In this case, only the vertical component of the speed is modified for every frame:

```
speed.v = speed.v + kGravitation
```

This will cause resulting movement like that shown in figure 7.15.

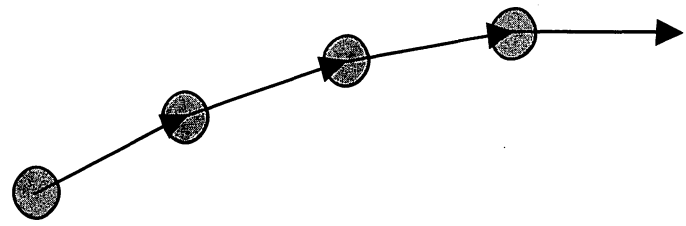


Figure 7.15 *Movement with gravity.*

There is a demo using gravitation, SE Gravity.

SE Gravity

SE Gravity is not unlike SE Simple. The difference is one simple line added to `MoveSprite()`!

```
void MoveSprite(SpritePtr theSprite)
{
   theSprite->speed.v++;
   theSprite->position.h += theSprite->speed.h;
   theSprite->position.v += theSprite->speed.v;
   KeepOnScreen(theSprite);
} /*MoveSprite()*/
```

Yes, I added an increment of the speed. It sure had some impact, don't you think?

Variable Acceleration

In some games, there can be acceleration that depends on the position. In the game Oxyd, for example, you have a top-view perspective of a marble. In that world, there are slopes, hills, and depressions, in which there exists an acceleration vector that depends on the position. The game is obviously grid-based, and I would expect each grid position to hold an acceleration vector.

For example, look at figure 7.16. It's supposed to be a large depression with a slope in each grid space, similar to what Oxyd has. Depending on what grid position the center of an object falls into, the object gets different acceleration.

Another variation appears in certain space games. In that environment, gravity is no longer a constant vector, but instead each planet or star attracts objects. In such games, one can build a new gravity vector for every position, adding contributions from all planets or stars.

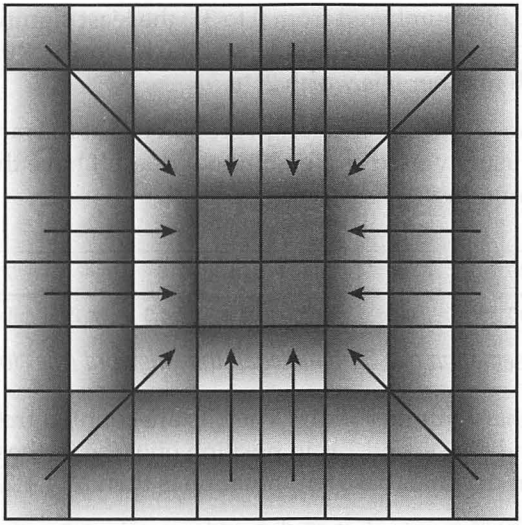


Figure 7.16 *In some games, the acceleration varies with the position.*

Fixed Point Arithmetics

So far I have been using integer screen coordinates, so that a change in one unit in the acceleration, speed, or position corresponds to a change of one pixel on the screen. This is the easiest scale to use, but it has a big drawback: It's insensitive when using small speeds. For some games this isn't a problem, but if you need some objects to move really slowly, integers can't do it. There is no middle ground between one pixel per frame and standstill.

One solution to this problem is to use floating point numbers. However, using floating point numbers is a common cause of speed loss on 68K Macs. Many 68K Macs, including several of the recent low-cost models, have no FPU. That means that floating point operations, even relatively few ones per frame, can result in a major speed loss. Some games demand an FPU to be present, but that means giving up a major part of the game market. Few LC or Performa computers have FPUs.

The third solution is to use integers, but use a finer scale than one pixel per unit. What you then get is fixed point arithmetics. With fixed point arithmetics, you use integer variables (typically longwords), but use a number of the least significant bits as decimals.

Note

A little note on the term "decimal." It's an oxymoron to use that word for binary digits, since "decimal" implies base ten, but I can't find another understandable word for the digits below the "decimal" point. By "decimals" I mean any digits with a weight smaller than one, no matter what the base is. By "decimal notation" I mean a number written in base ten.

A normal longword has 32 bits numbered from 0 to 31; the least significant bit has the weight 1 and each bit above that is worth double, so bit 0 is worth 2**0 = 1, bit 1 is worth 2**1 = 2, and so on, so a bit with number *n* is worth 2**n.

Figure 7.17 shows the value of each bit in a longword. Bit 0 is at the far right and is worth 1. The most significant bit is usually used as sign bit, so it doesn't have the weight 2**31, but rather -(2**31), a negative number.

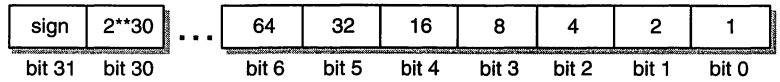


Figure 7.17 *The binary representation of a 32-bit integer, the longword (the "long" type).*

As an example, take the binary number 10010. The bits are counted from right to left, so the first 1 is in position four and the second 1 is in position one (the farthest right position is position zero). The value in decimal (base ten) notation is then 2**4 + 2**1 = 16 + 2 = 18.

However, the weights above are just what we have chosen them to mean. If you use 4 bits as decimals, each bit is worth 2**(*n*-4) instead. That is a fixed point representation with four binary decimals.

That gives us the representation of the longword shown in figure 7.18.

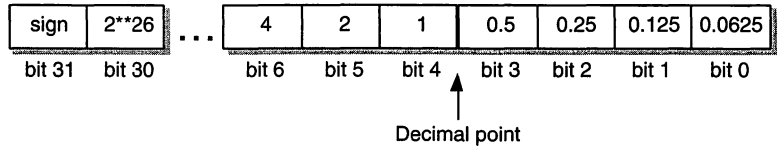


Figure 7.18 *The binary representation of a fixed point longword with 4 bits used for values smaller than 1.*

Apple provides a fixed point system in which 16 bits are used as "decimals," leaving 16 bits for values larger than 1. While that has its advantages (in particular making the conversion to integer very simple—just grab the high 2 bytes), I find 4 "decimal" bits to be quite enough for most gaming purposes, and it leaves much more for large numbers. With 4 bits, I can even use 16-bit integers (as in the `Point` data structure). If you need more precision, try 8 bits and use longwords only.

Now I'm sure you're wondering how to implement this. It's pretty easy. You need at least the following functions:

- Add and subtract fixed point numbers
- Multiply fixed point numbers
- Divide fixed point numbers
- Convert between fixed point numbers and integers

In the following section, I use four binary "decimals." You can easily adapt the snippets for other fixed point systems.

Add and Subtract Fixed Point Numbers

This is no problem. You can add and subtract fixed point numbers as if they were integers. You just have to watch out so that you don't add a fixed point number to an integer.

Multiply Fixed Point Numbers

You can multiply a fixed point by an integer. Just do it—the result is fixed point.

Multiplying fixed point numbers requires some care. If you multiply two fixed point numbers, the decimal point moves upwards, so if you use 4-bit fixed point, the result is 8-bit fixed point. You have to shift that down a few steps, four binary shifts.

```
temporary = fixedPoint1 * fixedPoint2;
fixedPointResult = temporary >> 4;
```

However, this will only work for positive products! If the product is negative, the shift will make four binary zeroes at the top, creating a large positive value.

That can be fixed by a binary OR operation. Use this if one but not both of the operands are negative. I check that by making a binary exclusive OR on the two operands and masking out the highest bit with a binary AND:

```
if ((fixedPoint1 ^ fixedPoint2) & 0x80000000 != 0)
   fixedPointResult |= 0xf0000000;
```

Divide Fixed Point Numbers

Dividing a fixed point number by an integer is straightforward. However, if you divide a fixed point number by another fixed point number, the result is an integer, not fixed point! If you want the result to be fixed point, you should shift the numerator up four steps.

```
fixedPointResult = (fixedPoint1 << 4) / fixedPoint2;
```

> **Note**
> *Note that the shifting above limits the values allowed. In this particular case, if `fixedPoint1` is a 16-bit integer (`int`), then it must not be larger than 512!*

Convert between Fixed Point Numbers and Integers

To convert an integer to fixed point, you shift it upward four steps:

```
fixedPoint = integer << 4;
```

To convert from fixed point to integer, you shift down, but you also have to check for negative values:

```
integer = fixedPoint >> 4;
if (fixedPoint1 < 0)
   integer |= 0xf0000000;
```

The program VelocityDemo, which you have seen in chapter 5, "User Interaction," uses fixed point in this way, although it is simplified a little bit by using only positive numbers.

Several of the SpriteEngine-using demos use fixed point numbers. The simplest ones are SE FixedPoint and SE Gravity FixedPoint.

SE FixedPoint

SE FixedPoint is similar to SE Simple, but the sprites move much slower. In particular, note how the sprites move only one pixel in 16 moves in some directions! That kind of slow movement is not possible with integer-based positions.

In `MoveSprite()`, the speed is added to `fixedPosition`. Then, the position is made by shifting the `fixedPointPosition` down 4 bits, that is, dividing by 16.

```
void MoveSprite(SpritePtr theSprite)
{
   Point   where;

   theSprite->fixedPointPosition.h += theSprite->speed.h;
   theSprite->fixedPointPosition.v += theSprite->speed.v;
   theSprite->position.h = theSprite->fixedPointPosition.h >> 4;
   theSprite->position.v = theSprite->fixedPointPosition.v >> 4;
   KeepOnScreenFixed(theSprite);
} /*MoveSprite()*/
```

SE Gravity FixedPoint

SE Gravity FixedPoint makes the same move as from SE Simple to SE Gravity: The speed is incremented. However, with fixed point arithmetics, much smoother and slower movement is possible.

```
void MoveSprite(SpritePtr theSprite)
{
   theSprite->speed.v++; // Simple gravity
   theSprite->fixedPointPosition.h += theSprite->speed.h;
   theSprite->fixedPointPosition.v += theSprite->speed.v;
   theSprite->position.h = theSprite->fixedPointPosition.h >> 4;
   theSprite->position.v = theSprite->fixedPointPosition.v >> 4;
   KeepOnScreenFixed(theSprite);
} /*MoveSprite()*/
```

Perhaps you prefer fast action before slow gliding? Fixed point doesn't in any way *demand* that all sprites are slow, it just enables the slow ones to move really nicely. Using fixed point also allows for fine-tuning speeds of objects in games with high frame rates.

I'll talk more about Game Mechanics in the next chapter. But first...

Profile: Will Wright

Will Wright played a huge role behind the phenomenon known as SimCity. He did all of the design on SimCity Classic, and wrote the simulation code for all versions. Wright also programmed the Commodore 64 version and about half of the Mac version interface. SimCity Classic was released in 1989, SimCity2000 in 1993. Along with these efforts, he also worked on other games at Maxis, Inc., including SimEarth and SimAnt.

Q: When the original SimCity first came out, some people said, "This isn't much of a game. It's more of an 'activity.'" Now, SimCity is probably the best-selling computer game of all time. What is the key to SimCity's success?

A: I was very surprised at how successful SimCity became. I think part of the attraction is that it's more of a hobby than a game. It's sort of like having a model train set that you can go home and work on each night, twiddling parameters, adding things here and there, never quite finishing. The computer adds the ability to make this model seem "alive."

Q: Will people always be fascinated by "god games" or is this a passing trend?

A: I tend to think of these games not so much as "god games" but rather as "creativity games." These kinds of games have a much larger possibility space (or state space) than most other game genres. I think part of the fascination lies in knowing that no one else will create exactly the city (or whatever) that you create. Also, most of the problems will tend to have many solutions, so you can be creative on that end too.

Q: What is the city simulator code like? How does it work?

A: The simulator is sort of like a large three-dimensional cellular autonoma with tendrils connecting it to a more traditional system dynamics model. The cellular layers each run different rule sets (crime, traffic, etc.) but are highly connected to each other; they are primarily used for spatial data. The system dynamics portion of the model has inputs and outputs connected to the CAs. Some of these connections are sources and sinks for the data in the CAs, while other connections are used to moderate or "valve" the data flowing through the CAs.

continued

Q: When evaluating the state of the city, does the simulator evaluate the state in one big gulp and then pass program control to the rest of your code? Or do you fragment the evaluation up into smaller tasks?

A: The simulator sequences itself over the course of a game month. It is called once per game day and slices up its tasks into fairly even chunks. SimCity Classic was actually programmed with preemptive multitasking because even some of the small chunks were bogging down the interface. On the Mac, this was a big no-no and a real pain. But it was fun to solve the problem. I had to go under the level-7 interrupt vector, and while it was in the interrupt, I moved the stack pointer so that when the RTI was hit, it returned to a different place with a different stack.

Q: What takes up the most time in the simulator? Why?

A: Traffic and power scans. The traffic in SimCity is simulated using a technique known as "trip generation." Each building generates a number of trips based on its population. These trips have to crawl down the roads looking for a good destination. If a destination is found, the traffic along that route is incremented. To get good results with this technique, you need to go to a rather high trip frequency; with many buildings this adds up.

The power scan is similar, but unlike the traffic it is deterministic. It's kind of a specialized flood-fill which must be done all at once, whereas the traffic can be spaced out over several simulator ticks. Also, the number of cells involved in a power scan tends to be much higher than in traffic.

Q: From where did you obtain the ideas and knowledge that went into the simulator?

A: I just started reading everything I could find and at the same time started iteratively building my first models. I mostly read city planning books to get a sense of the noncomputer-based models (economic models, etc.) that planners thought in terms of. I started with the work of Jay Forrester (who sort of started the system dynamics stuff) and worked my way up to the present. I started this process in about 1985, and it continues to this day (I love a good excuse to read).

Q: Is the simulator a nice independent module, or did you have to sprinkle it throughout the code?

A: It's not a library that we call, but it is walled off from the rest of the code. There is just one routine called `Simulate()` that is called in our null event loop. The editor (which is what we call the interface) communicates with the simulator through global variables (budget, game speed) and through certain arrays in the CAs.

Q: What portion of the original SimCity takes up the most processor time?

A: For SimCity Classic, the power scan was the big drain. For a large city, it might take up to 15 seconds to run a full power scan. This was the reason I had to design this simulator as a preemptable task. As the machines got faster, this got to be less and less of an issue and we were able to redesign it as a time-sliced architecture.

Q: When optimizing the game, did you approach the simulator in any different way than the rest of the code?

A: The main difference in optimizing the simulator had to do with portability. After spending so much time tuning the simulation, we wanted to make sure that it didn't change from machine to machine. In almost all of the ports, we've been able to move the simulator code over with very minor changes. The entire simulator is written in ANSI C with the exception of the random number routines. The random routines are called so often and are so simple to rewrite that we tried (up until the PowerPC that is) to keep them in assembly.

Q: What was the most technically challenging part of writing SimCity?

A: On the Mac, without a doubt it was the multitasking thing. After designing and coding the thing up for two weeks, I turned it on and, of course, the machine crashed. I sat and thought about it for days. Then it hit me. I remembered the stacker-sniffer in the VBL task. This is a small routine that continually checks to see if the stack has collided with the heap. Since my alternate stack was buried down in the heap, it was just a matter of moving it high. In five minutes, I had it working like a charm (until the Mac II came out, but that's a different story).

Q: Do you have any regrets about the way that the final version of SimCity turned out?

A: Not really. For the level of hardware at the time, I don't think we could have done much more with it. I had so many things that I wanted to add at the time, but I managed to get most of these things into SimCity2000.

Q: Let's turn to SimCity2000. What are you most proud of in SimCity2000?

A: In SimCity2000 I think we did a nice job of balancing all the elements of the design. There were so many things we could have expanded, but for diminishing returns. Part of the balance we tried to achieve here was to give SimCity Classic players an easy route into the game. By keeping the tool bar layout substantially similar and hiding most of the added complexity in the pop-up menus, I think we did a reasonable job of it.

Q: Did any basic design philosophies change between SimCity and SimCity2000?

A: I think SimCity2000 was more influenced by faster hardware and more memory than any change in design philosophy. User comments that we received after

continued

SimCity Classic also became a large contribution to the design. I had a box in my office that contained all the letters full of suggestions and I read all 2,000 of them as well as a billion online messages. The three most common suggestions were schools, hills, and a water system.

Q: What part of SimCity2000 takes up the most processor time?

A: Screen animation and updates are the big time-eater. When we went to the isometric viewpoint, many display issues became much more critical, and I had to spend a lot of time dealing with updating techniques. Mick Foley (who wrote chapter 3 in this book) wrote a set of high-speed graphic assembly routines which I called to build the isometric view. We decided early on not to go with a Z-Buffer scheme, primarily for memory considerations. Moving objects across the landscape (planes, trains, boats) became the thorniest problem and each was maintaining its own cliprect to draw into. Most of the other animation (traffic, water, smoke) was done with color-cycle animation and took very little time.

Q: Could the simulator be enhanced if you had written the game for, say, a PowerPC 620 chip and were assured of massive processing power?

A: The game is limited more by its complexity than by processor speed. Given a 620 to design on, I would probably focus more on added graphics and sound than on simulation. A more complex simulation does not necessarily mean a better game, but perhaps more detail could help increase the suspension of disbelief that the player experiences. On the hardware side, it is just now getting to the point (with the PowerPC) where my primary limitation is becoming memory, not processor.

Q: Are you satisfied with the simulators? In ideal circumstances (i.e., if the product never actually had to ship), would you have liked to tinker around with little aspects of the simulators for months, or do you think they form a really good game that wouldn't really be improved upon?

A: Actually, I do spend months tinkering with various simulator pieces. I'm moving more and more to a kind of experimental design approach. By building small simulation pieces in isolation, I and others at Maxis can experiment with their behaviors and get familiar with them. When they work nicely, we put them aside for later use as "building blocks" for the products.

Q: What's the coolest thing technically about SimCity2000?

A: For me the coolest technical thing about SimCity2000 is that it runs. I mean we're dealing with so many subtle simulation elements interacting at so many different time scales, and at the same time there are all these millions of possible ways that the player can alter the underlying data that the simulator is crunching, and on top of that all these pixels are being blasted to the screen on a nonstop basis. It's a wonder that the thing runs at all.

Q: Give us three cool tricks that fool users into believing that the world they're looking at is a real, complete world.

A: 1. Time flow should be visual. This can be accomplished at many scales. In SimCity you see time flowing from seconds (stoplights, smoke, water) to minutes (planes, trains, fires) to hours (traffic) to days (newspapers) to months (new construction) to years (city evolution). This gives the simulated world an almost fractal-time scruffiness which meshes well with reality.

2. The world should be full of unexpected surprises. It's best if these surprises arise as a combination of the dynamics in the game rather than explicit programming by the designer. The very best ones are those that surprise the designer, the "Cool, I didn't expect *that* to happen" type of things.

3. Try to give the world a very large and apparent "phase space." This is a high-dimensional space that represents all the possible states the world can exist in (this is roughly the saved game size). Each game played will become a different exploration of (a vector through) that space. Not only should this space be large, but players should notice real differences as they move around here. Many games will have a win condition but don't make that condition one point in the space, or everyone will be attracted to the same path.

Q: Do you ever play either game?

A: I still play SimCity2000 occasionally. I haven't played SimCity Classic in years.

Q: What advice would you give a beginner to Mac programming who wants to learn about programming games on the Mac?

A: I would say first start with a good compiler (Metrowerks or Symantec) and a few books (maybe one on C/C++, one on the Mac Toolbox, and one like this one). Second, work your way through the books roughly in that order (C, Toolbox, games). Third, when you feel ready for it, program a game. Start with some simple "shareware" idea and try your best to complete it. Completing it is the hard part; many programmers can produce 90 percent of a product but that last 10 percent is where most of the "real" work happens. At that point on your first project, you'll realize why you did the first 90 percent all wrong and why you should just quit right now. Resist that urge—it leads to the dark side (or at least it doesn't lead to a career in game programming). Just get the thing finished to the point where you won't be embarrassed to upload it for other people to try and complain about. If you manage to get that far, the rest is a piece of cake.

Q: What kind of games do you like to play when you're not at work?

A: I try a lot of different games. I really enjoy most the stuff that LucasArts makes (TIE fighter, Sam & Max) as well as the occasional strategy game (Panzer General).

Chapter 8
Game Mechanics: Behavior

The behavior of a game entity is sometimes called "artificial intelligence," or AI. But artificial intelligence is too strong a term for what appears in most games. Even though game entities sometimes exhibit behavior that makes them seem marginally intelligent, a more modest term is more appropriate. I call it game entity behavior.

Game Entity Behavior

If a game entity is to appear intelligent, it needs to have a *plan* and a *memory*. The more sophisticated the plan and memory usage, the smarter the game entity appears. Most entities you see in current games have very little plan and sometimes no memory at all. They don't appear too bright, but are still smart enough for the game to be entertaining.

Compare, for example, a computer opponent in a chess game to the aliens in Space Invaders. The chess-playing game entity makes use of many real AI techniques. The aliens in Space Invaders, however, merely follow a predefined path, although they still pose a challenge to the player thanks to their numbers and good game design.

A chess-playing entity uses sophisticated algorithms to search among the available moves to identify the optimal plan given the current situation. It remembers its conclusions over time, so it can keep building on its plan. Obviously this entity has both plan and memory. The Space Invaders aliens also have a plan and memory. The plan is to follow a path and occasionally fire a shot at the player. The memory is almost nonexistent—all they remember is whether they move right or left, and *possibly* when they fired a shot last time.

The difference isn't just that the chess-playing entity is bigger and more advanced. What's essential to AI is the *adaptability* of the entity. The entity adjusts its behavior to that of the player. If the player makes a move that the entity hasn't planned on, it can replan. That makes it reactive to the player's activities. The aliens in Space Invaders couldn't care less about the player.

So, how do we create seemingly intelligent entities with adaptive plans? There are countless ways to design game entity behavior, but I will concentrate on cases in two different categories: action games, in which the entities are usually sprites; and board games, card games, and strategy games, in which the entities are the player's opponents, playing the game under the same rules as the player does.

Action Game Entities

In this section, I talk about sprite behavior rather than game entity behavior, since the demos on the CD-ROM use sprites anyway.

In action games, the game entities are usually very simple. You can't really call them smart. It is possible to make sprites appear very smart, but most action games demand rather little intelligence from their game entities.

In action games, the following behaviors are common:

- Hunting
- Evading
- Moving randomly
- Following a program
- Sleeping

After discussing these behaviors individually, I discuss how to combine several behaviors using state models.

Hunting

Hunting another entity does not necessarily mean the hunter is hostile. An entity can follow another because it is allied with it, in love with it and wants to be close to it, or because it hopes to get protection against some common enemy.

The simplest hunting algorithm is the following (for two sprites, one named `hunter`, which hunts a sprite named `target`):

```
if (hunter->position.h < target->position.h)
  hunter->position.h++;
else if (hunter->position.h > target->position.h)
     hunter->position.h--;
if (hunter->position.v < target->position.v)
  hunter->position.v++;
else if (hunter->position.v > target->position.v)
  hunter->position.v--;
```

The problem with this algorithm is that the movements are unrealistic. The hunter moves diagonally until it is on the same horizontal or vertical line as the target, and then it moves horizontally or vertically, respectively.

If the hunter can move straight toward the target, the movement looks much better. If the hunter moves faster, it can use speed vectors that aren't exactly horizontal, vertical, or diagonal.

Here's a quick way to get results along these lines: The speed of the hunter is set to a fraction of the distance to the target. This means that it moves fast when far away and slow when close. That may or may not be what you want. Tune it for your own needs.

```
vector.h = target->position.h - hunter->position.h;
vector.v = target->position.v - hunter->position.v;
hunter->speed.h = vector.h / kScale;
hunter->speed.v = vector.v / kScale;
```

If integers are used for this, the hunter often moves very fast, or there are noticeable quantization effects when the speed is scaled down close to 0 or 1. In those low speeds, it falls back into the eight directions it had in the simple hunting algorithm above. If fixed point arithmetics (which were thoroughly described in chapter 7) are used for both speeds and positions, there are no noticeable quantization effects. Movement occurs in very small steps, in fractions of a pixel per frame, which also allows movement in any direction, even in low speed.

If you want the hunter to move in constant speed toward the target, some calculations are needed. What you need to do is to *normalize* the vector toward the player.

If you have a vector `v1` and want a vector `v2` in the same direction but of a specified length, you calculate it as:

$$v2 = v1 * \frac{length}{|v1|}$$

where $|v1|$ is the length of `v1`. Since the length of a vector is

$$|v1| = \sqrt{v1.h^2 + v1.v^2}$$

this can be implemented as:

```
v2.h = v1.h * length / sqrt(v1.h * v1.h + v1.v * v1.v)
v2.v = v1.v * length / sqrt(v1.h * v1.h + v1.v * v1.v)
```

Unfortunately, this normalization operation requires square root operations, which tend to make calculations too slow. In some cases, you can fix this by making look-up tables for the square root operation. I'll return to that in appendix A, "Top Speed—or Not?"

Another option is to use Pseudo-Euclidean distance, which I mentioned in the previous chapter, in the "Pseudo-Euclidean and Euclidean Metrics" section. For the particular purpose of calculating a distance, you can use the following method: The distance is the length of the longest vector component plus a constant `k` times the length of the shorter component. The constant `k` can, for example, be 1/3. This can be implemented as follows:

```
#define max(a,b)   (a>b?a:b)
#define min(a,b)   (a<b?a:b)
#define abs(a)     (a>0?a:-a)

int PEDistance(int h, int v)
{
   return max(abs(h),abs(v)) + min(abs(h),abs(v)) / 3;
}
```

This is just another way to express the Pseudo-Euclidean distance measure from chapter 7, where it appeared as different movement costs in different directions. Just as we can use different movement costs there, we can consider different k values here. 1/3 is very close to optimal. Using 3/8, you can avoid the division, replacing it with shifts.

Evading

Evading is the exact opposite of hunting. Instead of moving toward the player (or some other game entity), the evader moves away. To make evasion algorithms of the hunting algorithms above, just move in the opposite direction.

Hunting and evading are demonstrated in two demos, SE Hunter & Evader and SE Hunter & Evader 2. You will find them among the other SpriteEngine demos on the CD-ROM (in the Libraries folder).

SE Hunter & Evader

In the Hunter & Evader demos, there are three kinds of sprites: `playerSprite`, `hunterSprite`, and `evaderSprite`. A field with this information has been added in the `SpriteRecord`. One of each kind of sprite is used. A green icon is used for `hunterSprite`, a red one for `evaderSprite`, and a gray one for `playerSprite`.

The `playerSprite` follows the mouse cursor all the time. The `hunterSprite` uses the simple algorithm above to hunt the `playerSprite`, while the `evaderSprite` uses the same algorithm to move away. The `evaderSprite` also has an extra bound check to keep it not just within the window, but 50 pixels inside it.

```
void MoveSprite(SpritePtr theSprite)
{
   switch (theSprite->kind)
   {
```

The `playerSprite` sets its position to the mouse position using `GetMouse()`. It also copies its position to `gPlayerPosition` in order to make it available to the other sprites.

```
      case playerSprite:
         GetMouse(&theSprite->position);
         gPlayerPosition = theSprite->position;
         break;
```

```
    case hunterSprite:
      if (theSprite->position.h < gPlayerPosition.h)
        theSprite->position.h++;
      else
      if (theSprite->position.h > gPlayerPosition.h)
        theSprite->position.h--;
      if (theSprite->position.v < gPlayerPosition.v)
        theSprite->position.v++;
      else
      if (theSprite->position.v > gPlayerPosition.v)
        theSprite->position.v--;
      break;
```

The `evaderSprite` starts with the reverse of the algorithm you saw above...

```
    case evaderSprite:
      if (theSprite->position.h < gPlayerPosition.h)
        theSprite->position.h--;
      else
      if (theSprite->position.h > gPlayerPosition.h)
        theSprite->position.h++;
      if (theSprite->position.v < gPlayerPosition.v)
        theSprite->position.v--;
      else
      if (theSprite->position.v > gPlayerPosition.v)
        theSprite->position.v++;
```

...but after that, it also makes an extra border check. Otherwise, the `playerSprite` wouldn't be able to get behind the `evaderSprite` and chase it away from the walls and corners.

```
/* Keep this sprite a bit inside the bounds so it doesn't just stand in a corner. */
      if (theSprite->position.h < 50)
        theSprite->position.h = 50;
      if (theSprite->position.h > gOffScreen->portRect.right - theSprite->face-> ➥
portRect.right - 50)
        theSprite->position.h = gOffScreen->portRect.right - theSprite->face-> ➥
portRect.right - 50;
      if (theSprite->position.v < 50)
        theSprite->position.v = 50;
      if (theSprite->position.v > gOffScreen->portRect.bottom - theSprite->face-> ➥
portRect.bottom - 50)
        theSprite->position.v = gOffScreen->portRect.bottom - theSprite->face-> ➥
portRect.bottom - 50;
      break;
  }
```

`KeepOnScreen()` is the standard border check from SpriteTools.c, which is used for all sprites.

```
  KeepOnScreen(theSprite);

} /*MoveSprite()*/
```

Note how the sprites move! They always move horizontally, diagonally, or vertically, never in any directions in between. This is not particularly realistic. The other demo, SE Hunter & Evader 2, uses the second algorithm that I suggested to take the difference between the positions and use a fraction of that as speed.

SE Hunter & Evader 2

In SE Hunter & Evader 2, the hunter has the following algorithm. First, it moves using fixed point, which works the same way you saw previously in SE FixedPoint.

```
case hunterSprite:
   theSprite->fixedPointPosition.h += theSprite->speed.h;
   theSprite->fixedPointPosition.v += theSprite->speed.v;
   theSprite->position.h = theSprite->fixedPointPosition.h >> 4;
   theSprite->position.v = theSprite->fixedPointPosition.v >> 4;
   KeepOnScreenFixed(theSprite);
```

Then, it calculates a vector from the hunter to the player sprite.

```
vector.h = gPlayerPosition.h - theSprite->position.h;
vector.v = gPlayerPosition.v - theSprite->position.v;
```

The speed is then set to 1/8 of the vector. Since 4-bit fixed point numbers are used for the speed, this is really 1/128 of the vector.

```
theSprite->speed.h = vector.h / 8;
theSprite->speed.v = vector.v / 8;
```

The evader works the same way, but sets the speed in the other direction.

```
theSprite->speed.h = -vector.h / 8;
theSprite->speed.v = -vector.v / 8;
```

There are also some changes in the extra bound check for the evader, but those are modifications just for the fixed point arithmetics.

Moving Randomly

A very simple sprite behavior is random movement. Random movement does not make a sprite appear smart, but at least it makes it not totally predictable.

There are several ways to make a sprite move randomly.

- Add random values to the position.
- Add random values to the speed.
- Change the speed occasionally, at random or fixed intervals.

The simplest random movement is to add a random offset to the position for every frame. That makes the sprite shake. It's not particularly pleasing.

Another way is to change the speed by a random addition for every frame. This makes the sprite glide around aimlessly, slowly changing directions and speed. It looks a lot better than the first method.

If you only change the speed once in a while, the sprite will move in a little more determined way. This may be better or worse depending on your game.

These three random movements, as well as a variation of the last one, are demonstrated in the demo SE Random.

SE Random

SE Random includes four kinds of sprites and uses one of each kind. They are `randomPositionSprite`, `randomSpeedSprite`, `changeSometimesSprite`, and `randomImpulseSprite`.

The `randomPositionSprite` sprite (green) adds random values to its position.

The `randomSpeedSprite` sprite (pink) adds random values to its speed. It also lowers the speed a little when it hits the border, or else it would reach very high speeds over time.

The `changeSometimesSprite` sprite (yellow) changes its speed to a random value once in a while. For every frame, it decides randomly if it should change the speed or not.

The `randomImpulseSprite` sprite (blue) is a variation on the occasional change method. If the sprite is still, it gives itself a random, somewhat high speed. That speed is slowly lowered until the sprite is still again, at which point it gives itself a new speed. This looks as if the sprite gets an impulse in a random direction every time it stops.

The source code of the `MoveSprite()` procedure is as follows.

```
void MoveSprite(SpritePtr theSprite)
{

  switch (theSprite->kind)
  {
```

Random additions to the position are simple enough. Note that `Rand(7)-3` results in a random value from -3 to 3.

```
    case randomPositionSprite:
      theSprite->position.h += Rand(7) - 3;
      theSprite->position.v += Rand(7) - 3;
      KeepOnScreen(theSprite);
      break;
```

Random additions to the speed are a bit more complicated, partially because I use fixed point arithmetics...

```
case randomSpeedSprite:
  theSprite->fixedPointPosition.h += theSprite->speed.h;
  theSprite->fixedPointPosition.v += theSprite->speed.v;
  theSprite->position.h = theSprite->fixedPointPosition.h >> 4;
  theSprite->position.v = theSprite->fixedPointPosition.v >> 4;
```

...but also because I lower the speed slightly when the sprite hits the border.

```
  if (KeepOnScreenFixed(theSprite))
  {
    theSprite->speed.h -= Sgn(theSprite->speed.h);
    theSprite->speed.v -= Sgn(theSprite->speed.v);
  };
```

The actual addition is achieved just like the addition to the position above.

```
  theSprite->speed.h += Rand(15) - 7;
  theSprite->speed.v += Rand(15) - 7;
  break;
```

For changing the speed occasionally, I use fixed point again, so the code begins like the previous example:

```
case changeSometimesSprite:
  theSprite->fixedPointPosition.h += theSprite->speed.h;
  theSprite->fixedPointPosition.v += theSprite->speed.v;
  theSprite->position.h = theSprite->fixedPointPosition.h >> 4;
  theSprite->position.v = theSprite->fixedPointPosition.v >> 4;
  KeepOnScreenFixed(theSprite);
```

I change the speed once every 50 frames, on the average.

```
  if (Rand(500) < 10)
  {
    theSprite->speed.h = Rand(63) - 31;
    theSprite->speed.v = Rand(63) - 31;
  }
  break;
```

For the random impulse sprite, I use fixed point again.

```
case randomImpulseSprite:
  theSprite->fixedPointPosition.h += theSprite->speed.h;
  theSprite->fixedPointPosition.v += theSprite->speed.v;
  theSprite->position.h = theSprite->fixedPointPosition.h >> 4;
  theSprite->position.v = theSprite->fixedPointPosition.v >> 4;
  KeepOnScreenFixed(theSprite);
```

Here's the interesting part. In order to simulate friction, the speed is slowly lowered by setting it to 19/20th of the previous value for every frame. When the speed is 0 in both components, I set a new speed.

```
        theSprite->speed.h = theSprite->speed.h * 19 / 20;
        theSprite->speed.v = theSprite->speed.v * 19 / 20;
        if (theSprite->speed.h == 0 && theSprite->speed.v == 0)
        {
          theSprite->speed.h = Rand(511) - 255;
          theSprite->speed.v = Rand(511) - 255;
        }
        break;
    }
} /*MoveSprite()*/
```

There are many other ways to make a sprite move randomly. In a grid, you can set a sprite to a random direction bound to the grid, i.e., up, down, left, or right. You can keep a sprite at a constant speed, changing only its direction. You can create swarms by making one randomly moving sprite accompanied by a number of hunters hunting that sprite. The list goes on. Adapt what I have shown here to what you need.

Sleeping

Sleeping is also a behavior, though not a very interesting one. A sleeping sprite sits still, doing nothing. As a behavior, this is interesting for sprites that can change behavior, a topic I return to later.

Following a Program

In some arcade games, most notably Phoenix and similar "improved Space Invaders" games, sprites follow "programs," or predefined paths. They may make dives, go sideways for a while, and then zig-zag back up again.

There are several ways to implement such behavior. One way is to do it all in code, with a lot of `if` statements or cases based on some timing variable. Simple programmed behaviors, such as the aliens in Space Invaders, are best made this way. All these aliens have to do is to move sideways until one of them hits the border; when that happens, they move downward a little bit and continue in the other direction.

Another way to implement programmed behaviors, a method which is fairly easy to fine-tune compared to the code-only method, is to make a list of movement instructions for the sprites to follow. The demo SE Programmed Sprite demonstrates one way to do this. Here, I call the list a "program," though others may call it a "script" or "pattern."

SE Programmed Sprite

In SE Programmed Sprite, three sprites move around following a predefined path. The path is defined as a sequence of instructions. An instruction is defined as a `ProgramPart`:

```
typedef struct ProgramPart
{
   short   howLong;
   short   dh;
   short   dv;
};
```

The `howLong` field defines how long (the number of frames) the instruction should be carried out. The `dh` and `dv` fields are offsets for the position. You could consider adding other fields, such as how often the sprite should fire during the instruction.

The whole list of instructions is an array of `ProgramParts`. In the demo, I initialize it directly in the code. In a real game, I would probably put it in a resource.

In SE Programmed Sprite, there's only one instruction list, named `program1`. It holds six instructions, and after that a line in which `howLong` is -1; this means that the end of the program is reached, so it should start over from the beginning.

```
struct ProgramPart  program1[] =
{
   {60, 0,1},
   {50, 1, 0},
   {30, 0, -1},
   {20, -1, 0},
   {30, 0, -1},
   {15, -2, 0},
   {-1, -1, -1}
};
```

The sprites need two new fields defined in the `SpriteRecord` structure in SpriteHandlers.h. They are `timeOnCurrent`, which tells for how many frames the current instruction has been in use, and `currentInstruction`, which holds the index of the current instruction.

Given the `program1` instruction list, `MoveSprite()` turns out to be rather simple. First, the `dh` and `dv` fields from the current instruction are added to the position:

```
void MoveSprite(SpritePtr theSprite)
{
   theSprite->position.h += program1[theSprite->currentInstruction].dh;
   theSprite->position.v += program1[theSprite->currentInstruction].dv;
```

The `timeOnCurrent` field is incremented each time the `MoveSprite()` routine is called. If `timeOnCurrent` has reached the `howLong` value, the `currentInstruction` is incremented. If that takes us to the last instruction (in which the `howLong` field is less than 0), `currentInstruction` is set to 0, which takes the sprite back to the first instruction.

```
   theSprite->timeOnCurrent++;
   if (theSprite->timeOnCurrent >= program1[theSprite->currentInstruction].howLong)
   {
      theSprite->timeOnCurrent = 0;
      theSprite->currentInstruction++;
      if (program1[theSprite->currentInstruction].howLong < 0)
         theSprite->currentInstruction = 0;
   }
   KeepOnScreen(theSprite);
} /*MoveSprite()*/
```

If a sprite just follows the same program over and over again, looping a fixed set of instructions, it won't be very interesting. However, if it can change programs and even switch to other behaviors, the result can be pretty interesting. That's what the next section covers.

Combining Several Behaviors

A sprite using only one of the behaviors above will be single-minded and appear stupid, even if the behavior is one of the more advanced ones. However, if a sprite changes its behavior, especially when that change is triggered by some event, the sprite can appear much smarter.

What it takes is enabling the sprite to make decisions based on what happens to it and what it remembers, so the sprite can choose the appropriate behavior based on that. The memory can be represented as *states*, and the sprite changes state after certain rules.

Take Marathon as an example. How smart are the aliens? They sit still, walk around randomly, or attack the player. That's about it. What they do is to switch between three of the behaviors I have already described: hunting, moving randomly, and sleeping. This can be described as a system of states and be drawn in a graph (see figure 8.1).

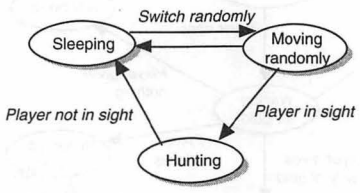


Figure 8.1 *A state-based model of the aliens in Marathon.*

Each alien should have one field in its game entity record (`SpriteRecord`) telling which state it's currently in. When an alien is processed, the state is used as a case switch and different subprograms are executed depending on the state.

If the alien is in the sleeping state, it will remain so until it randomly selects to start moving randomly. It will also stop sleeping if it is attacked, but that is not reflected in figure 8.1.

If the alien is in the random movement state, it processes random movement and checks either if it's time to sleep or if the player is within reach. In either of these cases, it switches state to one or the other.

Finally, if the alien is hunting, it will do so, attacking when near the player, until the player is no longer within sight.

To make this model complete, I would have to add a few more states and branches. In particular, the death of the player or the alien itself should be reflected. There should be one state for a dying alien, in which it moves away from the attacker, as well as a state for a dead alien, which is a kind of "sleeping" state.

This alien has very little memory. A state model can also be used for more long-term behavior. Let's take a somewhat different example: a "smart" monster in a dungeon-digging game.

I can imagine a monster with the following behavior: Before it meets the player, the monster is neutral. When it sees the player, it approaches and demands gold. If attacked, the monster attacks back and gets angry—from then on it remembers that it hates the player. If the monster is not given gold, it attacks, but it will not hate the player—if the attack is successful, it will take the player's gold and go away. If the monster is given a little gold, it is content and stays neutral, leaving the player alone. Finally, if the monster gets plenty of gold, it allies itself with the player and follows the player around. As long as it gets some share of the player's findings, it remains an ally. If not, it eventually becomes neutral again (see figure 8.2).

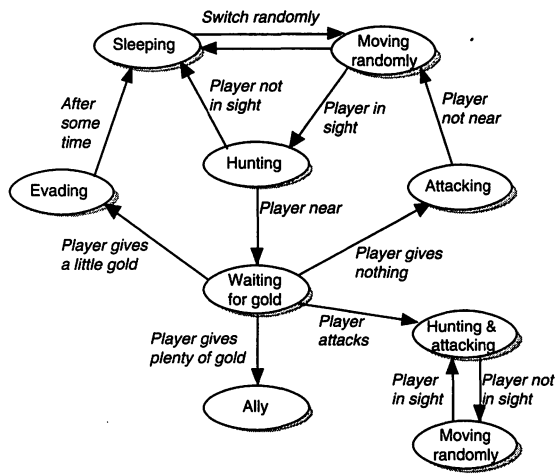


Figure 8.2 *State model for a fairly smart monster.*

That's plenty of states, and the monster ends up appearing pretty smart. You can't discuss philosophy with it, but it does more than the Marathon aliens which you can only kill.

In the top two states, the monster behaves just like the Marathon aliens, sleeping and walking around aimlessly. When the monster encounters the player, it hunts the player until they meet. Then it demands gold. At that state, there should probably be a branch to allow the player to run away.

If the monster gets a little gold, it's content and starts evading, just to get away from the player. When it has done that for a while, the monster falls back into the sleep and wander procedure again, the top two states.

If the monster gets lots of gold, it goes to the "ally" state. This is actually a "hunting" state, but in a friendly fashion. The monster will follow the player and help in battles.

If nothing happens within some time period, the monster decides that it wants to take the money by force and attacks. That attack will only continue until the player manages to run away, after which the monster will be back at the top states and demand gold again the next time they meet.

If, however, the player attacks, the monster changes into the two bottom right states. Now it is *pissed*! It's decided that the player has no good intentions and will only fight. It's also so angry that it will not sleep, but wander around looking for the player constantly. Note that it cannot get out of this state—it remembers!

State models of game entity behavior are very useful, but don't overdo it! A sprite can remember things without representing that information as states. A healthy mix between states and other memory-carrying variables is the best way to go.

So much for action games. Let's turn to game entity behaviors in board games, which is a *very* different topic.

Board Game Entities

In the games in the previous section, the game itself is the human player's opponent. Now I want to discuss games in which several players, some human and some artificial, participate in a game and follow the same rules. The game entities are part of the program, but not part of the game itself. I call this kind of game entity a *computer player*. For example, your opponent in chess is not the king, but the one who moves the black pieces. The chess pieces are not game entities. They take no actions on their own, but are moved by the computer player.

There are two kinds of games to discuss here. One is the kind you find in chess: The entire game situation is known. There is no element of chance except for the human player's moves. If the computer player had access to infinite processing power, it would always play the optimal game. Other examples of this deterministic type of game are Go, Tic-Tac-Toe, Checkers, Go-Moku, and Othello (also called Reversi).

The other kind of game is what you find in card games such as Bridge or Poker. You as a human player don't know how the cards are distributed, only what you have yourself. From the little information you have, you must select a card to play or a bid to make. Other

examples of this *indeterministic* type of game are Hearts, Gin Rummy, MahJong (which is a four-player game—the solitaire game that uses tiles is called Shanghai), Stratego, Monopoly, Civilization, the computer game Power-Players, and Railroad Tycoon.

Analyzing Deterministic Games

When you make a computer player for a deterministic game, the fundamental strategy is straightforward: Go through all the possible moves the computer player can make. For each of those moves, go through all moves that the human player can make. Then, for all these possible moves by the human player, examine all the moves the computer player can make, and so on. At some point, the computer player stops and decides the best move according to some criteria. This is called a search tree algorithm.

Figure 8.3 illustrates how this works. Each box is a game state, a configuration of game pieces. For each possible move, there are a number of countermoves. Figure 8.3 shows a game with very few possible moves for each turn. Real games have many more. In chess, for example, there are 20 different first moves, and even more possibilities in the turns thereafter.

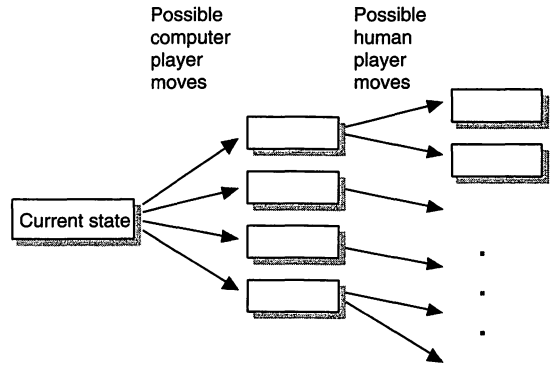


Figure 8.3 *Searching through the possible moves for two turns.*

Did you notice something with this approach? It grows exponentially! If there are ten different moves the human player can make each turn, first you get ten positions, from which the computer player can make ten different moves for each. That gives the human player a hundred positions after those two turns, from which to analyze a second move. For games with a large number of possible moves, a complete search of even a few turns is downright impossible.

What must be added is an analysis of the positions that are found. You find all possible moves and then you evaluate them. Does it look like there's any point in going to that particular position? If a win or lose situation is found, then the choice is easy—but it's the gray shades in between that are interesting.

Let's look at a simple modification of the search tree algorithm. Figure 8.4 shows a simple case. Four different moves are found. According to the goodness criterion used, one positions looks good, another okay, the third downright bad, and the last so-so.

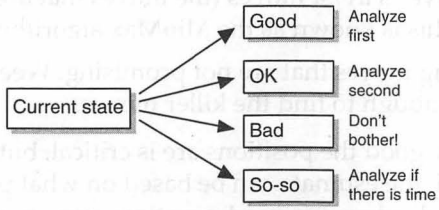


Figure 8.4 *Evaluate all moves to decide on which route to concentrate.*

The computer player now decides that it should have a good look at the Good case. It should check out the OK case and probably the So-so case too, but it can decide to skip the Bad case altogether. Chances are high that there are no good moves to find in that branch, although that depends on how you define the criteria.

Then the computer player goes on and finds all possible moves for the human player from the promising positions (see figure 8.5).

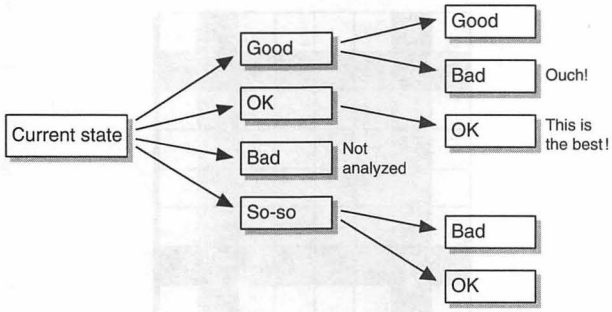


Figure 8.5 *The analysis after two moves.*

That second move consists of the human player's possible replies to the move the computer player could make. While the best situation is of interest in the first move, the worst is what should be examined here, since the human player will try to make the best of the situation. In figure 8.5, there are two possible moves after the move labeled Good. One of them leads to a position that is good for the computer player, and the other to a poor position.

The computer player assumes that the human player will put it in as much trouble as possible. Therefore, the Good position after the second move is not likely to happen, but rather the Bad one. Thus, the computer player finds that what at first appears to be the best move isn't so good after all! The OK move, however, can be countered only by a move that puts the game in a position that is still okay for the computer player. That looks most promising so far!

To summarize, the principles are

- The computer player assumes that the human player will pick the path that is worst for the computer player. For each move, the computer player searches for its own *best* moves and the human player's *worst* moves (the moves that are best for the human player). In AI textbooks, this is known as the MinMax algorithm.
- Don't waste time analyzing moves that are not promising. Weed out enough poor moves so you have time enough to find the killer moves.

Obviously, the estimate of how good the positions are is critical, but it is totally game-dependent. In Othello/Reversi, the estimate can be based on what positions each player occupies: Corners are very good, edges are good, positions near good positions are bad. But this must be done using numbers. What values should be attached to the different positions? How should you use them to calculate the final value? The easiest way is to try some formula and see if you agree with its results. If the formula makes decisions different than you would make, then it probably isn't good.

Figure 8.6 shows the starting point for the estimate calculator. Lighter spaces are good, making the estimate high, while darker ones are bad, resulting in a low estimate. This setup is based entirely on my own experience of playing Othello. You will learn that you are your own best source of evaluations.

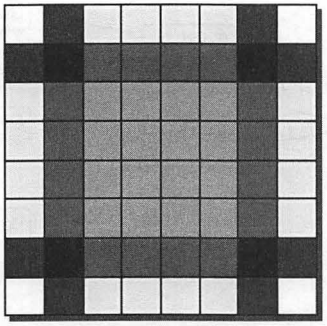


Figure 8.6 *A first draft for evaluating Othello.*

White (the corners) can represent a value of 10, for instance; the different gray shades can represent 5, 2, -2, and -5; and then -10 for the black. The value of a game state (a configuration) is then calculated as the sum of the values that your pieces cover.

It's time for an example. I picked the simplest one I could think of: Tic-Tac-Toe.

Tic-Tac-Toe

My Tic-Tac-Toe demo game (found in the 08 - Mechanics: Behavior folder on the CD-ROM) is stripped down to the minimum to make the code short—no event processing, no bells and whistles. It just plays one game of Tic-Tac-Toe and then quits. The only way to quit the game is to win or lose (see figure 8.7).

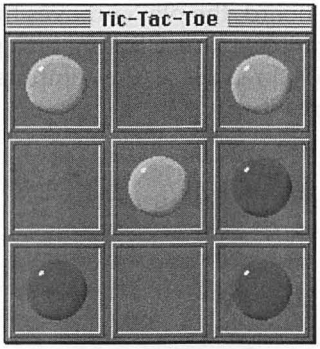


Figure 8.7 *The Tic-Tac-Toe demo.*

This demo plays the kind of Tic-Tac-Toe in which each player has three pieces. First, the players take turns placing pieces, and when all pieces are placed, the players take turns moving pieces until someone wins (or gets bored). Thus, it is not the same as the game you play on paper, in which you simply place pieces until the board is full.

Despite its simplicity, this demo serves the purpose of illustrating the principles described in the previous section. It searches through possible moves recursively, up to a given search depth.

There is an evaluation procedure that does only one thing: It checks if someone has won.

At the heart of the program are two almost identical routines: `FindComputerMove()` and `FindPlayerMove()`. `FindComputerMove()` is called after the player has moved to find a good move for the game entity. It tries all possible moves in the current state and assigns a value `stateValue`, which says how good the move is. There are three possible values: win, unknown, and lose. If a win is found, it is immediately returned. No more searching is necessary. The value is calculated using the `FindPlayerMove()` routine.

In order to evaluate a move, `FindPlayerMove()` makes exactly the same search, but for possible *player* moves, given the current opponent move. In this search, `FindPlayerMove()` calls `FindComputerMove()` with the modified game state. This is a recursive procedure. The routines keep calling each other until the depth reaches `kMaxDepth`.

Note that there must be enough stack space to allow the search depth you want. If you use a very large search depth, the search will take longer, but most importantly it may cause a stack overflow.

In the search, it's assumed that both players are trying to win, so the computer (that is, the Tic-Tac-Toe-playing game entity) avoids moves that let the player win, and assumes that the player does that too.

The game consists of a board—the player gets three green pieces, the computer opponent plays red. First, they take turns placing the pieces one by one on the board, in the player's case by clicking on an empty square; the computer opponent simply calculates a suitable

square. When all pieces are on the board, the player moves his or her pieces by clicking on the one to move and then on an empty square. Clicking on a green piece always selects that piece to be moved, while clicking on a red piece (the computer opponent's piece) has no effect. Whoever gets three pieces in a row wins.

Here's the source code behind it:

```
/* Tic-Tac-Toe -- game example for the Mac Game book*/
/* By Ingemar Ragnemalm 1995*/

#include <Sound.h>
```

We define some constants, as usual...

```
/*Size of the array*/
#define kArraySizeH 3
#define kArraySizeV 3

/*Size of the tiles*/
#define kTileSizeH 64
#define kTileSizeV 64

#define kNumTiles 9
```

The search depth defines how many steps the search should go. If 0, the entity will just test if it can win in the next move; if it can't, it will pick some arbitrary move.

```
/*Search depth -- higher gives a better computer player*/
/*Everything above 0 plays ok*/
#define kMaxDepth 2

/* A macro for taking the abs of a value */
#define abs(x) (x>0?x:-x)
```

The `TileState` variable defines the values that a tile can have.

```
/* All the possible states of a tile (space in the dungeon) */
typedef enum {empty, red, green} TileState;

/* The window pointer */
WindowPtr myWindow;
```

The `GameState` data structure holds all information about a game state—not necessarily the current one. I declare this as a data structure in order to pass it as a parameter in the search recursion.

```
/* Data structure describing the game board */

typedef struct
{
   TileState tiles[kNumTiles]; /* What is in each tile? */
   short   redCount, greenCount;
} GameState;
```

The current game state, the one the player sees on the screen, is stored in a global `GameState`.

```
GameState gGameState;
```

The player can mark tiles when moving them. The marked one is referred to by `gMarkedTile`.

```
/*Which tile is marked?*/
Point gMarkedTile = {-1,-1};

/* A Boolean telling if we should quit yet or not */
Boolean gDone = false;
```

Five pictures are needed: empty space, red piece, green piece, marked red piece, and marked green piece.

```
/* Pictures*/
PicHandle emptyTile;
PicHandle redTile;
PicHandle greenTile;
PicHandle redMarkedTile;
PicHandle greenMarkedTile;
```

Since I store the nine tiles in a one-dimensional array, I need the `Point2Index()` function to convert two coordinates to an index in the array.

```
/* A function that converts two coordinates to a value that can be used as index ➥
in the tile array*/

static short Point2Index(short h, short v)
{
   return h + v * kArraySizeH;
} /*Point2Index()*/
```

`DrawTile()` is similar to the routine with the same name in Dungeon. `DrawTile()`, however, checks if the tile being drawn is `gMarkedTile`, in which case it uses the marked symbol.

```
/* Draw a tile */

static void DrawTile(short h, short v)
{
```

```
  Rect tileRectangle;

  SetRect(&tileRectangle, h * kTileSizeH, v * kTileSizeV, (h + 1) * kTileSizeH, ➥
(v + 1) * kTileSizeV);
  switch ( gGameState.tiles[Point2Index(h, v)] )
  {
    case empty:
      DrawPicture(emptyTile, &tileRectangle); break;
    case red:
      if (h == gMarkedTile.h && v == gMarkedTile.v)
        DrawPicture(redMarkedTile, &tileRectangle);
      else
        DrawPicture(redTile, &tileRectangle);
      break;
    case green:
      if (h == gMarkedTile.h && v == gMarkedTile.v)
        DrawPicture(greenMarkedTile, &tileRectangle);
      else
        DrawPicture(greenTile, &tileRectangle);
      break;
    default:
      PaintRect(&tileRectangle);
  }
} /*DrawTile()*/
```

`IsLine()` checks if a line, given by starting point (`h`, `v`) and direction (`dh`, `dv`), is a winning line. It is used by `Analyze()`, below.

```
static Boolean IsLine(GameState state, short h, short v, short dh, short dv)
{
  short i;

  for (i=0; i<2; i++)
  {
    if (state.tiles[Point2Index(h + i * dh, v + i * dv)] == empty)
      return false;
    if (state.tiles[Point2Index(h + i * dh, v + i * dv)] !=
      state.tiles[Point2Index(h + (i+1) * dh, v + (i+1) * dv)])
        return false;
  }
  return true;
} /*IsLine()*/
```

`Analyze()` checks whether there is a winning combination in a game state. This is used both to check for end of game and to choose moves.

```
static Boolean Analyze(GameState state)
{
```

```
// The analysis here is just to check if there is a victory
// There are eight combinations:
   if (IsLine(state, 0,0,1,1))   // diagonal from top left
      return true;
   if (IsLine(state, 2,0,-1,1))  // diagonal from top right
      return true;
   if (IsLine(state, 0,0,1,0))   // first row
      return true;
   if (IsLine(state, 0,1,1,0))   // second row
      return true;
   if (IsLine(state, 0,2,1,0))   // third row
      return true;
   if (IsLine(state, 0,0,0,1))   // first column
      return true;
   if (IsLine(state, 1,0,0,1))   // second column
      return true;
   if (IsLine(state, 2,0,0,1))   // third column
      return true;
   return false;
} /*Analyze()*/
```

`InitTicTacToe()` is just like `InitDungeon()`—it creates a window and loads pictures. It also initializes `gGameState`.

```
/* Initialize -- create window, load graphics */

static void InitTicTacToe()
{
   Rect windowRectangle;
   short   i;

/*Set up the window*/
   SetRect(&windowRectangle, 50, 50, 50 + kArraySizeH * kTileSizeH, 50 + ➥
kArraySizeV * kTileSizeV);
   myWindow = NewCWindow(nil, &windowRectangle, "\pTic-Tac-Toe", true, 0, ➥
(WindowPtr)-1L, false, 0);
   SetPort(myWindow);

   qd.randSeed = TickCount (); /*Seed the random number generator*/

   emptyTile = GetPicture(131);
   redTile = GetPicture(132);
   greenTile = GetPicture(133);
   redMarkedTile = GetPicture(134);
   greenMarkedTile = GetPicture(135);

   for (i=0; i<kNumTiles; i++)
      gGameState.tiles[i]=empty;
   gGameState.redCount = 0;
```

```
   gGameState.greenCount = 0;
} /*InitTicTacToe()*/
```

`ValidMove()` is the same as in Dungeon, but a little simpler—it checks if a point is within the game grid.

```
/* ValidMove() checks if a tile clickedTile is inside the array bounds *and* near ➥
the player */

static Boolean ValidMove(Point clickedTile)
{
/* Valid tile?*/
   if ( clickedTile.h >= 0 )
      if ( clickedTile.v >= 0 )
         if ( clickedTile.h < kArraySizeH )
            if ( clickedTile.v < kArraySizeV )
               return true;
   return false;
} /*ValidMove()*/
```

Next are the twin routines `FindComputerMove()` and `FindPlayerMove()`. These routines define the game entity behavior; that is, they decide what moves the computer makes.

```
/*
FindComputerMove() and FindPlayerMove() are the two routines that perform the search.
They call each other, recursively, up to the search depth.
*/

// return 1 if win
// return 0 if undecided
// return -1 if lose

#define kWinningMove    1
#define kNoWinMove      0
#define kLosingMove     -1
#define kNoMove         -2

//prototype FindPlayerMove()
short FindPlayerMove(GameState state, GameState *returnState, short depth);
```

`FindComputerMove()` tries to find the best move for the computer from a given game state. It returns one of the values: -1, 0, or 1. The parameter state is the game state from which the move should be made. The state after the chosen move is returned is `returnState`. The parameter depth is the search depth, which is increased by 1 for every step into the recursion.

```
static short FindComputerMove(GameState state, GameState *returnState, short depth)
{
```

```
GameState tempState, junkState;
short   moveTo, moveFrom;
```

The variable `bestStateValue` tells what the rating is of the chosen move. To begin, it is set to `kNoMove`, which is lower than a loss.

```
short   bestStateValue = kNoMove;
short   stateValue;
```

The `returnState` is initialized to be a copy of state, just to make sure that something valid is always returned.

```
*returnState = state; // if all else fails
```

There are two different paths to follow, depending on whether the game entity has placed all its pieces yet. The number of red pieces played shows in the `redCount` field in the state data structure.

```
if (state.redCount < 3)
{
```

This branch is used when all tiles are not placed yet. Loop through all spaces and look for empty spaces.

```
  for (moveTo=0; moveTo<kNumTiles; moveTo++)
  {
    if (state.tiles[moveTo] == empty)
    {
```

A possible move has been found! Copy `state` into `tempState`, which is a working copy that may be modified. Perform the move by setting the appropriate space to red, then increase `redCount`.

```
      tempState = state;
      tempState.tiles[moveTo] = red;
      tempState.redCount++;
```

Is this a win? If it is, there's no reason to continue searching. Copy `tempState` to `returnState` and return a "win."

```
      if (Analyze(tempState))
      {
        *returnState = tempState;
        return kWinningMove;
      }
```

Okay, it wasn't a win. If depth is smaller than `kMaxDepth`, call `FindPlayerMove()` to learn what countermove the player is likely to make. The returned value is inverted to negative, so a win for the player is converted to a loss, and a loss to a win.

```
        if (depth < kMaxDepth)
          stateValue = - FindPlayerMove(tempState, &junkState, depth+1);
```

If the allowed search depth has been reached, there is no way to tell if the following moves will result in a win or a loss. Set the value of the move to `kNoWinMove`—neither win nor lose.

```
        else
          stateValue = kNoWinMove; // When we can't search further, set to ➥
undecided
```

If the move being analyzed is better than `bestStateValue`, the value of the previously chosen move, then the current move is copied into `returnedState` and its value to `bestStateValue` in order to make it the chosen move.

```
        if (stateValue > bestStateValue)
        {
          *returnState = tempState;
          bestStateValue = stateValue;
        }
```

If the values are equal, pick either of them at random. Note that this makes the last moves more likely than the first ones.

```
        else if (stateValue == bestStateValue)
          if (Random() > 0)
            *returnState = tempState;
      } //if empty
    } //for
  }//if
  else
  {
```

Now we've reached the second half of the routine! If all pieces have already been placed, this part is executed. A double loop is used to find all combinations of a red piece and an empty space—that is, all possible moves.

```
  for (moveTo=0; moveTo<kNumTiles; moveTo++)
    for (moveFrom=0; moveFrom<kNumTiles; moveFrom++)
    {
      if (state.tiles[moveTo] == empty)
      if (state.tiles[moveFrom] == red)
      {
```

The state is copied to `tempState`, and the piece is moved.

```
tempState = state;
tempState.tiles[moveTo] = red;
tempState.tiles[moveFrom] = empty;
```

From this point on, it looks exactly as it did previously. `Analyze()` is called to check for a winning move. If the move isn't, then `FindPlayerMove()` is called. If the result is better than the previously chosen move, it is made the chosen move.

```
if (Analyze(tempState))
{
   *returnState = tempState;
   return kWinningMove;
}
if (depth < kMaxDepth)
   stateValue = - FindPlayerMove(tempState, &junkState, depth+1);
else
   stateValue = kNoWinMove;
if (stateValue > bestStateValue)
{
   *returnState = tempState;
   bestStateValue = stateValue;
}
else if (stateValue == bestStateValue)
   if (Random() > 0)
      *returnState = tempState;
   } //if empty
  } //for

 }//if else
```

If the program reaches this point, the `returnedState` is already set to the best move it could find. Return `bestStateValue`.

```
  return bestStateValue;
} /*FindComputerMove()*/
```

`FindPlayerMove()` is almost identical. The only difference is that it handles green pieces instead of red ones. The same comments as before apply.

```
//Same routine but for finding the best player move
short FindPlayerMove(GameState state, GameState *returnState, short depth)
{
  GameState tempState, junkState;
  short   moveTo, moveFrom;
  short   bestStateValue = kNoMove;
  short   stateValue;
```

```
   *returnState = state; // if all else fails

   if (state.greenCount < 3)
   {
      for (moveTo=0; moveTo<kNumTiles; moveTo++)
      {
         if (state.tiles[moveTo] == empty)
         {
            tempState = state;
            tempState.tiles[moveTo] = green;
            tempState.greenCount++;
            if (Analyze(tempState))
            {
               *returnState = tempState;
               return kWinningMove;
            }
            if (depth < kMaxDepth)
               stateValue = - FindComputerMove(tempState, &junkState, depth+1);
            else
               stateValue = kNoWinMove; // When we can't search futher, set to ➥
undecided
            if (stateValue > bestStateValue)
            {
               *returnState = tempState;
               bestStateValue = stateValue;
            }
            else if (stateValue == bestStateValue)
               if (Random() > 0)
                  *returnState = tempState;
         } //if empty
      } //for
   }//if
   else
   {
   for (moveTo=0; moveTo<kNumTiles; moveTo++)
      for (moveFrom=0; moveFrom<kNumTiles; moveFrom++)
      {
         if (state.tiles[moveTo] == empty)
         if (state.tiles[moveFrom] == green)
         {
            tempState = state;
            tempState.tiles[moveTo] = green;
            tempState.tiles[moveFrom] = empty;
            if (Analyze(tempState))
            {
               *returnState = tempState;
               return kWinningMove;
            }
```

```
            if (depth < kMaxDepth)
               stateValue = - FindComputerMove(tempState, &junkState, depth+1);
            else
               stateValue = kNoWinMove;
            if (stateValue > bestStateValue)
            {
               *returnState = tempState;
               bestStateValue = stateValue;
            }
            else if (stateValue == bestStateValue)
               if (Random() > 0)
                  *returnState = tempState;
         } //if empty
      } //for


   }//if else

   return bestStateValue;
} /*FindPlayerMove()*/
```

`PlayerMove()` handles a click from the player. If a move is made, it also calls `FindComputerMove()` and makes the computer's move.

```
/* Try to move the player to the position where we clicked. */

static void PlayerMove(Point clickedTile)
{
   short h, v;
   short tileIndex;
   Point oldMarkedTile;
   GameState newState;
   short   computerMoveResult;
   OSErr   err;
```

First of all, the coordinates in `clickedTile` are converted to an array index.

```
   tileIndex = Point2Index(clickedTile.h, clickedTile.v);
```

Next, check if it's a valid move.

```
/* Valid move?*/
   if (ValidMove(clickedTile))
/* Yes! What is there? */
   {
```

Here, too, the program must select one out of two different branches: one if there are pieces left to place, and another if all pieces are placed.

```
    if (gGameState.greenCount < 3)
    {
```

If all pieces aren't placed, only a click on an empty space is valid.

```
/* All pieces are not placed yet. Click must be in an empty space! */
       if (gGameState.tiles[tileIndex] != empty)
       {
          SysBeep(1);
          return;
       }
```

If the click is on a space, then the piece is placed and `greenCount` is incremented.

```
       gGameState.tiles[tileIndex] = green;
       gGameState.greenCount++;
       DrawTile(clickedTile.h, clickedTile.v);
```

If this is a win, play the victory sound and set the `gDone` flag.

```
       if (Analyze(gGameState))
       {
          // Victory! Play a sound and set gDone!
          err = SndPlay(nil, (SndListHandle)GetNamedResource('snd ', "\pOK, you ➥
won"), false);
          gDone = true;
          return;
       }

    }
    else
    {
```

If all tiles are placed, there are two valid clicks: on an empty space or on a green piece. If on a green piece, then that piece is marked as the currently marked piece, for moving with a second click.

```
/* All pieces are placed. */
       if (gGameState.tiles[tileIndex] == green)    // Mark a tile for later movement
       {
          oldMarkedTile = gMarkedTile;
          gMarkedTile.h = -1;                       // Make gMarkedTile invalid!
```

```
            DrawTile(oldMarkedTile.h, oldMarkedTile.v);        // Redraw old marked ➥
tile, if any
            gMarkedTile = clickedTile;                         // Set new gMarkedTile
            DrawTile(gMarkedTile.h, gMarkedTile.v);            // Draw new marked tile
            return;
         }
```

If the click is on an empty space and some piece has been marked before, a move is made.

```
         else // Move a tile that was marked before
         if (gGameState.tiles[tileIndex] == empty)
         {
            if (ValidMove(gMarkedTile))
            {
               gGameState.tiles[Point2Index(gMarkedTile.h, gMarkedTile.v)] = empty;
               gGameState.tiles[tileIndex] = green;
               oldMarkedTile = gMarkedTile;
               gMarkedTile.h = -1;
                  // Make gMarkedTile invalid!
               DrawTile(oldMarkedTile.h, oldMarkedTile.v);
                  // Redraw old marked tile, if any
               DrawTile(clickedTile.h, clickedTile.v);
                  // Draw moved tile

               if (Analyze(gGameState))
               {
                  // Victory! Play a sound and set gDone!
                  err = SndPlay(nil, (SndListHandle)GetNamedResource('snd ', "\pOK, ➥
you won"), false);
                  gDone = true;
                  return;
               }
            }    // end valid gMarkedTile
            else return;
         }       // end click in empty space
         else
/* Click in red tile. Beep and return. */
         {
            SysBeep(1);
            return;
         }
      }          // end if 3 green else
```

The program gets to this point if and only if a player move has been carried out. Call `FindComputerMove()` to find a move.

```
      // If we get here, the player has made a valid move!
      computerMoveResult = FindComputerMove(gGameState, &newState, 0);
```

The move is made by copying the returned state into `gGameState` and redrawing all tiles.

```
      gGameState = newState;
      gMarkedTile.h = -1;
      /*Draw all tiles!*/
      for ( h = 0 ; h < kArraySizeH ; h++)
        for ( v = 0 ; v < kArraySizeV ; v++)
          DrawTile(h, v);
```

Check if this results in a win. If so, it must be a computer victory.

```
      if (Analyze(gGameState))
      {
        // Computer victory! Play a sound and set gDone!
        err = SndPlay(nil, (SndListHandle)GetNamedResource('snd ', "\pHaha, I ➥
won"), false);
        gDone = true;
        return;
      }

    }    // end ValidMove

} /*PlayerMove()*/
```

`InitToolbox()` is the standard inits. See the source code of other examples. After `InitToolbox()` follows the main program.

```
/* Main program */

void main(void)
{
  Point   clickPoint, clickedTile;
  short   h, v;
```

Call initialization procedures.

```
  InitToolbox();
  InitTicTacToe();
```

Redraw all tiles.

```
/*Draw all tiles!*/
  for ( h = 0 ; h < kArraySizeH ; h++)
    for ( v = 0 ; v < kArraySizeV ; v++)
      DrawTile(h, v);
```

Run the game loop. This should be quite familiar from Dungeon 1.

```
/*Initializations done! Run the game loop until the game ends.*/
  do
  {
    if (Button())
    {
      GetMouse(&clickPoint);        /* Get the position of the click */
      clickedTile.h = clickPoint.h / kTileSizeH;          /* Convert to grid. */
      clickedTile.v = clickPoint.v / kTileSizeV;
      PlayerMove(clickedTile);      /* Try to move there */
      while (Button());             /* Wait until the mouse click ends */
    }
  } while (! gDone);

  FlushEvents(mDownMask, 0);        /* Get rid of mouse-down events! */
} /*main program*/
```

Try recompiling Tic-Tac-Toe with different values for `kMaxDepth`. Of course, this is such a simple game that the computer player goes from dumb to unbeatable pretty quick. Still, the technique is useful for far more complex games.

Analyzing Indeterministic Games

So far we've talked only about deterministic games, games in which the computer player is allowed to know everything. Creating a computer player in a game with *incomplete* information is quite different.

In those games, there is usually no way to analyze all possible countermoves to a move. It is theoretically possible to go through all possible situations that the game *may* have, as well as all possible results of all random events. But even if you want to analyze only all possible countermoves to a move, this goal quickly becomes unreasonable.

The computer player must analyze the *facts* it has, what it has itself and what it knows about the other players, and decide upon an action based on that. For example, one of the problems when making a MahJong game is how to pick a tile to discard. (In case you're unfamiliar to MahJong, it's a Chinese game similar to Gin Rummy, using tiles rather than cards.)

I won't go into great detail here, but just outline how such a game can work. Let's start with Bridge. I assume that you know the basics of Bridge. If you don't, just skip this short example.

Given a hand of 13 cards, you have to decide what to bid. Ask yourself how you would do it yourself. Can you break down your own strategy into rules that can be put in a computer program? Usually, at least with some simplifications, you can!

Start by counting the high cards and the length of each suit. You can, like in the Culbertson bidding system, figure out a measure of how good the hand is by giving each card a point value. Give four points for each Ace, three points for a King, two points for a Queen, and one point for a Jack. (This point count is not exactly the original Culbertson system, but a simplification that is easier to handle.)

Then, have a look at the length of each suit. In the Culbertson system, you are given one point for each two-card suit, two points for a one-card suit, and three points for a suit you don't have at all. There are a few more rules, but this is enough for now.

That sum is enough to decide if you should pass or give an opening bid. If it is 13 or more, go ahead and bid.

What suit should you bid? You generally bid the longest suit, but there should be some high cards in it. You can pick some rule that you find appropriate, such as demanding at least a queen or skipping the color, or making a point value for each color. I would consider something like this: the number of cards times, say, four, plus the point values of the high cards in that suit. Finally, add a rule saying that you shouldn't bid a suit with less than four cards.

Get the picture? You must make some formula for evaluating the situation. That formula is best based on your own strategy when playing the game. You should make the game entity "think" the way you do!

Let's pick one more example before leaving this topic. I mentioned MahJong, the Rummy-related Chinese tile game. (No, I don't mean Shanghai—that's a solitaire game using the same tiles.) One of the Mac implementations of the game is shown in figure 8.8.

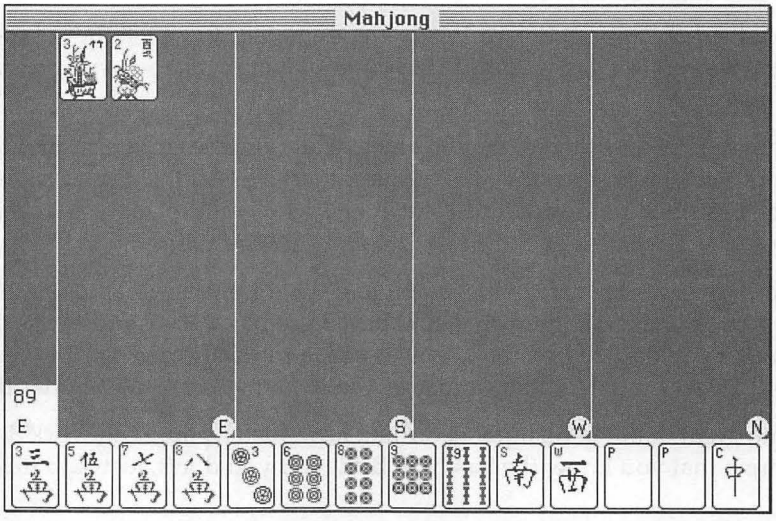


Figure 8.8 *A screen shot from the shareware game Mahjong by Matti Larborn.*

Every time it's your turn, you pick up one tile and then discard one. When making a MahJong-playing game entity, one of the problems it must be able to handle is picking a tile to discard.

Have a look at the hand in figure 8.8. If you know MahJong, you can probably suggest a few tiles that could be discarded. Much of what you think can be expressed as an algorithm for a computer. For each tile, you can calculate a rating. Are there any tiles of the same kind? If there is a pair, it can become a three-of-a-kind later. Are there any tiles in the same suit with a number one or two steps away? Those tiles are or can become a sequence. If there are no related tiles in the hand, the tile is a good candidate for trashing.

So, let's take the hand and decide on a point value for each tile. As a first draft, I use the following algorithm for each tile:

- Add one point if the tile is a 1, 9, or wind (wind tiles give more score).
- Add two points if the tile is a dragon (dragon tiles give even more).
- For every tile of the same kind, add two.
- For every tile of the same suit one or two steps away, add one.

No, this isn't perfect, but it is a start. Figure 8.9 shows what rating each tile gets.

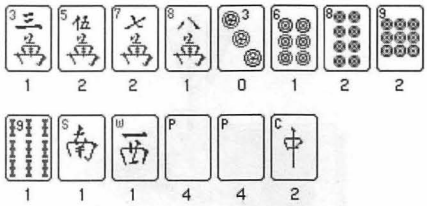


Figure 8.9 *Numeric evaluation of a MahJong hand.*

The 3 of circles has no excuse for staying. Off it goes. This is a good result, since that's what many human MahJong players would have done too. If you are a MahJong player, you probably argue that it's wrong that a completed straight gets the same rating as a single dragon. A good MahJong-playing game entity would use more than just this rating. It would check if tiles are part of complete combinations and also take into account tiles already discarded. If you have a 5 and a 7 in some suit, but all 6's are used in other combinations by other players or discarded, then the two tiles can never become a sequence.

So, the conclusion about these kinds of games is that it's hardly possible to make a detailed analysis in several steps, since there are too many possible countermoves. Instead, the problem can be approached in this way:

- Make measures, assigning a numerical value to each possible move the game entity can make. This may include sophisticated game-specific algorithms for analyzing the situation.

- Use other information such as what cards/tiles the other players have played or discarded, what bids they have made, and so on.
- Add special rules for special situations.
- Make the game entity do what you would have done by breaking down your own strategy into rules and measures!

Collision Detection

The most fundamental feature for making game entities interact in the appropriate way is *collision detection*. Collision detection is needed to make objects bounce off each other, to make shots and bonus objects detect when they hit a target, and many other uses.

A collision occurs when two game entities overlap. However, what "overlapping" means depends on the game world and the game entity representation. It usually means that two objects overlap as presented in image space (see figure 8.10).

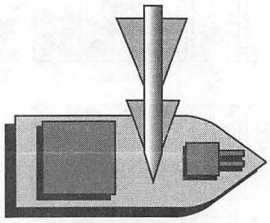


Figure 8.10 *Two game entities—in this case a boat and a rocket—overlap.*

From the top-view perspective, the boat and the rocket in figure 8.10 overlap. If the game in question is strictly a 2-D game, this is indeed a collision; but if it's a 3-D game with a height variable for each object, this might not be a collision at all.

Collision Detection between Sprites

When you're using sprites, each sprite has a position specified in screen coordinates. Thus, it is natural to do the collision detection in screen coordinates.

When you're doing collision detection, every sprite must be checked against all sprites that it can collide with, which for many games means *all* other sprites. This can be a fairly large number of checks for every frame. That means that you have to be careful and avoid making time-consuming operations if they aren't necessary.

Given two sprites to check for collision, the first thing to do is check if their bounding boxes overlap—to be more precise, if the bounding boxes of their current faces overlap. This does not necessarily mean that the sprites' faces overlap. Figure 8.11 shows the rocket and the boat in different positions in which the bounding boxes overlap but the images (sprite faces) themselves do not.

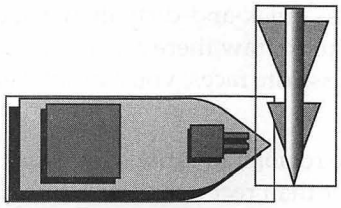


Figure 8.11 *The rocket and the boat in positions in which the bounding boxes overlap but the sprite faces don't.*

Once an overlap between the rectangles has been detected, there are several options.

Some games use rectangles only. A trick to get fewer false collisions is to use smaller rectangles instead of the bounding boxes (see figure 8.12).

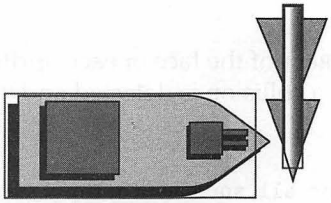


Figure 8.12 *With smaller rectangles, the collision detection gives fewer false collisions.*

If you have used my Sprite Animation Toolkit (SAT), you might recognize this trick. (You can find SAT on the enclosed CD-ROM, in the Libraries folder within the More Source folder.) In the sprites in SAT, there is a `hotRect` field, the "hot rectangle," which gives a rectangle to use for collision detection.

In some special cases, it's possible to calculate whether the collision is a hit or not. A particularly simple case is a collision between two circular objects. In that case, you have to calculate the distance between the centers and check if it is small enough so that the objects must be touching (see figure 8.13).

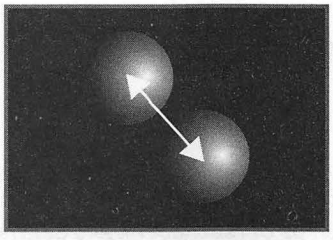


Figure 8.13 *Collision detection between two circular objects can be done by measuring the distance between the centers.*

However, there is a better way, less quick-and-dirty than just checking rectangles and more general than circular objects. In QuickDraw there are *regions*. If you have a region of the same size and shape as each of the sprite faces, you can call `SectRegion()` to learn if two sprite faces overlap.

It would be unreasonable to make region operations for every collision detection, because region operations are much slower than rect operations, but if a check with rectangles is made first, the region test has to be done only for the (usually) few sprites that do overlap.

The Collision Detection in SpriteEngine

Let's have a look at SpriteEngine to see how it handles collisions. In SpriteEngine.c, in the `RunSpriteEngine()` routine, you find the "standard" collision detection. It just checks whether the two sprites overlap or not based on the bounds of their faces.

It makes a double loop through the sprite list and checks each sprite against all sprites that are *after* that sprite in the list.

The bounds are found as the `portRect` of the face of each sprite offset by the position of the sprite. If these rectangles overlap, a collision is detected and `HitSprite()` is called.

```
/*3: Check for collisions*/
  theSprite = gSpriteList;
  while (theSprite != nil) /*For all sprites in the list…*/
  {
     bounds1 = theSprite->face->portRect;
     OffsetRect(&bounds1, theSprite->position.h, theSprite->position.v);

     anotherSprite = theSprite->next;
     while (anotherSprite != nil)
     {
     /*compare its position to all following sprites*/
        bounds2 = anotherSprite->face->portRect;
        OffsetRect(&bounds2, anotherSprite->position.h, anotherSprite->position.v);

        if (SectRect(&bounds1, &bounds2, &tmpRect))
           HitSprite(theSprite, anotherSprite);

        anotherSprite = anotherSprite->next;
     }
     theSprite = theSprite->next;
  } /*collision loop*/
```

Note

Note that I use QuickDraw calls such as `OffsetRect()` and `SectRect()`. It is possible to speed up the collision detection a little bit by calling the trap addresses directly or by reimplementing them yourself. See appendix A, "Top Speed—or Not?"

In `HitSprite()`, it's then possible to go one step further by calling `RegionHit()`. `RegionHit()`, which you find in SpriteTools.c, copies the mask region of each face and offsets it by the position of the sprite. `SectRgn()` calculates the intersection of the two regions. If the result is not an empty region, the sprites overlap and `RegionHit()` returns TRUE.

```
/* Collision test using regions! */

Boolean RegionHit(SpritePtr theSprite, SpritePtr anotherSprite)
{
  RgnHandle faceRegion1, faceRegion2;
  Boolean result;

  faceRegion1 = NewRgn();
  faceRegion2 = NewRgn();

  CopyRgn(theSprite->face->clipRgn, faceRegion1);
  OffsetRgn(faceRegion1, theSprite->position.h, theSprite->position.v);

  CopyRgn(anotherSprite->face->clipRgn, faceRegion2);
  OffsetRgn(faceRegion2, anotherSprite->position.h, anotherSprite->position.v);

  SectRgn(faceRegion1, faceRegion2, faceRegion1);
  result = !EmptyRgn(faceRegion1);

  DisposeRgn(faceRegion1);
  DisposeRgn(faceRegion2);

  return result;
} /*RegionHit()*/
```

`RegionHit()` can be optimized a little by not allocating and disposing of regions every time. Also, region operations such as `SectRgn()` may take some time for complex regions. However, this isn't one of the most time-critical operations in SpriteEngine, since it is preceded by a rectangle-based collision detection.

You now know everything you need about collision detection. I'll just make some additional notes on special situations that may be of interest.

Collision Detection in a Grid

Detecting collisions in a strictly grid-based game is trivial enough: If two entities have the same position, they collide. This is the "collision detection" used in Dungeon. You can either compare the positions of the player to all other entities or check in the grid.

However, in grid-based sprite games, in which there is an underlying grid but the sprites are not bound to it, it's also possible to use the grid for collision detection.

Previously I proposed that you can make an initial check between rectangles, followed by a more precise check with regions. This is efficient because most sprites don't collide, so only very few sprites would check with regions in a given frame. The grid can also be used to

make the collision detection more efficient. Each sprite can mark its presence in the grid space(s) in which it is located. Other sprites can inspect that information and learn whether they have to do any more collision detection work.

Remember, speed is limited by what you do many times for each frame. If you can avoid many-to-many collision checking, you might be able to squeeze out yet a few more frames per second!

Collision Detection in 3-D

For 3-D games, it's not enough to check whether the sprite faces overlap. Something more is needed using the depth information.

The most straightforward method is to have a bounding box in 3-D, so each object has a distance to the screen and a depth of the object. These numbers, unlike the bounding box on the screen, are unrelated to the graphics and are significant only for the game mechanics.

You can also simplify the game a bit by deciding that all objects have the depth 1. Then you just check if the distance from the screen is the same for both objects. This may work for some games, especially games in which sprites move in layers rather than freely. Take for example the pinball game Crystal Caliburn. Sprites can move in the transparent ramps as well as under them. All that is needed to tell whether two balls with intersecting faces collide or not is to check if they are in the same layer, either both in a ramp or both below it.

What you cannot easily do is make a full shape check for any possible shape. You must accept doing it with boxes or calculated shapes (such as spheres).

Image Space Collision Detection

Some games inspect pixel values in the screen image to detect collisions. I have one word to say about this: Don't.

Okay, I'll give some reasons. Image space collision detection works well only in games in which you use very few colors. Ideally, all objects should have only one color, so that you can get a pixel and look up the object that you have found. That might have been okay ten years ago, when games weren't so colorful, but today...we want better games than that.

Also, image space collision detection is generally not faster than doing the detection in object space. Checking if two rectangles overlap is very quick. Checking one pixel is also quick... but how many pixels do you have to check to be sure? Start counting. It won't be worth it.

Collision Handling

Once a collision is detected, what should be done? Yes, it depends on the game.

In some games, all a sprite needs to know is that it's been hit, say, by an enemy sprite. When that happens, the sprite should be removed, perhaps replaced by an explosion. Space Invaders works this way.

In other games, the sprites must know what kind of sprite hit them and take some action depending on that. For example, in the Collision II demo in SAT, the apples can collide with the player. If they are green (fresh), they are eaten, and if they are red (rotten) the player sprite makes a disgusted frown. However, a green apple can also collide with a red apple, and if that happens, the green apple turns red.

You will find yet another equally simple example in my game Bachman. The bonus objects can be eaten either by the player sprite or by the tomato-heads. If the player sprite eats a bonus object, it gets a score and might get extra speed, but if a tomato-head eats it, it stops to digest. That is, there are several kinds of actions depending on what sprites hit each other.

Separating Overlapping Objects

A somewhat more interesting handling of collision is needed in games in which objects are not supposed to overlap, but must bounce off each other. This doesn't have to imply any physics models at all, but merely moving the sprites away from each other. This is easiest with rectangular sprites.

For example, suppose that two sprites collide and one of the two can be considered heavier, so the other should move away. This is a very simple "physical model." Speeds and mass measures are ignored. All that count are the rectangles.

Figure 8.14 shows an example, a car colliding with a trashcan. This is not a case in which either of the objects should explode—unless, perhaps, it happens in a James Bond game—so the result of the collision should be displacing the trashcan. Disregarding any changes in the speed of the two objects (I will return to that shortly), one problem is whether to move either or both objects to positions where they no longer overlap.

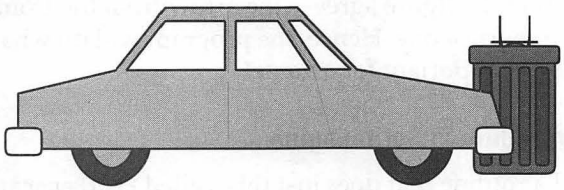


Figure 8.14 *A car and a trashcan overlapping each other, a collision that should result in movement changes rather than explosions.*

Since the trashcan is the smaller, lighter object, it makes sense to displace it rather than the car. One could also consider moving both objects away from each other, but that is a minor variation of what I suggest here. Now, where should the trashcan go?

There are two approaches to this problem. The trashcan could move in the same direction in which the car is moving. However, that assumes that there is a speed vector for the car that can be used. If the collision results in movement of the trashcan, it can be moved along its new speed vector until the objects no longer overlap.

An approach that is simple and more generally usable—especially in cases in which there are no physical models, no speed vectors, and so on—is to move the trashcan the *shortest distance* that will make it free from the car.

Calculating such a distance in a general case is a hard problem for which a game has neither need or time. The problem can be simplified by using only the bounding boxes of the shapes. That makes it a lot easier. There are only four possible directions to move a rectangle to make it free from another rectangle, namely up, down, left, or right.

Figure 8.15 shows how these distances are calculated from the bounding boxes of the two objects. Each measure is the difference between either the top of one object and the bottom of the other, or the left and right sides.

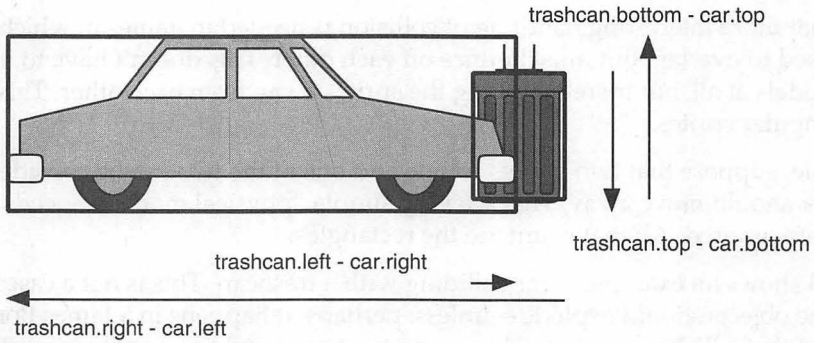


Figure 8.15 *Calculating the shortest distance for moving the trashcan away from the car.*

Even before this, it was obvious to the eye that the trashcan should move to the right to be free of the car. The arrows in the figure agree—the arrow from the front of the car to the left side of the trashcan is the shortest one. Hence, the program will do what makes sense to the eye—something that is very important for a game!

The `RectSeparate()` Procedure in SpriteEngine

In SpriteEngine you find a routine that does just this, called `RectSeparate()`. You can find it in the SpriteTools.c file.

`RectSeparate()` first calculates the bounds of each sprite by taking the `portRect` of each face and offsetting it by the position of the sprite. This is exactly what I did in the collision detection earlier, but since I didn't pass these rectangles along, I have to do it again.

```
/* Moves two sprites apart to separate them with respect to their bounding boxes. */

short RectSeparate(SpritePtr theSprite, SpritePtr anotherSprite)
{
```

```
short   distance[4], shortest, shortestDistance, i;
Rect bounds1, bounds2;

bounds1 = theSprite->face->portRect;
OffsetRect(&bounds1, theSprite->position.h, theSprite->position.v);

bounds2 = anotherSprite->face->portRect;
OffsetRect(&bounds2, anotherSprite->position.h, anotherSprite->position.v);
```

The next step is to calculate the four distances. They are put in an array to make the search for the smallest easier.

```
/*Calculate the distance to separate the sprites in every direction*/
  distance[0] = bounds2.top - bounds1.bottom; //up
  distance[1] = bounds2.bottom - bounds1.top; //down
  distance[2] = bounds2.right - bounds1.left; //right
  distance[3] = bounds2.left - bounds1.right; //left
```

The smallest distance is found by looping through the array. Note that the `abs` operation must be used, removing the sign of the distances when comparing them.

```
/*Find the shortest distance*/
  shortest = 0;
  shortestDistance = abs(distance[0]);
  for (i=1; i<4; i++)
  {
    if (abs(distance[i]) < shortestDistance)
    {
      shortest = i;
      shortestDistance = abs(distance[i]);
    }
  }
```

The shortest distance is then added to the appropriate component of the position of one of the sprites.

```
/*Move the sprite in the appropriate direction*/
  switch (shortest)
  {
    case 0:
    case 1:
      theSprite->position.v += distance[shortest]; break;
    case 2:
    case 3:
      theSprite->position.h += distance[shortest]; break;
  }
  return shortest;
} /*RectSeparate()*/
```

If you feel like experimenting a little bit, try to modify this routine so that it separates the sprites by moving both.

I will use this routine in a demo shortly, as soon as I've discussed elastic collisions.

Realistic Bounces

The most interesting problem when handling collisions, however, is how to make sprites bounce off each other in a realistic way, to calculate new speed vectors that make sense and agree with physical laws.

For the following section, I hope that you are comfortable with the relations between positions, speed, and acceleration, and that you have no problems with vectors, because there will be a lot of it. If you have some high school physics book available, it would not hurt to study its sections about mechanics and particle dynamics.

To handle a collision in a general case, you would have to take the shape of each object into account, changing the rotation of the objects as well as their speeds. That is far above the level of ambition here, as well as for most games. I will initially deal only with objects hitting each other head-on, so all movement is along a straight line. Later I will also deal with collisions between circular objects moving in arbitrary directions, but that is as far as I will go here.

There are two variables that are essential when calculating collisions, the mass of each object and its speed. If an object has the speed `v` and the mass `m`, there are two other variables that can be calculated from that, the momentum and the kinetic energy.

The momentum is calculated as the mass multiplied by the speed; that is...

$$p = m * v$$

The kinetic energy is the mass times the square of the speed, divided by two; that is...

$$e = \frac{v^2 * m}{2}$$

A physical law that should always hold in a collision is the following: The sum of the momentum of all objects is the same immediately before and after the collision. For two objects, that can be written as:

$$m_1 * v_{1b} + m_2 * v_{2b} = m_1 * v_{1a} + m_2 * v_{2a}$$

where m_1 and m_2 are the mass of each object, v_{1b} is the speed of object #1 before the collision, v_{1a} is its speed after the collision, and so on.

What about the energy, then? The energy is interesting in elastic collisions. If a collision is totally elastic, no energy is lost; in this case, the sum of the kinetic energy of the colliding objects should be the same immediately before and after the collision. The opposite is a

plastic collision, in which energy is lost. There are also semielastic collisions, in which just a little energy is lost; this result is somewhere between an elastic and a plastic collision.

Does it seem complicated? Don't worry. There are plenty of simplifications you can make that will make these problems much easier to handle.

Collisions between Very Large and Very Small Objects

A special case that is very easy to handle is the collision of two objects in which one object is very *large* and *stationary*, and one is very small in comparison. A rather extreme example is the earth and a small rubber ball. The collision does not affect the large object in any significant way. Only the small object is affected.

It's fairly obvious what happens. After an elastic collision, the small object has the same speed as before, but in the opposite direction. But after a plastic collision, the small object will stop at the large one. If a semielastic collision is wanted, some appropriate fraction (say, 90 percent) of the speed is preserved.

Let's look at an example collision (see figure 8.16). A rubber ball is thrown in sideways from the left, and an egg from the right, over a flat, horizontal concrete floor. This is a 2-D example with gravity involved, but the point I'm making is what happens with the vertical component of the speed.

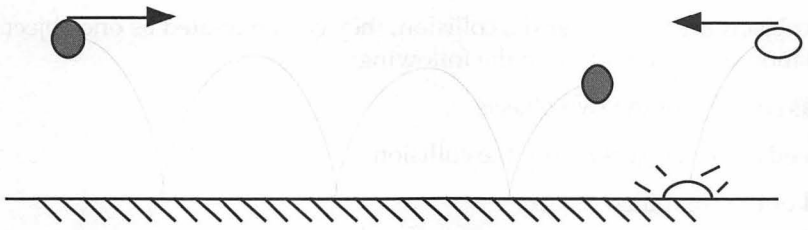


Figure 8.16 *A rubber ball and an egg behave differently when dropped on a floor.*

The rubber ball (coming in from the left) makes almost 100 percent elastic collisions with the floor, making it bounce almost as high every time. The egg (coming in from the right) makes a plastic collision and doesn't bounce at all. If it weren't for friction, however, it would still slide over the floor with the horizontal speed component.

For each of the objects, the collision can be handled with the following calculation:

```
speed.v = -speed.v * elasticity
```

The elasticity is a number between 0 and 1 and depends on the object. For the rubber ball it is close to 1, and for the egg it is 0.

In all the SpriteEngine demos, this kind of elastic collision is used for collisions with the borders of the window, though the actual collision handling is done in the routines

`KeepInside()` or `KeepInsideFixed()`. The effect is simple enough, though: The speed component in the direction that the collision is made is reversed.

Plastic Collisions

Way back in the tenth century, when the Viking Palnatoke took his bow and shot an arrow through an apple on the head of his son, the apple probably fell to the ground with the arrow in it. But Palnatoke's successor, Wilhelm Tell (in a more recent story), used a crossbow, so the crossbow bolt probably went straight through the apple.

In this section I deal with the first kind of collision, in which one object hits another and sticks in it. This is an example of plastic (inelastic) collision (see figure 8.17). As in an elastic collision, the momentum is preserved, but here the kinetic energy is not.

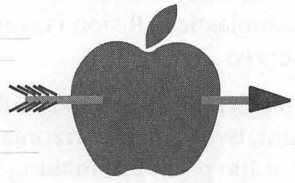


Figure 8.17 *Example of the result of a plastic collision.*

Since the two objects are stuck after the collision, they can be treated as one object. That simplifies the momentum equation to the following:

m_1, m_2 = the mass of each of the two objects

v_1, v_2 = the speed of each object before the collision

v_a = the speed of the combined object

$$m_1 * v_1 + m_2 * v_2 = (m_1+v_2) * v_a$$

which transforms to the following:

$$v_a = \frac{m_1 * v_1 + m_2 * V_2}{m_1 + m_2}$$

That formula is simple enough to use. If the two objects have the same mass, it gets even simpler:

$$v_a = \frac{v_1 + v_2}{2}$$

The conclusion is that if two objects with the same mass collide and are stuck in each other, the resulting speed should be the *average* of what they had before.

Elastic Collisions

When two steel or rubber balls hit each other, for example in a pinball game, they bounce off each other. This is an elastic collision (see figure 8.18).

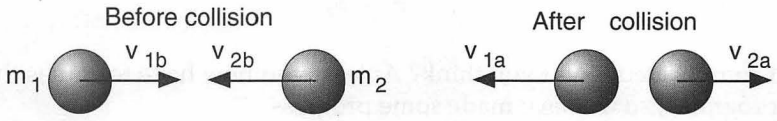


Figure 8.18 *Two objects before and after collision.*

In an elastic collision, not only the momentum but also the kinetic energy of both objects is preserved.

When two elastic objects are colliding, their mass and speeds are what matter. In the following formulas, I use these symbols:

m_1 = the mass of object 1

m_2 = the mass of object 2

v_{1b} = the speed of object 1 before the collision

v_{2b} = the speed of object 2 before the collision

v_{1a} = the speed of object 1 after the collision

v_{2a} = the speed of object 2 after the collision

As mentioned previously, the preservation of momentum gives the following formula:

$$m_1 * v_{1b} + m_2 * v_{2b} = m_1 * v_{1a} + m_2 * v_{2a}$$

The preservation of kinetic energy gives another, similar formula:

$$\frac{m_1 * v_{1b}^2}{2} + \frac{m_2 * v_{2b}^2}{2} = \frac{m_1 * v_{1a}^2}{2} + \frac{m_2 * v_{2a}^2}{2}$$

This follows directly from the formula for kinetic energy above. These two formulas can be combined to an equation of the second degree. Solving that demands something like a course in calculus at the university level, so I'm not going to show in detail how to solve it nor demand that you be able to solve it. However, if you know the math, don't let me stop you.

The solution to the equation turns out to be as follows:

$$v_{1a} = \frac{2 * m_2 * v_{2b} + v_{1b} * (m_1 - m_2)}{m_1 + m_2}$$

$$v_{2a} = \frac{2 * m_1 * v_{1b} + v_{2b} * (m_2 - m_1)}{m_1 + m_2}$$

It still feels a bit complicated, don't you think? At least you now have formulas that you can type into your programs, so we have made some progress.

In many games, you can make considerable simplifications. One that is very useful is using objects with the same mass. In a pinball game, for example, you certainly won't use balls of different sizes. Since the mass is just a measure for the game, there is usually no need to use a mass that corresponds to the mass of the object in real life, like a certain number of grams. I can decide to set the mass to 1 for all objects. If I do that, the formulas above become:

$$v_{1a} = \frac{2 * 1 * v_{2b} + v_{1b} * (1-1)}{1 + 1} = v_{2b}$$

$$v_{2a} = \frac{2 * 1 * v_{1b} + v_{2b} * (1-1)}{1 + 1} = v_{1b}$$

How about that for simplicity! The two objects just exchange speeds with each other.

Note

In an elastic collision between equally heavy elastic objects and along a straight line, each object gets the speed the other had before the collision.

This kind of collision handling, as well as the rectangular object separation procedure, are demonstrated in a SpriteEngine demo, SE RectBounce.

SE RectBounce

In SE RectBounce, three rectangular sprites glide around and bounce off each other when they collide. The sprites have different faces, but are otherwise identical, that is, they have the same behavior.

The `MoveSprite()` routine is totally trivial in this demo. It adds the speed to the position and calls `KeepOnScreen()` for the border check. Note that this demo uses integers, not fixed point.

```
void MoveSprite(SpritePtr theSprite)
{
   theSprite->position.h += theSprite->speed.h;
   theSprite->position.v += theSprite->speed.v;
   KeepOnScreen(theSprite);

} /*MoveSprite()*/
```

`HitSprite()` is not much more complex. It calls `RectSeparate()` to separate the objects. `RectSeparate()` returns an integer that tells in what direction it has moved the sprites. The speed components in that direction are swapped between the sprites.

```
void HitSprite(SpritePtr theSprite, SpritePtr anotherSprite)
{
   short   tempSpeed;

   if (RectSeparate(theSprite, anotherSprite) >= 2) // 2 or 3: horizontal, ➥
otherwise vertical
   {
      tempSpeed = theSprite->speed.h;
      theSprite->speed.h = anotherSprite->speed.h;
      anotherSprite->speed.h = tempSpeed;
   }
   else
   {
      tempSpeed = theSprite->speed.v;
      theSprite->speed.v = anotherSprite->speed.v;
      anotherSprite->speed.v = tempSpeed;
   }
} /*HitSprite()*/
```

That was easy, don't you think? The result is pretty pleasing too, although you would expect the objects to rotate. Unfortunately, it gets much harder if the objects are not rectangular, as you will soon see.

Two-Dimensional Elastic Collisions

So, is the problem of making realistic elastic collisions solved by swapping speeds? Not quite. It works only in one dimension, for objects moving along the same line. Observe the two objects in figure 8.19.

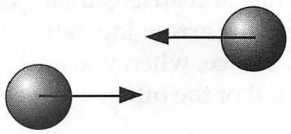


Figure 8.19 *Two objects that should not simply exchange speeds when colliding.*

They are not moving along the same line, that is, their centers are not moving toward each other. If you simply swap the speeds of two colliding objects, in many cases it will look right; but for "touches" like the one in figure 8.19, it will look wrong. If you are not happy with that, it takes a little more work to correct it. Get ready for some geometry—and beware, you might not find this easy to grasp!

Let's take a close look at the two balls at the moment they collide (see figure 8.20). I've put in two extra help lines, one through the centers of the balls and one parallel to the touching surfaces of the balls. I will call these lines the *perpendicular help line* and the *parallel help line*, respectively.

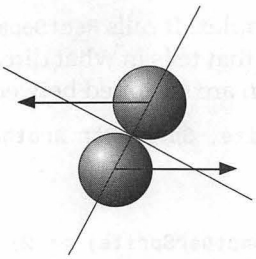


Figure 8.20 *Two colliding balls, with some help lines needed for precise calculation of the new speeds.*

Now let's concentrate on one of the balls. Its speed vector is not parallel to either of the two help lines. However, its speed vector can be split into two vectors or two components, each along each help line. This gives me one parallel component and one perpendicular component. The result is shown in figure 8.21.

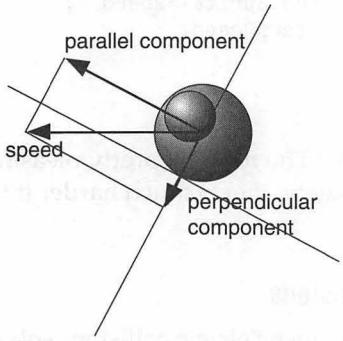


Figure 8.21 *The parallel and the perpendicular components of the speed vector.*

If you aren't used to vectors, a brief refresher might be helpful. Vectors have no fixed origin and no position, but describe a *direction* and an *amplitude*. Most commonly, vectors are displayed as arrows, with the head of the arrow indicating the direction and the length of the arrow indicating the amplitude. Hence, when you add two vectors, you just move them so the head of one vector meets the tail of the other.

To determine speed, if you have a force vector, you really need to know the mass the vector is being applied to, but in computer games, you can be less strict with your definitions. In reality, the images on the monitor have no mass, so I use what I call a "speed vector" and calculate speed instead of force. That way, the length of the arrow can directly relate to the distance the moving object travels.

If you add the parallel and perpendicular components to each other, the sum is the speed vector. You can verify that by starting at one point, moving the direction and distance that one component defines, and from there making another move in the direction and distance of the other component.

So, what is the point of splitting the speed vector this way? Well, the parallel component will not be affected *at all* by the collision. Assuming that there is no rotation in the balls, the impulse in a collision is *only* along the perpendicular help line.

Thus, given the two components, the collision can be handled by swapping the perpendicular components of the speeds but keeping the parallel components as they are. I can express this with symbols as well:

v_{1b}, v_{2b} = the two speed vectors before the collision

p_1, p_2 = the parallel components

n_1, n_2 = the perpendicular components

v_{1a}, v_{2a} = the two speed vectors after the collision

Again, the components are created by splitting the speed vectors before the collision. The speeds after the collision are calculated this way:

$$v_{1a} = p_1 + n_2$$
$$v_{2a} = p_2 + n_1$$

Figure 8.22 summarizes the process. In the left part of the figure, you see the speeds before the collision, which are the sum of the components. In the right part of the figure, you see the result. The vectors n_1 and n_2 are moved to the other ball and added to the p_2 and p_1 vectors, respectively; the resulting speed vectors are built.

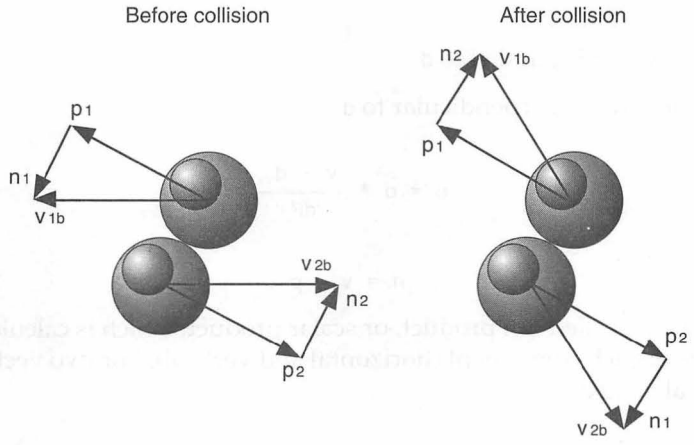


Figure 8.22 *The speed vectors before and after the collision.*

To get a better look at the result, I cleaned up figure 8.22 a bit and moved the speed vectors. The result is figure 8.23, which shows the trajectories the two balls have immediately before and after the collision.

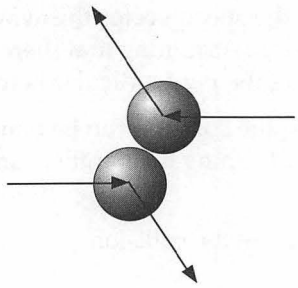


Figure 8.23 *The resulting trajectories.*

Doesn't that look neat? Actually, it looks absolutely great on the screen.

If you're still with me after all these vectors, congratulations! Vectors are a difficult topic; don't be discouraged if it's not clear the first time through. Alas, I'm not quite through yet. There is one more thing that I haven't explained yet: how to split a vector into its components.

To fully understand how to do it, you have to know some vector calculus. Don't worry, I'll show you just how to do it.

The formula for calculating the component of a vector that is parallel to another vector is as follows:

`v` = vector to be split

`d` = another vector

`p` = the component of `v` that is parallel to `d`

`n` = the component of `v` that is perpendicular to `d`

$$p = d * \frac{v \cdot d}{|d|^2}$$

$$n = v - p$$

The · operation is the so-called dot product, or scalar product, which is calculated as the sum of the products of each component (horizontal and vertical). For two vectors `a` and `b`, the dot product is calculated as:

$$a \cdot b = a.b * b.h + a.v * b.v$$

This vector splitting is easily implemented. The following code snippet takes the two vectors `v` and `d` and returns two vectors `p` and `n` as defined above.

```
void SplitVector(Point v, Point d, Point *p, Point *n)
{
   long length2, dotProduct;

   length2 = d.h * d.h + d.v * d.v;      /*Squared length of d*/

   dotProduct = v.h * d.h + v.v * d.v;   /*Scalar product*/

   (*p).h = d.h * dotProduct / length2;
   (*p).v = d.v * dotProduct / length2;
   (*n).h = v.h - (*p).h;
   (*n).v = v.v - (*p).v;
} /* SplitVector() */
```

I have made a demo using this technique to make some balls bounce in a realistic way. It uses SpriteEngine and is called SE Balls.

SE Balls

In SE Balls, three balls bounce around, making elastic collisions with each other. Fixed point arithmetic is used.

MoveSprite is very simple: Just add for gravity and add speed to `position` in fixed point fashion.

```
void MoveSprite(SpritePtr theSprite)
{
   theSprite->speed.v++; // Simple gravity
   theSprite->fixedPointPosition.h += theSprite->speed.h;
   theSprite->fixedPointPosition.v += theSprite->speed.v;
   theSprite->position.h = theSprite->fixedPointPosition.h >> 4;
   theSprite->position.v = theSprite->fixedPointPosition.v >> 4;
   KeepOnScreenFixed(theSprite);
} /*MoveSprite()*/
```

`HitSprite()` is where it all must happen. I call `RegionHit()` to make sure that there is a *real* collision.

```
void HitSprite(SpritePtr theSprite, SpritePtr anotherSprite)
{
   Point   perpendicularHelpLine, parallelHelpLine, p1, p2, n1, n2;

//Check for a *real* collision!
   if (RegionHit(theSprite, anotherSprite))
   {
```

The variable perpendicularHelpLine is calculated as the difference between the positions.

```
perpendicularHelpLine.h = theSprite->position.h - anotherSprite->position.h;
perpendicularHelpLine.v = theSprite->position.v - anotherSprite->position.v;
```

From perpendicularHelpLine I can find the parallel help line.

```
parallelHelpLine.h = perpendicularHelpLine.v;
parallelHelpLine.v = -perpendicularHelpLine.h;
```

SeparateBalls() is called to move them apart. After that, the new position is converted to fixed point and put in fixedPointPosition to make fixedPointPosition reflect the change.

```
//Move them away from each other}
     SeparateBalls(theSprite, anotherSprite);
//Make the fixed point positions follow!
     theSprite->fixedPointPosition.h = theSprite->position.h << 4;
     theSprite->fixedPointPosition.v = theSprite->position.v << 4;
     anotherSprite->fixedPointPosition.h = anotherSprite->position.h << 4;
     anotherSprite->fixedPointPosition.v = anotherSprite->position.v << 4;
```

Then, SplitVector() is called to split the vectors into one component parallel to parallelHelpLine and one that is perpendicular to it. The components of theSprite's speed are in p1 (parallel) and n1 (perpendicular), and the ones for anotherSprite go in p2 and n2.

```
//Swap the speed components that are parallel to "vector" (this allows for ➥
"touches," very}
//nice and realistic bounces)}
     SplitVector(theSprite->speed, parallelHelpLine, &p1, &n1);
     SplitVector(anotherSprite->speed, parallelHelpLine, &p2, &n2);
```

Swap the perpendicular component and keep the parallel one!

```
     anotherSprite->speed.h = p2.h + n1.h;
     anotherSprite->speed.v = p2.v + n1.v;

     theSprite->speed.h = p1.h + n2.h;
     theSprite->speed.v = p1.v + n2.v;
   } // if RegionHit
} /*HitSprite()*/
```

The subroutine SeparateBalls() separates the two balls from each other. This is honestly a hack that could be made better. What I do here is calculate the vector from one ball to the other (initVector coming up) and run a Bresenham line-drawing algorithm for moving the balls along initVector. I keep separating them until the distance between them is bigger than the constant kBallDiameterSquared.

The variables absH and absV hold signless versions of the initial distance between the balls. The signs of the distances are put in moveH and moveV.

The variable frac is the key to the algorithm. Each time the algorithm advances one step in the main direction (the larger of the horizontal and vertical components), the abs value in the other direction is added to frac. If frac exceeds abs in the main direction, a step is taken in the minor direction and frac is decreased again.

```
/* SeparateBalls() is explicitly written to separate two balls with diameter 32. */

#define kBallDiameterSquared (33*33)

static void SeparateBalls(SpritePtr theSprite, SpritePtr anotherSprite)
{
   Point initVector, nowVector;
   short absH, absV;
   short moveH, moveV;
   short frac;
```

Calculate the initial vector along which the backward Bresenham will run, as well as the abs and move variables.

```
   frac = 0;
   initVector.h = theSprite->position.h - anotherSprite->position.h;
   initVector.v = theSprite->position.v - anotherSprite->position.v;
   absH = abs(initVector.h);
   absV = abs(initVector.v);
   moveH = Sgn(initVector.h);
   moveV = Sgn(initVector.v);
```

Just in case the distance would be 0, one of the move variables is set to a non-0 value; otherwise the algorithm could lock up.

```
   if ( moveH == 0 )
      if ( moveV == 0 )
         moveV = 1;
   do
   {
```

If the horizontal component is biggest, I move one step per turn in the horizontal direction and less often in the vertical direction. Just how often depends on how much larger the horizontal component is than the vertical component.

```
      if ( absH > absV )
      {
         theSprite->position.h = theSprite->position.h + moveH;
         anotherSprite->position.h = anotherSprite->position.h - moveH;
         frac = frac + absV;
```

```
        if ( frac > absH )
        {
           theSprite->position.v = theSprite->position.v + moveV;
           anotherSprite->position.v = anotherSprite->position.v - moveV;
           frac = frac - absH;
        }
     }
```

If the horizontal component is lower, I move one step per turn in the vertical direction and less often in the horizontal direction.

```
     else
     {
        theSprite->position.v = theSprite->position.v + moveV;
        anotherSprite->position.v = anotherSprite->position.v - moveV;
        frac = frac + absH;
        if ( frac > absV )
        {
           theSprite->position.h = theSprite->position.h + moveH;
           anotherSprite->position.h = anotherSprite->position.h - moveH;
           frac = frac - absV;
        }
     }
```

Calculate the current distance. If it is smaller than the constant `kBallDiameterSquared`, continue until it is equal or larger.

```
     nowVector.h = theSprite->position.h - anotherSprite->position.h;
     nowVector.v = theSprite->position.v - anotherSprite->position.v;
  } while (!  (long)nowVector.h * nowVector.h + (long)nowVector.v * nowVector.v > ➥
kBallDiameterSquared);
} /*SeparateBalls()*/
```

That was a pretty long routine. If you understand what I was doing, you've just learned the principles of the Bresenham line-drawing algorithm. If you don't quite get it, you haven't missed anything that is critical to the collision-handling problem.

I'm sure you can find a better way to solve the separation problem. I could have even used the rectangle-based separation routine, but that would have separated the objects too far.

Now you can handle any kind of collision, right? Well, I haven't gone into collisions of objects with arbitrary shapes or the handling of rotation. However, that is beyond the needs of most games. What you know now is enough to create fairly impressive games.

Profile: Eric Johnston

Eric works for LucasArts and has sole programming responsibility for the *MacWorld* Game Hall of Fame-winning versions of Pipe Dream, Indiana Jones and the Last Crusade, Loom, and The Secret of Monkey Island, among other Mac conversion projects. He is currently technical lead on an entirely new real-time 3-D console project for LucasArts. He chose his career primarily because, in games, "looks cool" is justification enough to proceed with a project.

Q: When you guys do games, I hear LucasArts has this huge crack team of programmers that put these games together. Is that really how you work?

A: Typically, the structure of a game group for the first version of a game is unique. For the ports of a game, it's just one person. For the original version, you have: a project leader, who's in charge of the whole project; a producer, who's coordinating and making sure that facilities are available (and ordering pizza in the final hours); one or more programmers (between one and five, depending upon the project); and others, such as artists and musicians. There is one sound person usually and a bunch of artists, depending upon the games. A lot of games that LucasArts does are very art intensive because there's cartoon art, animation, and 3-D rendering that take a lot of man-hours. That's where most of the people on the projects are.

Q: Let's take Rebel Assault. How was that put together?

A: That was originally done by a programmer named Vince Lee, and it was built on a piece of technology, which was the ability to spool video off of a CD-ROM drive, which had previously been too slow to do. Now there are more companies finding good ways to do it. Vince came up with a way to not only spool the picture and the sound and the music off the CD, but also to warp the image so that it looked like you were flying the ship and actually had control. He built the underlying engine, and some 3-D artists and animators made the video clips for him. Everything from the pictures of the spaceships to the lasers to the canyon that you're flying in is a prerendered animation.

Q: When you do prerendered animation, is that like workstations working with a wireframe-type simulation, or what do you use?

A: We use a bunch of commercial and in-house tools to do all the modeling and rendering. It really depends on what the particular project is needing. I did the port of Rebel Assault so I did not see which of the modeling packages was used. I came in when it was almost ready to ship on the IBM.

Q: You're handling the Mac ports. What is there to porting stuff to the Mac?

A: The one thing that has just become possible is that the Quadras and PowerPCs are powerful enough to do full-screen drawings. The entire screen is changing

continued

every single time. That was difficult with IIci's because the resolution is so much higher on the Mac that it's a lot of information to move. The two key challenges for getting games from the PC to the Mac are resolution and sound—the fact that PCs use sound cards, whereas the Mac has to handle it on a CPU. Sound generation is still pretty significant overhead.

Q: How about QuickTime MIDI instruments?

A: That will take some development time away from the software developers, because as far as mixing digital instruments, that had to be done by the game programmer on the CPU. The other nice thing about QuickTime, if you're using the QuickTime Library that happens to take advantage of new hardware, is that the programmer doesn't have to know about the hardware and it will perform better.

Q: You mentioned moving more data around. Do you tweak `CopyBits()`, roll your own, or do you let QuickTime field some of this?

A: For Rebel Assault, I had to do some special things. I always keep a `CopyBits()` call around in case what I'm doing doesn't work on a particular Mac, but there are a couple of things that I do. One is line skipping, where I only draw every other line on a display, and because the display is a nice high resolution, it still works and it looks pretty normal.

Q: Sort of a pseudo interlace, like on a TV.

A: It's like you're looking at it through a horizontal comb, but it ends up being bright enough that if you don't know the breaks are there, you probably don't notice them at all. Since the PC games are low resolution when they're brought over, they look chunky and pixelated. There is something that was used in all of the story games and is also used in parts of Dark Forces. It's pixel smoothing, whereas the artwork is drawn to the screen, the corners get rounded and it's run through a filter which does real-time smoothing on it. The smoother works really well on text, which normally looks really blocky. It takes the text and makes it look nice, like it was meant for the resolution where you're displaying it. It also works well on lines and character spaces and any dithered artwork with checkerboard patterns or a lot of gradients. It does a good job when there are a small number of colors.

Q: The smoothing engine or algorithm knows when to leave something out or hop in?

A: It does it on color equality by looking at a single pixel at a time and checks to see if its neighbors are exactly the same color as it is. And then, based on patterns of equality, which neighbors are which colors, it decides whether or not to round off that pixel. The only way it knows text from a dithered background pattern is that it gets the equality patterns it's looking for in the text and not in the background pattern.

Q: It's got to detect sharp changes in the colors of the neighbors.

A: Out of the four neighbors of the pixel, if two of them match each other and the other two don't match anything, it knows it's sitting at a corner.

Q: Round-off is a terminology thing; you take the average of the pixel values of the neighbors to create a new color value?

A: Actually, no. When I expand the image into the high-res screen, one PC pixel becomes 2 x 2 Mac pixels. If a pixel gets smooth, that means that one or two of its new pixels gets a different color so it doesn't create any new color values, it just borrows colors from the neighbors. If the top neighbor and the left neighbor match each other, then the top left of the new pixels gets the new neighbor color. It's a weird algorithm, but not only did it work and produce a Mac effect, it ran just fine on the original Mac LC without much speed problem.

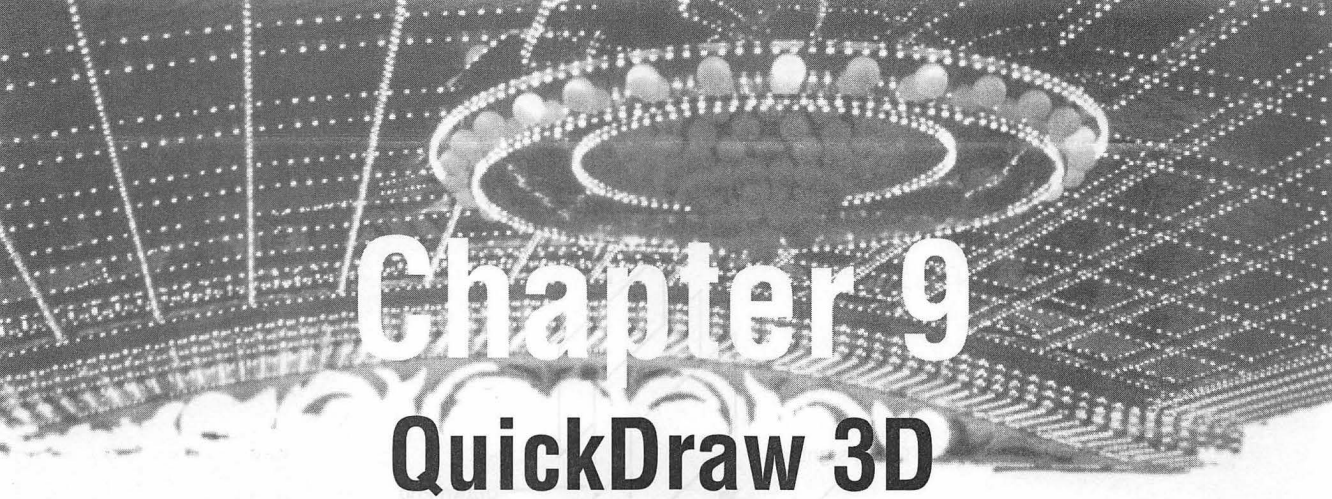
Q: That's a brilliant piece of work. What else have you done?

A: Smoothing was written because I was sitting in front of a game that I was porting and working on it at night at home and my wife came up behind me and looked at the text and said, "That text is ugly." She was right. The text was really pixelated, so that's when the smoothing algorithm was invented.

The other thing that was invented by necessity for one of these ports was a note mixer which would take any sample sound or instruments and mix them together so that we essentially had a pretty good MIDI player on the Mac, but it didn't depend on which sound driver you had. The thing that takes the most time in Note Player is not the actual mixing of the notes but the shifting of the pitches, so that if you have a note at middle C and you want to play it at A, shifting it into a different note value can be very time-consuming. There's a particular thing that's used to drop polygons very quickly that I adapted to shift note values in a very small number of instructions. It's not really a correct note value shift and so if the instruments aren't sampled well, then it can sound like a high school orchestra out of tune, but the nice thing is that we can control the sampling of the instruments.

One of the biggest issues that we're dealing with on the Mac right now is the joystick. As we get full-screen games on the Mac, joysticks are becoming more important because we get a lot of flight simulators and stuff where there weren't that many before. The joysticks that are available all come with software that lets you make the joysticks behave like a mouse or a keyboard, but none of them give you the option to make the joystick behave like a joystick. So as a game developer, there's no joystick standardization, which is something that PCs have always had. We're going through more stuff now, but basically you have to write special code for each joystick, which means that it won't work for new joysticks, or you need to have them all meet at a central point, like have them all control the mouse cursor. That means you're dependent on the joystick's mouse simulation software. It's become kind of a weird issue, but it might be solved if we can get some joystick standards in place.

continued



Chapter 9
QuickDraw 3D

QuickDraw 3D is Apple's latest addition to the QuickDraw family of graphics routines. This new suite of Toolbox calls provides even a novice game programmer with the ability to quickly and easily create highly detailed, three-dimensional worlds for Power Macs with only a minimal understanding of 3-D mathematics. Apple has provided a library of powerful 3-D routines numbering in the hundreds, and the *Inside Macintosh: QuickDraw 3D* volume is one of the largest volumes in the *Inside Macintosh* series. But do not let this new library's friendly interface fool you, for QuickDraw 3D is very powerful, fairly fast, and produces very high-quality rendered scenes.

QuickDraw 3D can render in only two different modes: wireframe and Z-Buffer. Although it is designed for extensibility, plug-in renderers will not be available for QuickDraw 3D until early 1996. Some of the renderers will be based on painter's and BSP (Binary Space Tree) algorithms. So for the present QuickDraw 3D does not support "depth sort" rendering, which is perhaps the most common and fastest rendering method used in 3-D video games. You may have heard the depth sort render also referred to as the "painter's algorithm." To render an image with this method, all of the polygons in a scene are sorted from farthest away to nearest, with all intersections resolved. Then, they are drawn in that order: back to front so that closer polygons are drawn over farther polygons (see figure 9.1). There are two benefits to a depth sort renderer. First, it's very fast if your scene or environment is fixed (more time is sometimes spent doing the 3-D math to rotate, scale, and translate polygons than is spent actually drawing them). Second, it uses far less memory than the Z-Buffer method for simple scenes with few intersections. Depth sort rendering is used in almost all polygon-based games and is the method of choice in most dedicated game machines such as the 3DO, Sega Saturn, and Sony PlayStation.

Although the depth sort method is the most commonly used method in video games, QuickDraw 3D does not currently use it. The next best thing that it provides is the Z-Buffer renderer. Z-Buffer rendering is much slower than depth sort rendering, and Z-Buffer rendering requires a lot of RAM. The basic principle behind this rendering algorithm is that for each pixel on the screen, there is a value which represents the distance from the camera to the currently drawn pixel. This distance value is called the Z value, hence the "Z-Buffer." This way, when a new polygon is being drawn, each pixel's Z value is compared with the Z value of the pixel already there. If the new pixel has a smaller Z value (i.e., it's closer to the camera) than the old pixel, then the new pixel is drawn over the old one.

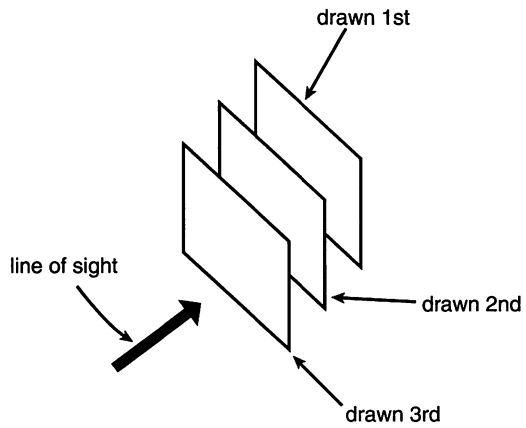


Figure 9.1 *Depth sorting.*

The reward for the increased complexity and decreased performance of the Z-Buffer algorithm is a much higher-quality rendering, because this algorithm allows for polygons to cleanly intersect each other (something that is almost impossible to do with a depth sort). Figure 9.2 shows the classic example of an impossible situation for a depth sort renderer. In this figure, every polygon overlaps another polygon; therefore, no polygon is in front or in back. Since a depth sort renderer relies entirely on accurate back-to-front sorting of polygons, it will fail in this case. A Z-Buffer renderer, however, does not care about the depth sorting of polygons, thus it can render this case without any problem.

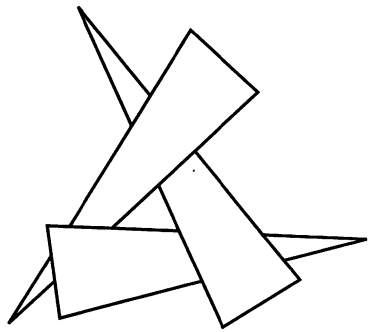


Figure 9.2 *The classic "impossible case" scenario for depth sort rendering.*

The QuickDraw 3D designers have done an excellent job in optimizing their Z-Buffer renderer, so much so that obtaining the frame rates necessary in certain kinds of video games is easily within reach. You probably won't be doing any full-screen flight simulators with QuickDraw 3D, but a lot of possibilities still exist. With QuickDraw 3D and a good knowledge of fighting games, you could easily create a 3-D polygon fighting game such as Sega's Virtua Fighter. Puzzle games that don't require high frame rates could be done

incredibly well. Even adventure games could be written to make excellent use of QuickDraw 3D to generate camera fly-throughs of the enemy village, or perhaps your character's eye view of the inside of a dragon's cave.

It should also be pointed out here that QuickDraw 3D is not just a software library. There is also an accelerator card for the PCI-based Power Macintoshes that can boost performance up to 12 times depending on the implementation of QuickDraw 3D in your game! Even though you would be limiting your game's market, you could write a game specifically for use with the accelerator card, in which case you probably could do a full-screen flight simulator. As a matter of fact, you could probably do the best-looking flight simulator the Mac has ever known!

Aside from QuickDraw 3D's quick rendering speed and quality, it touts many other great features:

- *Built-in Geometries*—Support for boxes, cones, cylinders, disks, ellipses, ellipsoids, general polygons, lines, polylines, meshes, NURB Curves, NURB Patches, points, tori, triangles, and trigrids. However, not all of these are supported in QuickDraw 3D 1.0.
- *Hierarchical Attributes*—Attributes may be assigned to groups of objects, individual objects, faces of objects, and vertices of faces. Attributes include color, ambient reflection values, specular colors, and so on.
- *Multiple Drawing Styles*—The ability to adjust the quality of the rendering. This includes techniques such as polygon backface removal, vertex color interpolation, and tessellation of spline-based geometries.
- *Powerful Lighting Features*—The four types of lights include ambient lights, directional lights, point lights, and spot lights. Spot lights can easily be aimed and focused. Light attenuation ("fall-off" values) can be individually set for each light.
- *Object Picking*—The ability to provide easy user interface with the mouse. Clicked objects and their associated data can easily be found from mouse and window information.
- *3D Math Utilities*—A large set of 3-D mathematical utilities for performing simple tasks such as setting vectors and points, as well as the more involved tasks such as calculating distances, dot products, and matrix manipulations.

Getting Started

The most difficult part of QuickDraw 3D is getting your first application up and running. Unlike plain old QuickDraw, which requires nothing more than a `SetPort()` and a `LineTo()` call to see results, QuickDraw 3D requires a *lot* of setup code. Even to display a line on the screen, you must perform many initialization tasks including setting the camera, creating lights, initializing the draw environment, and so on. But before we get started, it's important to have a basic understanding of the QuickDraw 3D Object Hierarchy and a rough idea of how attributes work.

Objects

Almost every element in QuickDraw 3D is an "object." Cameras are objects, geometries are objects, even attributes, shaders, and styles are objects. All of these objects are linked together hierarchically (see figure 9.3). The highest-level object is the group object. A group object (or simply a "group") is an object that contains references to more objects, even other group objects. It is a convenient way to collect all of the objects in a scene or part of a scene into one easily manageable entity. If you have 50 missiles flying around your scene, it is much easier to attach all of them to one group called `gMyMissleGroupObject` and draw that single object than it would be to always draw the 50 missiles one at a time. It's also more efficient to pass groups of objects to QuickDraw 3D than to pass lots of individual objects one at a time.

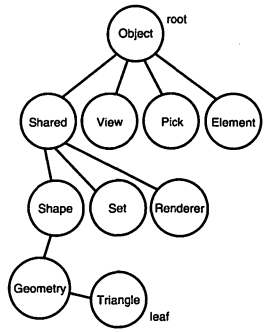


Figure 9.3 *The generic QuickDraw 3D hierarchy.*

Attributes

An attribute object defines how a geometric object should be drawn in a scene. Attributes include such things as what color the object is, how shiny the object is, and how to apply a texture map to the object. Attributes are always assigned to geometric objects or elements of a geometric object. These elements are usually polygon faces and vertices.

Attributes function by a system known as "inheritance." Inheritance means that attributes of higher-level elements are passed down to lower-level elements of a geometry unless those lower-level elements have overriding attributes of their own. For example, the color attribute of a vertex will override the color attribute of a face, which in turn will override the color attribute of the geometry. If you assign a mesh object a color attribute of green, the geometry will appear blue if the individual faces of the mesh were assigned a color attribute of blue.

QuickDraw 3D is immense—this entire book could be dedicated to a discussion of the intricacies of QuickDraw 3D's hierarchy and attribute rules, but since that would bore us all to tears, detailed discussions about attributes, objects, and hierarchies will be left up to the massive *Inside Macintosh: QuickDraw 3D* volume. Instead, we're going to delve right into the code to get us up and running.

Initializing QuickDraw 3D

The first thing we must do to get up and running is to initialize QuickDraw 3D. In your application's startup routine, you must insert the following code to activate QuickDraw 3D.

```
    /* Initialize QD 3D */

TQ3Status myStatus;

  myStatus = Q3Initialize();
  if ( myStatus == kQ3Failure )
    DoFatalAlert("\pQ3Initialize returned failure!");
```

QuickDraw 3D Toolbox routines often return an error status of either `kQ3Failure` or `kQ3Success`. Unlike many other Macintosh tool calls, the Boolean values here are reversed. `kQ3Success` equals 1, and `kQ3Failure` equals 0; therefore, be careful not to write code like you may have in the past, in which you perform simple error checks like this:

```
OSErr   error;

  error = ToolboxCall();
  if (error)
    HandleQ3ror();
```

Since the returned value for an error is no longer 1, this will give you the opposite effect that you want. It is good practice in QuickDraw 3D to always check error conditions by explicitly comparing against `kQ3Success` or `kQ3Failure`.

Creating a Draw Environment

Once we have initialized QuickDraw 3D, we need to create a draw environment before we can do anything. The draw environment defines all of the basic information that QuickDraw 3D needs to know in order to set up a scene. We do this by executing the following ten steps.

1. Create a window in which the scene will be drawn.
2. Create a new view object.
3. Attach a draw context to the view.
4. Attach a renderer to the view.
5. Attach a camera to the view.
6. Attach a light group to the view.
7. Create an interpolation style object.

8. Create a backfacing style object.
9. Create a fill style object.
10. Create a shader object.

At first glance, this may seem like a lot to do—and it is. Luckily for you, however, I have provided all of the code necessary to perform these ten tasks. The code necessary to set up our drawing environment teaches many of QuickDraw 3D's most commonly used principles, so it is a good idea to thoroughly read the following examples and to get a good feel for how and why things work. If it makes sense to you, then you should have no problem with anything else in the QuickDraw 3D world. The first step is trivial: If you're ready to tackle QuickDraw 3D, I presume you are familiar with creating Mac windows. The technique is listed in the code and I won't repeat it here.

Creating the View Object

The following routines initialize a view object and perform steps 2 through 6 as described previously.

```
/****************** CREATE MY VIEW ************************/
//
// This creates our view object.
//
// INPUT  : none
// OUTPUT : none
//
// NOTE: gMyViewObject is a global variable of type TQ3ViewObject
//

void CreateMyView(void)
{
TQ3Status         myErr;
TQ3DrawContextObject   myDrawContext;
TQ3RendererObject   myRenderer;
TQ3GroupObject      myLights;


        /* CREATE NEW VIEW OBJECT */
  gMyViewObject = Q3View_New();
  if (gMyViewObject == nil)
    DoFatalAlert("\pQ3View_New failed!");

      /* CREATE A DRAW CONTEXT OBJECT */
  myDrawContext = CreateMyDrawContex(gMyWindow);
  if (myDrawContext == nil)
    DoFatalAlert("\pMyNewDrawContext Failed!");

      /* ASSIGN DRAW CONTEXT TO THE VIEW */
  myErr= Q3View_SetDrawContext(gMyViewObject, myDrawContext);
  if ( myErr== kQ3Failure )
```

```
    DoFatalAlert("\pSetDrawContext Failed!");

        /* CREATE RENDERER OBJECT */
  myRenderer = Q3Renderer_NewFromType(kQ3RendererTypeInteractive);
  if (myRenderer == nil)
    DoFatalAlert("\pRenderer_NewFromType Failed!");

      /* ASSIGN RENDERER TO THE VIEW */
  myErr= Q3View_SetRenderer(gMyViewObject, myRenderer);
  if ( myErr== kQ3Failure )
    DoFatalAlert("\pSetRenderer Failed!");

      /* CREATE A CAMERA OBJECT */
  gMyCameraObject = CreateMyCamera(myDrawContext);
  if (gMyCameraObject == nil)
    DoFatalAlert("\pCreateMyCamera Failed!");

      /* ASSIGN CAMERA TO THE VIEW */
  myErr= Q3View_SetCamera(gMyViewObject, gMyCameraObject);
  if ( myErr== kQ3Failure )
    DoFatalAlert("\pSetCamera Failed!");

      /* CREATE A LIGHT GROUP OBJECT */
  myLights = CreateMyLightGroup();
  if ( myLights == nil )
    DoFatalAlert("\pCreateMyLightGroup Failed!");

    /* ASSIGN LIGHT GROUP TO THE VIEW */
  myErr= Q3View_SetLightGroup(gMyViewObject, myLights);
  if ( myErr== kQ3Failure )
    DoFatalAlert("\pSetLightGroup Failed!");
}
```

The routine `CreateMyView()` initializes a QuickDraw 3D view object in a very straightforward manner. It creates the new view object simply by calling the `Q3View_New()` function, and then it proceeds to attach the required objects to the view. The first object attached is the draw context object (see `CreateMyDrawContex()` coming up). A draw context defines where a scene will be rendered, such as GWorld, screen, and so on.

The next object to be assigned to our view is a renderer object. The routine `Q3Renderer_NewFromType()` takes a predefined renderer type as input. The setting `kQ3RendererTypeInteractive` tells QuickDraw 3D to use the Z-Buffer renderer and to automatically use hardware acceleration if available.

Note

There is an additional call that can be made immediately after creating a renderer that can improve performance with hardware accelerators.

continued

```
Q3InteractiveRenderer_SetDoubleBufferBypass(renderer,kQ3True);
```

This line of code tells QuickDraw 3D not to double buffer if the acceleration hardware is capable of displaying fast enough without it. This can theoretically improve performance up to 10 percent under the right conditions. Even on fast renderers, however, you still may see some flickering due to the elimination of the double buffer.

Now that we set up where our scene will appear, we proceed to define our scene's camera via `CreateMyCamera()`. As you will see later, this routine creates a camera object that defines where our "eye" is located in 3-D space and what it is looking at.

The final object that needs to be attached to our view is the light group object. The light group object is a group object that contains all of the lights in our scene. Any scene must have at least one light if you want to be able to see anything. A scene rendered with no lights will be black.

Creating the Draw Context

Let's take a closer look at some of the routines that were called in the previous code. The first routine that we called was `CreateMyDrawContex()`. Here is the code for `CreateMyDrawContex()`:

```
/*************** CREATE MY DRAW CONTEXT ********************/
//
// Creates a DRAW CONTEXT which will be attached to the fly-through VIEW object.
// The draw context defines how a frame is drawn to a window or off-screen buffer.
//
// INPUT  : pointer to the window we will be drawing in
// OUTPUT : the new draw object
//

TQ3DrawContextObject CreateMyDrawContex(WindowPtr theWindow)
{
TQ3DrawContextObject   myDrawContext;
TQ3DrawContextData     myDrawContextData;
TQ3MacDrawContextData  myMacDrawContextData;
TQ3ColorARGB        clearColor = {1,0,0,0.25};                    // dark blue

      /* FILL IN DRAW CONTEXT DATA */
  myDrawContextData.clearImageMethod = kQ3ClearMethodWithColor;    // how to clear
  myDrawContextData.clearImageColor = clearColor; // color to clear with
  myDrawContextData.pane.min.x = (theWindow->portRect).left;       // set bounds
  myDrawContextData.pane.max.x = (theWindow->portRect).right;
  myDrawContextData.pane.min.y = (theWindow->portRect).top;
  myDrawContextData.pane.max.y = (theWindow->portRect).bottom;
```

```
    myDrawContextData.paneState = kQ3True;                    // use bounds = yes
    myDrawContextData.maskState = kQ3False;                   // no mask
    myDrawContextData.doubleBufferState = kQ3True;            // use ➥
double buffering

          /* SET MAC-SPECIFIC DATA */
    myMacDrawContextData.drawContextData = myDrawContextData;
    myMacDrawContextData.window = (CWindowPtr)theWindow;      // set window to ➥
draw into
    myMacDrawContextData.library = kQ3Mac2DLibraryNone;
    myMacDrawContextData.viewPort = nil;
    // (for QuickDraw GX only)
    myMacDrawContextData.grafPort = nil;

      /* CREATE MACINTOSH DRAW CONTEXT BASED ON THAT DATA */
    myDrawContext = Q3MacDrawContext_New(&myMacDrawContextData);
    if (myDrawContext == nil)
      DoFatalAlert("\pQ3MacDrawContext_New failed");

    return (myDrawContext);
}
```

This routine creates and returns a draw context object. To create the object, we first set up a `TQ3DrawContextData` structure and a `TQ3MacDrawContextData` structure. They will contain all of the information necessary to define our draw context.

The first two items initialized are the `clearImageMethod` field and `clearImageColor` field. The `clearImageColor` field defines the RGB color for our scene's background. In this example, we have set the clear color to dark blue. RGB colors in QuickDraw 3D are expressed as three floating point numbers between 0.0 and 1.0, where 0.0 is the darkest value and 1.0 is the brightest value for each color component red, green, and blue. Therefore, an RGB value of 1.0, 1.0, 1.0 is white, and the value 0.0, 0.0, 0.0 is black. Solid red would be 1.0, 0.0, 0.0. However, it is important to note that there are two types of color definitions: `TQ3ColorRGB` and `TQ3ColorARGB`. The `TQ3ColorARGB` structure is used in the code sample. The `A` stands for "Alpha." The alpha value of a `TQ3ColorARGB` structure determines the amount of transparency for special hardware effects. We won't be worrying about alpha values in our discussion, so we'll just set it to 1.0. The value 1.0 tells QuickDraw 3D that there is no transparency. A value of 0.0 would be maximum transparency.

The `pane`, which is the next field to be initialized, is analogous to the QuickDraw `Rect` structure because it defines the four edges of a rectangle. QuickDraw 3D uses the `pane` to determine where in the window we want it to render our scene. In the code, we are setting the `pane` to the size of the entire window by getting the window's `portRect` structure and copying it to the draw context's `pane` structure. Suppose we did not want to draw into the entire window—suppose we wanted to leave a 20-pixel-wide area on the right side of the window for a game's status bar. To do this, simply change the line:

```
myDrawContextData.pane.max.x = (theWindow->portRect).right;
```

to:

```
myDrawContextData.pane.max.x = (theWindow->portRect).right-20;
```

That will reduce the right side of the pane to leave a 20-pixel-wide gap.

There is also a `paneState` flag that is set. If this flag is set to `kQ3False`, then the `pane` min/max settings will be ignored and the entire window will be used by default. Since in our example we are using the whole window, we could have simplified this code by just setting the `paneState` flag to `kQ3False` and not even bothered with the `pane` bounds.

The next item in the draw context's data structure is the `doubleBufferState` flag. Double buffering is the process of drawing to one draw buffer while displaying another. We have set the `doubleBufferState` flag to `kQ3True` and you will usually want to have it set that way when you are rendering directly to a window. Turning double buffering off will result in a tremendous amount of tearing on the screen as you animate the scene. Power Macs equipped with hardware acceleration for QuickDraw 3D may be able to live without double buffering at the cost of minimal screen tearing.

Our draw context data structure now needs to be attached to a `TQ3MacDrawContextData` structure. The `TQ3MacDrawContextData` structure defines Macintosh-specific information about the draw context. Because QuickDraw 3D will be available for Microsoft Windows as well as the Macintosh, there are separate data structures that define the draw context for each machine. Luckily, we don't care about Windows, so we'll pretend that QuickDraw 3D exists only for the Macintosh (the way it should be, eh?). Should you want to render to an off-screen buffer and not a window, then you need to use a `TQ3PixmapDrawContextData` structure instead. You cannot render to an off-screen buffer such as a GWorld by using a `TQ3MacDrawContextData` structure.

The `TQ3MacDrawContextData` structure's `window` parameter is set to the `WindowPtr` that we passed into `CreateMyDrawContex()`. This will tell QuickDraw 3D which window to render into. Make sure that a window exists before doing this. Passing a bogus window will obviously have, oh, shall we say, "adverse" side effects.

The next parameter set is the `library` parameter. For our purposes, this should always be set to `kQ3Mac2DLibrary_None`. Violà! All the data we need to define a draw context has been set. Once we call `Q3MacDrawContext_New()`, we have our new draw context object which gets passed back to the calling routine.

The only part of this routine that you will probably ever change is the `clearImageColor` setting, because you will want to experiment with background colors for your scene. All the other settings will most likely remain unchanged.

Creating the Camera

The next two code listings bring us into the realm of more tangible things: cameras and lights. This first listing shows how our camera object is created and initialized.

```
/****************** CREATE MY CAMERA ********************/
//
// Creates a camera which will be attached to the VIEW object.
//
// INPUT  : the draw context object
// OUTPUT : the camera object
//

TQ3CameraObject CreateMyCamera(TQ3DrawContextObject drawContext)
{
TQ3CameraObject          myCamera;
TQ3CameraData            myCameraData;
TQ3ViewAngleAspectCameraData  myViewAngleCameraData;
TQ3Point3D               cameraFrom = { 4.0, 5.0, 6.0 };
TQ3Point3D               cameraTo = { 0.0, 0.0, 0.0 };
TQ3Vector3D              cameraUp = { 0.0, 1.0, 0.0 };
TQ3Area                  pane;


        /* FILL IN CAMERA DATA */

  myCameraData.placement.cameraLocation = cameraFrom;    // set camera coords
  myCameraData.placement.pointOfInterest = cameraTo;     // set "looking at" coords
  myCameraData.placement.upVector = cameraUp;            // set a vector that's "up"
  myCameraData.range.hither = 0.1;                       // set frontmost Z dist
  myCameraData.range.yon = 100.0;                        // set farthest Z dist
  myCameraData.viewPort.origin.x = -1.0;                 // set view origins
  myCameraData.viewPort.origin.y = 1.0;
  myCameraData.viewPort.width = 2.0;
  myCameraData.viewPort.height = 2.0;

  myViewAngleCameraData.cameraData = myCameraData;
  myViewAngleCameraData.fov = 0.8;                       // set field of view angle
  Q3DrawContext_GetPane(drawContext,&pane);              // get window pane info
  myViewAngleCameraData.aspectRatioXToY =                // set aspect ratio of window
      (pane.max.x-pane.min.x)/(pane.max.y-pane.min.y);


    /* CREATE THE NEW CAMERA OBJECT BASED ON THAT DATA */

  myCamera = Q3ViewAngleAspectCamera_New(&myViewAngleCameraData);

  return ( myCamera );
}
```

Creating the camera object is very similar to creating the draw context object described previously. We start by filling in a `TQ3CameraData` structure that contains all of the information needed to define our camera. The first parameter we set is `cameraLocation`, which, as its name implies, sets the coordinates in the 3-D universe where we want to place our camera. The `pointOfInterest` field contains the coordinates in 3-D space where we want our camera to look. In our example code, we are looking at the center of our 3-D universe which is at (0, 0, 0). We have placed the camera at (4, 5, 6). This way, the camera should be looking at the center of the universe from a bird's-eye perspective.

As if that were not enough data to define our camera's orientation, we also set the `cameraUp` field. This field tells QuickDraw 3D which direction the camera should consider to be "up" (see figure 9.4). In other words, it sets the amount of "roll" to apply to the camera. Unless you are writing a flight simulator, you will probably set `cameraUp` to the "real" up: x = 0, y = 1, z = 0. In a flight simulator, however, your aircraft needs to roll, in which case the up vector will change. When the aircraft is completely upside down, the up vector would be x = 0, y = -1, z = 0.

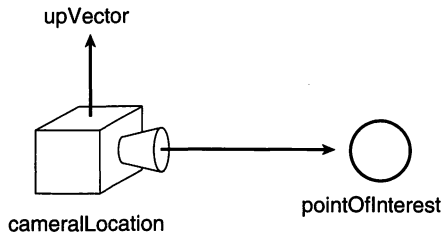


Figure 9.4 *Camera orientation data.*

Note that the `cameraLocation` and `pointOfInterest` fields are of the type `TQ3Point3D`, but the `cameraUp` field is of type `TQ3Vector3D`. Both points and vectors are just sets of x, y, and z floating point numbers. A point is obviously just a coordinate in 3-D space defined by the values x, y, and z, but a vector represents a direction and magnitude away from a point. It often helps to think of vectors as points in space relative to some other point. For example, say we have a base point at coordinates (0, 0, 0). A vector straight "up" from that point could be (0, 1, 0)—0 units away on the x-axis, 1 unit away on the y-axis, and 0 units away on the z-axis. As you may have noticed (0, 1, 0) could also represent a point in space that is physically located "up" from the base point. If our base point were located at (100, -30, 16), then the "up" vector would still be (0, 1, 0) because a point located "up" from the base is still 1 unit away on the y-axis.

You may choose to think of vectors in either of these manners. Think of them simply as directions and magnitudes away from a point in space, or think of them as points in space relative to some other point in space. Either way, they represent the same thing.

A related item of interest is what are called "normalized" vectors. The values of a vector can be any floating point number, but normalized vectors are vectors that are scaled so that their magnitude is 1 (to normalize a vector, divide each of its components by the square root of the sum of the squares of each component). For example, a vector might be (50, 100, 25). If we were to normalize this vector, the result would be (0.6, .9, 0.2). Or, if we had a vector of (0.01, 0, -0.001), its normalized value would be (1, 0, -0.1). Many QuickDraw 3D routines return normalized vectors or require normalized vectors as input. It's good practice to always work with normalized vectors; this is why our camera's up vector is (0, -1, 0). We could have also used the vector (0, -0.0001, 0) or (0, -2345, 0), since they are both geometrically identical to (0, -1, 0), but (0, -1, 0) is much better to work with since it is normalized.

The next two fields in the `TQ3CameraData` structure determine the range of sight that our camera has. The `yon` field defines how far the camera can see. Any pixel farther than "`yon`" units away from the camera will not be drawn. Similarly, the `hither` field defines how near the camera can see. Any pixel closer to the camera than "`hither`" units will not be drawn. Generally, you want objects to be able to come very close to the camera, so keep the `hither` field very small.

Caution

Do not set the `hither` field to 0. It must always be a number greater than 0.

Notice the `viewPort` fields of the `TQ3CameraData` structure. These set the origins of the `viewPort` and should be set to the numbers you see in the code listing. The meanings of these values go beyond the scope of what we can cover in this chapter. For more information, refer to *Inside Macintosh: QuickDraw 3D*.

We are now done filling in the `TQ3CameraData` structure, but there is still one more structure required in order to make the camera object. We need to fill out a `TQ3ViewAngleAspectCameraData` structure. First, we copy the `TQ3CameraData` structure that we just created into the `cameraData` field of the new `TQ3ViewAngleAspectCameraData` structure. Next, we set the lens size we want to use on our camera by entering it into the `fov` field, which represents the "field of view" angle. It is the view angle of the lens in radians. Larger numbers create a more wide-angle view like a fish-eye lens, and smaller numbers create a more narrow-view angle like a telephoto lens (see figure 9.5).

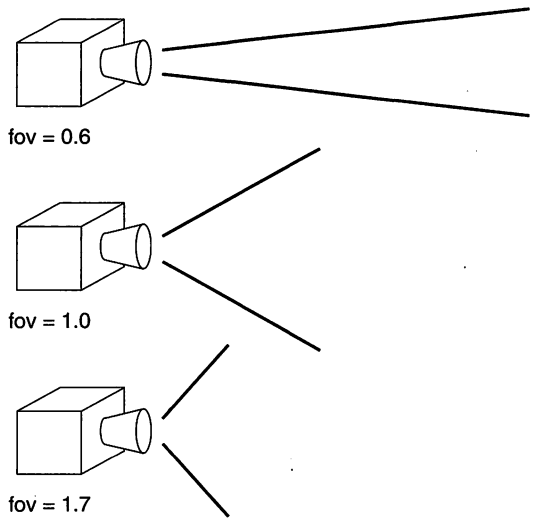


Figure 9.5 *Lens `fov` values.*

The final step in setting up the camera object is a little ambiguous. If the window you are rendering into is not perfectly square, QuickDraw 3D will squash or stretch the rendered image relative to the shape of the window. This can, however, be corrected. The `aspectRatioXToY` field of the `TQ3ViewAngleAspectCameraData` structure must be calculated and set. This value can compensate for nonproportionally sized windows. The following lines show how this is done:

```
Q3DrawContext_GetPane(drawContext,&pane); // get window pane info
myViewAngleCameraData.aspectRatioXToY =    // set aspect ratio for window
     (pane.max.x-pane.min.x)/(pane.max.y-pane.min.y);
```

That's all we need to do for the setup; all that remains is to pass the `TQ3ViewAngleAspectCameraData` structure to `Q3ViewAngleAspectCamera_New()`. That will create the camera object.

Adding a Light Group

We now have a view environment and a camera with which to do the viewing. The only thing remaining to prepare for our view object is the light group object. The next listing shows how we create a light group that contains two of the four different types of lights. We are going to discuss only the ambient and directional lights because those are the only two types of lights that you will be likely to use in a game. Spot lights and point lights are somewhat more computationally complex and therefore take more time to render. In other words: "Don't use 'em, it'll slow your game down!"

The ambient light determines the amount of nondirectional, general light that exists in the scene. Every polygon in a scene will be equally lit by an ambient light source. Computationally, it is the simplest and most efficient of all the types of lights, but because it shades all polygons in an object equally, it creates a very flat-looking scene if not used in conjunction with any of the other light sources (see figure 9.6).

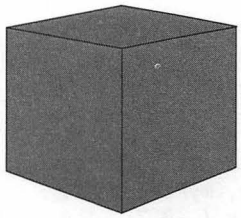


Figure 9.6 *Ambient lights shade all polygons equally.*

Directional lights, on the other hand, help in adding depth to a rendered scene. Like ambient light, directional lights have no location in space, but they do aim in a particular direction. Think of this kind of light as having parallel rays that emanate from an infinitely far away point and cast light for an infinite distance. These types of lights cause faces of an object to be shaded differently depending on the angle of the face to the light (see figure 9.7). Sunlight behaves like this.

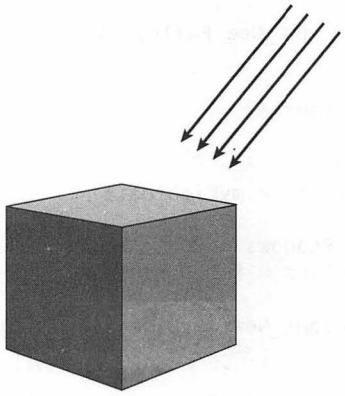


Figure 9.7 *Directional lighting shades the faces of a box differently.*

Now let's see how to create a light group object that contains an ambient and directional light.

```
/*************** CREATE MY LIGHT GROUP ******************/
//
// Creates a light group which will be attached to the view object.
//
// INPUT  : none
// OUTPUT : the new light group object
//

TQ3GroupObject CreateMyLightGroup(void)
{
TQ3GroupPosition    myGroupPosition;
TQ3GroupObject      myLightGroup;
TQ3LightData        myLightData;
TQ3DirectionalLightData  myDirectionalLightData;
TQ3LightObject      myAmbientLight, myFillLight;
TQ3Vector3D         fillDirection = { -0.7, -1.0, 0 };
TQ3ColorRGB         whiteLight = { 1.0, 1.0, 1.0 };

  myLightData.isOn = kQ3True;                                    // light is ON
  myLightData.color = whiteLight;                                // set color of ➥
light

      /* CREATE AMBIENT LIGHT */

  myLightData.brightness = 0.4;                                  // set brightness ➥
value
  myAmbientLight = Q3AmbientLight_New(&myLightData);             // make new light
  if (myAmbientLight == nil)
    DoFatalAlert("\pQ3AmbientLight_New Failed!");


      /* CREATE DIRECTIONAL LIGHT */

  myLightData.brightness = 0.8;                                  // set brightness
  myDirectionalLightData.lightData = myLightData;                // refer to general
light info
  myDirectionalLightData.castsShadows = kQ3False;                // no shadows
  myDirectionalLightData.direction = fillDirection;              // set fill direc- ➥
tion vector
  myFillLight = Q3DirectionalLight_New(&myDirectionalLightData); // make new light ➥
object
  if (myFillLight == nil)
    DoFatalAlert("\pQ3DirectionalLight_New Failed!");


      /* CREATE LIGHT GROUP */

  myLightGroup = Q3LightGroup_New();
  if ( myLightGroup == nil )
    DoFatalAlert("\pQ3LightGroup_New Failed!");
```

```
    /* ADD AMBIENT LIGHT OBJECT TO THE GROUP */

  myGroupPosition = Q3Group_AddObject(myLightGroup, myAmbientLight);
  if ( myGroupPosition == 0 )
    DoFatalAlert("\p Q3Group_AddObject Failed!");


    /* ADD DIRECTIONAL LIGHT OBJECT TO THE GROUP */

  myGroupPosition = Q3Group_AddObject(myLightGroup, myFillLight);
  if ( myGroupPosition == 0 )
    DoFatalAlert("\p Q3Group_AddObject Failed!");


    /* DISPOSE OF OUR REFERENCES TO THE LIGHT OBJECTS */

  Q3Object_Dispose(myFillLight);
  Q3Object_Dispose(myAmbientLight);

  return ( myLightGroup );
}
```

As with the previous examples, you'll notice that once again we start by filling in a data structure for each object we are about to create. First, we set the `isOn` flag of the `TQ3LightData` structure to `kQ3True`, since we obviously want the lights we are creating to be on, not off; then we set the color that we want our lights to be. We are setting all of them to white (RGB = 1.0, 1.0, 1.0).

Next, we set the specific data that will define our ambient light. There is not a lot to set for an ambient light, just the `brightness` value. Traditionally, the `brightness` value of a light should be between 0.0 and 1.0, but if you want a light to be a little "hot," you can set the value as high as you want. However, I would not recommend setting it over 2.0, because it will not look particularly good at that brightness. Once that's done, we call `Q3AmbientLight_New()` to create a new ambient light object based on the data we passed to it.

To create our second light source, the directional light, we need to fill out another one of those fun little data structures. This time it's the `TQ3DirectionalLightData` structure. After setting a new `brightness` value, we attach the `TQ3LightData` structure to our new `TQ3DirectionalLightData` structure, then we set the directional light's specific fields. The first field is the `castsShadows` flag. We definitely do not want to use shadows in a game. That would bring our performance to a complete halt. Next, we set the most important parameter of a directional light, the `direction`. The `direction` of the light is a vector that aims in the direction the light would be traveling from its infinitely far away source. For example, suppose we wanted our light to be shining from some positive x and y coordinate toward 0. A possible vector would be (-1, -1, 0) (see figure 9.8). Finally, we create the new directional light object by calling `Q3DirectionalLight_New()`.

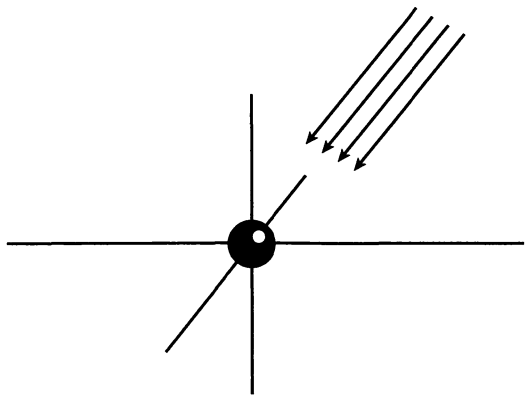


Figure 9.8 *Object being illuminated by a directional light with a* `direction` *vector of (-1, -1, 0).*

We have now created two lights. At this point we get to introduce an object that we have heard about but not seen until now: the group object. QuickDraw 3D likes us to put all of the light sources in a scene into a light group. A light group is just a regular `TQ3GroupObject`, which we will see a lot more of later in this chapter. Adding objects into groups is very simple. All it takes is a single call to `Q3Group_AddObject()`, which takes the destination group and source object as input. Anything can be put into a group because everything in QuickDraw 3D is an object. A group may contain transform objects, geometric objects, camera objects, shader objects, and so on. Any object can be added to any group.

`Q3Group_AddObject()` returns a `TQ3GroupPosition` value, which is used if you ever want to remove that object from the group or get information about that object. In this routine, we will never want to touch our lights again, so we just toss out the value—but if, for example, you ever wanted to change the `brightness` value of one of the lights in the group, you would want to keep the `TQ3GroupPosition` so that you could easily get access to the light again and change its `brightness` value.

Style and Shader Objects

Remember when I told you that getting up and running in QuickDraw 3D required a lot of initial preparation? Well, let's just say that there's no such thing as a quick "hello world" program in QuickDraw 3D. All we've done is create a draw context object, one of the three basic objects necessary to get up and running. Luckily, the next two steps, creating a style object and creating a shader object, are much simpler than creating the draw context object. Study this next code example.

```
/**************** CREATE MY STYLES AND SHADER ****************/
//
// Creates style objects which define how the scene is to be rendered.
// It also sets the shader object.
//
```

```
// INPUT     : none
// OUTPUT    : none
//

void CreateMyStylesAndShader(void)
{
            /* SET INTERPOLATION (FOR SHADING) */

  gMyStyleObject_Interpolation =                    // do vertex interpolation
        Q3InterpolationStyle_New(kQ3InterpolationStyleVertex);
  if (gMyStyleObject_Interpolation == nil )
    DoFatalAlert("\p Q3InterpolationStyle_New Failed!");


            /* SET BACKFACING */

  gMyStyleObject_Backfacing =                       // keeps backfaces
    Q3BackfacingStyle_New(kQ3BackfacingStyleBoth);
  if (gMyStyleObject_Backfacing == nil )
    DoFatalAlert("\p Q3BackfacingStyle_New Failed!");


          /* SET POLYGON FILL STYLE */

  gMyStyleObject_Fillstyle =                        // we want the polys filled
    Q3FillStyle_New(kQ3FillStyleFilled);
  if (gMyStyleObject_Fillstyle == nil )
    DoFatalAlert("\p Q3FillStyle_New Failed!");


          /* SET SHADER TO LAMBERT */

  gMyShaderObject_IlluminationShader = Q3LambertIllumination_New();
  if (gMyShaderObject_IlluminationShader == nil )
    DoFatalAlert("\p Q3LambertIllumination_New Failed!");
}
```

Interpolation Styles

The above three Toolbox calls are the same as we've seen a hundred times already, but there are some details to be aware of. First of all, we have set the Interpolation Style to `kQ3InterpolationStyleVertex`. This tells QuickDraw 3D that when drawing a polygon, we want it to calculate the colors for each pixel based on the color calculations of its vertices. The two other possible settings are `kQ3InterpolationStyleFace` and `kQ3InterpolationStylePixel`. Face interpolation means that it just does one calculation

for the face of the polygon. This method is not particularly faster than the `kQ3InterpolationStyleVertex` method and because it only calculates one color value for an entire polygon, it results in very flat and dull-looking scenes.

The `kQ3InterpolationStyleVertex` method calculates one color value for each vertex of a polygon; thus, the colors across its surface blend smoothly from vertex to vertex. With this method, one vertex of a triangle might be set to green, another red, and another blue. The rendered polygon will have a smoothly shaded surface that blends from red to green to blue. This type of interpolated shading is commonly referred to as Gouraud shading. The performance hit for this kind of shading is very minor, less than three percent, but the visual results are astounding and well worth it.

The final type of Interpolation Style, `kQ3InterpolationStylePixel`, performs a light calculation for every pixel on the surface of a polygon. It may result in nice images, but it is too slow to be practical in a game.

Backfacing Styles

Backfacing polygons are polygons that face away from the camera. In general, any polygon facing away from the camera would be on the back side of an object, and thus obscured from our view by the front of the object. In such a case, you might want to prevent the backfaces from being rendered because they will never be seen. But unfortunately, this is not always the case, as you are about to see.

Which backfacing style you want to use depends largely on your application. Many applications will want to remove backfaces to optimize performance, while other applications need to keep backfaces because there may exist geometries whose polygons are visible even when they face away from the camera. The tail fin of an airplane might be just one large triangular polygon. We would not want backface removal activated for that object, because it would cause the tail fin to disappear when it faced away from the camera.

In the previous code, we set our backfacing style object to `kQ3BackfacingStyleBoth`, which will work best for our general examples. The two other possible settings for the backfacing style object are `kQ3BackfacingStyleFlipNormals` and `kQ3BackfacingStyleRemoveBackfacing`. As it implies, `kQ3BackfacingStyleBoth` causes both sides of a polygon to be drawn. In other words, no backface removal will occur. On the other hand, `kQ3BackfacingStyleRemove` tells QuickDraw 3D to check for backfaces and remove them if found. `kQ3BackfacingStyleFlip` will invert all a polygon's surface normals (vectors used to determine backfacing). In QuickDraw 3D, the normal to a polygon will face the camera if the vertices of the polygon are ordered *counterclockwise* (see figure 9.9). If your mesh object has polygons with vertices ordered clockwise, you will have to use `kQ3BackfacingStyleFlip` for the light shading to be correct—otherwise the lighting of a polygon will be inverted. This is a very important concept to understand because all backface removal and light shading calculations rely on the integrity of a consistent vertex ordering. If backface removal is turned on and your vertices are ordered clockwise, then the visible parts of objects will become invisible and the hidden parts will be drawn instead.

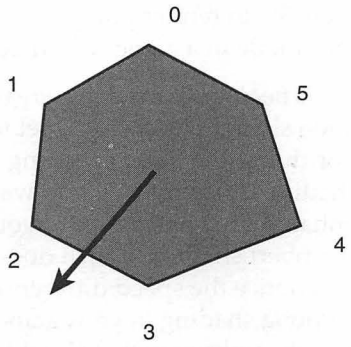


Figure 9.9 *Correctly ordered vertices with the surface normal facing the camera.*

It is important to note, however, that backface removal is not always a good optimization. If most of the polygons in your scene are not backfaces, then it can actually take longer to check all the polygons to see if they are backfaces than to just draw everything anyway. Say you have a sphere on the screen. In that case, about 50 percent of the sphere's polygons will be hidden backfaces at any given time; thus, removing the backfaces would probably be a good optimization. On the other hand, suppose you were rendering a mesh object which was a flat surface with a tiny bump in the middle. The only polygons that might be backfaces are the ones in back of the bump. In this case, perhaps only eight percent of the polygons could be eliminated by backface removal. Depending on the circumstances, it might take longer to check all the polygons for backface removal than it would take to just draw those eight percent anyway (see figure 9.9).

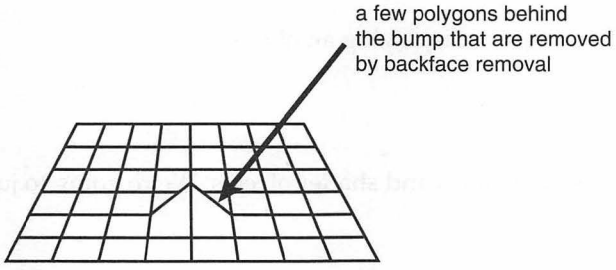


Figure 9.10 *Example of poor backface optimization.*

Fill Styles

Next, we need to set the polygon fill style object. This determines how the polygons are to be filled (or if they are filled at all). Our setting, `kQ3FillStyleFilled`, tells QuickDraw 3D to draw the polygons as filled, solid shapes. The setting of `kQ3FillStyleEdges` causes only the edges of the polygons to be rendered. This is essentially a wireframe draw.

`kQ3FillStylePoints` tells QuickDraw 3D to render polygons by drawing only points for the vertices. About the only use for this mode in a game would be to render a star field.

Now that all of the style objects have been initialized, we create a shader object. For general rendering, we create an Illumination shader object that is set to use the Lambert shading algorithm. The other shading algorithm is the Phong shading algorithm. Lambert and Phong are identical except that Phong shading is necessary if you want to have specular highlights. Specular highlights can greatly enhance the appearance of your 3-D scene, but they require slightly more computation than Lambert shading, which does not support specular highlights. In general you may not even notice the speed difference between Lambert and Phong shading, so you may wish to use Phong shading in your game to make it look it's best. To be safe, however, use Lambert shading to make sure that QuickDraw 3D is rendering as fast as it can. Which shader you use is up to you.

Well, that's it. QuickDraw 3D is initialized, the motor is running, and there's a road to drive on. The next step is simple: Let's finally get something on the screen!

Displaying a 3-D Object

It is traditional for a programmer to try drawing lines to the screen as a first attempt at programming graphics with an unfamiliar system. QuickDraw 3D does support lines and polylines (an unbroken series of successive line segments), but that would be incredibly dull considering the potential we have to do something much cooler for our first experiment. Instead, we are going to jump right into the world of 3-D shapes by drawing a 3-D cube. After our battle with the setup code, I think you will find everything that follows much more intuitive and rewarding.

There are really only two steps to displaying an object:

1. Create the object.
2. Draw the object.

That's it! No more convoluted views and shader objects. We're going to just make a box and draw it.

Making a Box

Take a look at the following code, which shows how to make a simple box. QuickDraw 3D refers to rectangular solids as "boxes" for some reason.

```
/****************** CREATE MY BOX OBJECT ******************/
//
// Creates and returns a box geometric object.
//
// INPUT     : none
```

```
// OUTPUT : box object
//

void CreateMyBoxObject(void)
{
TQ3ColorRGB       color = {1.0,1.0,1.0};      // make it white
TQ3AttributeSet   attribs;
float             ambient = 1.0;
TQ3Point3D        origin = {0,0,0};
TQ3Vector3D       orientation = {0,2,0};
TQ3Vector3D       majorAxis = {0,0,2};
TQ3Vector3D       minorAxis = {2,0,0};
TQ3BoxData        boxData;
TQ3Status         myStatus;


      /* CREATE ATTRIBUTES FOR BOX */

  attribs = Q3AttributeSet_New();                    // make new attrib object
  if (attribs == nil)
    DoFatalAlert("\pUnable to create new attribute Set!");

                                                     // add color attribute
  myStatus = Q3AttributeSet_Add(attribs, kQ3AttributeTypeDiffuseColor, &color);
  if ( myStatus == kQ3Failure )
    DoFatalAlert("\p Q3AttributeSet_Add Failed!");

                                                     // add ambient attribute
  myStatus = Q3AttributeSet_Add(attribs, kQ3AttributeTypeAmbientCoefficient, &ambient);
  if ( myStatus == kQ3Failure )
    DoFatalAlert("\p Q3AttributeSet_Add Failed!");


      /* SET BOX INITIALIZE DATA */

  boxData.origin = origin;                           // set box origin coords
  boxData.orientation = orientation;                 // set orientation vector
  boxData.majorAxis = majorAxis;                     // set major axis vector
  boxData.minorAxis = minorAxis;                     // set minor axis vector
  boxData.faceAttributeSet = nil;                    // faces will inherit attribs...
  boxData.boxAttributeSet = attribs;                 // ...from entire box's attribs

  gMyBoxObject = Q3Box_New(&boxData);                // make new box object from data
  if (gMyBoxObject == nil)
    DoFatalAlert("\pError creating new Box geometry");


      /* DISPOSE OF OUR REFERENCE TO THE ATTRIBUTES */

  Q3Object_Dispose(attribs);
}
```

Odds are that after reading this code, you understood most of it. The majority of the code looks similar to code we have seen previously. However, there is one new element: the attributes. The very first thing we do before creating any geometric object is to create an attribute object.

Assigning attributes

When you first create a new attribute object, the object is empty—it has no specific attributes in it. The `Q3AttributeSet_Add()` call is used to add attributes to the attribute object, and you should note that the order in which we add attributes is not important. Our box geometric object has two attributes associated with it: a color and an ambient coefficient. The color is obviously the color which we want to assign to the box. We chose white. The ambient coefficient is crucial for proper lighting, because it determines how much light is reflected from an object's surface. An ambient coefficient of 0.0 means that no light is reflected off the object; thus, the object will appear black no matter what color it is attributed with. Setting this value to 1.0 would be considered "normal," but values higher than 1.0 are perfectly acceptable if you want your object to be a little extra bright. Be careful, however, because polygons with high ambient coefficients do not shade well. High values tend to overexpose the object, thus filling in dark areas, which neutralizes any directional lighting effects.

Defining a Box Object

Creating the actual box geometric object requires nothing more than preparing a `TQ3BoxData` structure and then passing it to `Q3Box_New()`. Some of the `TQ3BoxData` elements should be explained, however. First of all, the `origin` is not the coordinate of the center of the box as you might have assumed. Rather, the `origin` is the coordinate of one of the eight corners of the box. The `orientation`, `majorAxis`, and `minorAxis` values determine the scale of the box along the y, z, and x axes from the `origin` (see figure 9.11).

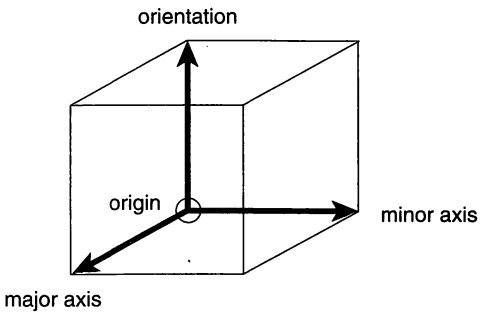


Figure 9.11 *A box geometry.*

The `TQ3BoxData` structure also has two attribute-related values: `boxAttributeSet` and `faceAttributeSet`. We have set the `boxAttributeSet` to be the attribute object that we created at the beginning of the routine. This will apply our color and ambient coefficient attributes to the entire box geometric object. Since we do not wish to apply any different attributes to the faces of the box, we have set `faceAttributeSet` to NIL.

Caution

It is extremely important to set any and all unused attribute references to NIL as we did in the example. If you fail to set any attribute values, your code is almost guaranteed to crash! This is one of the most common causes of fatal crashes while using QuickDraw 3D. Always be aware of what data structures have attribute fields in them, and be sure to set those fields to either an attribute object or NIL!

Referencing and Dereferencing

The final line of code in our example simply disposes of our "reference" to the attribute object. This does not delete the attribute object from memory because our box geometry is using it. Every time an object is assigned to another object, its reference counter is increased by one (all done internally by QuickDraw 3D). As long as an object's reference count is greater than zero, the object will remain in memory. Therefore, it is crucial to always dispose of unneeded references to objects or serious memory leaks will occur in your application. All QuickDraw 3D objects operate in this fashion.

When I use the term "memory leak," I am referring to cases in which memory occupied by an object is either not freed up or is freed when the object is still being used. If objects are not dereferenced enough times, their memory is never freed; thus, your application will eventually run out of memory as it slowly gets filled up by unused objects. If you dereference an object too many times, causing its memory to be freed while it's still being used, the most likely result will be a crash.

For example, suppose we wanted to add our new box object to a group object:

```
myGroupPosition = Q3Group_AddObject(gTheGroup, boxObj);
```

When we call `Q3Group_AddObject()`, the box's reference counter gets incremented by one. There are now two references to the box geometric object: the original reference and the reference from the group object. Should we ever dispose of the group object, it would only dispose of one reference to the box. In other words, our box would not be deleted from memory. To delete the box object from memory, we must call `Q3Object_Dispose()` enough times to decrement its reference count to zero.

```
Q3Object_Dispose(gTheGroup);  // disposes of one reference to the box
Q3Object_Dispose(boxObj);     // disposes of another reference to the box
```

Caution

Next to unassigned attribute values, fudged reference counts are the second most common cause of crashes in QuickDraw 3D. Failing to dereference objects correctly can result in memory leaks and fatal crashes. It is very, very important to meticulously track every reference to every object you create. Whenever you no longer need a reference to an object, call `Q3Object_Dispose()`. Also be aware that deleting a group object will automatically dereference any objects contained within the group.

Drawing Objects

At first glance, the code to draw objects into our window may seem a little peculiar. Don't worry, it *is* peculiar—you're not just seeing things. Each element of a scene must be "submitted" to QuickDraw 3D. Certain objects must be submitted before others. Style and shader objects must be submitted before any geometric objects because QuickDraw 3D needs to know how to draw the geometries before they can be submitted for drawing.

The Draw Loop

The following code shows how to draw the box object we created into the view object that we created earlier.

```
/****************** DRAW MY OBJECTS ********************/
//
// Draws a frame of animation for the fly-through window.
//

void DrawMyObjects(void)
{
TQ3Status         myStatus;
TQ3ViewStatus     myViewStatus;

          /* START RENDERING */

   myStatus = Q3View_StartRendering(gMyViewObject);          // start rendering
   if ( myStatus == kQ3Failure )
      DoFatalAlert("\p Q3View_StartRendering Failed!");

   do
   {
```

```
        /* SET INTERPOLATION STYLE */

    myStatus = Q3Style_Submit(gMyStyleObject_Interpolation,gMyViewObject);
    if ( myStatus == kQ3Failure )
      DoFatalAlert("\p Q3Style_Submit Failed!");

        /* SET BACKFACING STYLE */

    myStatus = Q3Style_Submit(gMyStyleObject_Backfacing,gMyViewObject);
    if ( myStatus == kQ3Failure )
      DoFatalAlert("\p Q3Style_Submit Failed!");


        /* SET FILL STYLE */

    myStatus = Q3Style_Submit(gMyStyleObject_Fillstyle, gMyViewObject);
    if ( myStatus == kQ3Failure )
      DoFatalAlert("\p Q3Style_Submit Failed!");


        /* SET SHADER TO USE */

    myStatus = Q3Shader_Submit(gMyShaderObject_IlluminationShader, gMyViewObject);
    if ( myStatus == kQ3Failure )
      DoFatalAlert("\p Q3Shader_Submit Failed!");


        /* DRAW THE BOX OBJECT */

    myStatus = Q3Geometry_Submit(gMyBoxObject,gMyViewObject);
    if ( myStatus == kQ3Failure )
      DoFatalAlert("\p Drawing object Failed!");


        /* SEE IF DONE */

    myViewStatus = Q3View_EndRendering(gMyViewObject);

  } while ( myViewStatus == kQ3ViewStatusRetraverse );
}
```

The first Toolbox call needed to commence the drawing of our scene is the `Q3View_StartRendering()` call, which takes our view object, `gMyViewObject`, as the input parameter. This tells QuickDraw 3D that we are about to start drawing objects and that we want to use `gMyViewObject` to determine where and how we want the drawing to be done.

Next is the "draw loop." This is a `do-while` loop which repeats as long as `Q3View_EndRendering()` returns the value `kQ3ViewStatusReTraverse`. Usually, only one pass

through the loop is required to successfully complete the drawing of all the objects, but under certain circumstances, multiple passes are required. In general, QuickDraw 3D's Z-Buffer renderer will complete in one pass, but future renderers such as a ray-trace renderer may require the loop, so you should keep it in your code for compatibility.

Submitting Objects

Inside the draw loop we perform several tasks. We first need to "submit" all of the style objects that we created earlier in this chapter. Submitting a style object into the active view object tells QuickDraw 3D to perform all rendering based on the information defined by these styles. Failure to submit the style objects every time may result in an incorrectly rendered scene, since QuickDraw 3D will use its defaults to determine how to do the rendering. The order in which we submit these styles is not important; it's simply important that we do it before any geometric objects are drawn.

After all our style objects, we need to submit our shader object. Like the style objects, the shader object is crucial for proper rendering of the scene and must be submitted before any geometries are drawn. The shader object tells QuickDraw 3D which shader to use when drawing the geometries.

Finally, after submitting our styles and shader object, we are ready to submit the box for drawing. The `Q3Geometry_Submit()` call is used to draw geometric objects. There is another call, `Q3DisplayGroup_Submit()`, which is used to draw groups which contain geometries and other objects.

Updating the Visible Window

When `Q3View_EndRendering()` returns a value other than `kQ3ViewStatusReTraverse`, we know that the rendering of our scene is complete. Because we are using double buffering, the scene was just rendered into a draw buffer which is maintained by QuickDraw 3D. Upon completion of the rendering, QuickDraw 3D automatically dumps this off-screen draw buffer to the window that we defined in the draw context.

Tweaking Parameters

After executing the `DrawMyObjects()` routine, we should see the box drawn in our window. Now that there is something visible on the screen, it would be a good idea to experiment with the parameters of the camera, lights, and box just to see what happens. For example, try moving the camera farther away from the box. Go into the `CreateMyCamera()` routine and change the `cameraFrom` setting:

```
TQ3Point3D              cameraFrom = { 0.0, 0.0, 40.0 };
```

This will move the camera farther way from the box directly along the z axis. When rendered, the box should appear much smaller in the window. Also try changing the camera's

lens by playing with the `fov` value. Try setting it to 1.9 and notice that the scene appears to be viewed through a fish-eye lens.

```
myViewAngleCameraData.fov = 1.9;   // larger = more fish-eye
```

Also try changing the intensity, angle, and color of the ambient and directional lights and observe the results. Performing little experiments like this will give you a much better feel for how QuickDraw 3D works. It's good to have an intuitive feel for how much you need to tweak different values such as light intensity, lens sizes, ambient values, and so on.

Stereoscopic 3-D

At this point I would like to briefly talk about stereoscopic 3-D using those funky old red and blue glasses. It is actually quite simple to get QuickDraw 3D to render a scene that when viewed with 3-D glasses looks amazingly dimensional. The basic principle behind stereoscopic rendering is very simple.

1. You have two cameras, one for each eye: a right camera and a left camera.
2. Aim both cameras at a common point.
3. Render the left camera in one GWorld, the right camera into another.
4. Filter the left camera to red.
5. Filter the right camera to blue.
6. Add the left and right GWorlds together while copying into a new third GWorld.
7. Dump the third GWorld to the screen.

I used this technique in the Gerbils demo which I wrote for Apple. There are, however, some key points to making red-blue stereoscopic images work:

- The source artwork should be as close to grayscale as possible.
- Detailed surfaces emphasize the effect.
- Bright, colored backgrounds also help the effect. Rendering a scene against a white backdrop works well.

The reason that grayscale source art (meaning the colors of the polygons being rendered) works well is that you want an equal balance of red and blue in an object. If an object is blue, then it will be completely filtered out by the red camera. This will destroy any 3-D effect from that object. If an object is red, then the same applies to the blue camera. If your source material cannot be grayscale for logistical reasons (like what I had with Gerbils), then you need to fudge the source data to work.

The way I fudged the source data in Gerbils was to read in the source RGB color for the red camera, then look at the blue and red channels. If the blue channel was brighter than the red

channel, then I set the red channel to the blue channel. In other words, I made the red channel equal to either the red or blue channel, whichever was larger. Then I did the same for the blue camera by taking the largest of the two channels and using it for the blue channel.

The following code sample was taken directly out of Gerbils.

```
/****************** FILTER STEREO WINDOWS ********************/


void FilterStereoWindows(void)
{
Rect         r;
PixMapHandle pixMapL,pixMapR,pixMapC;
unsigned long     rowBytesL,rowBytesR,rowBytesC;
long         width,height,x,y;
unsigned short *pixAddrL,*pixAddrR,*pixAddrC,lc,rc,rr,rb,lb,lr;
unsigned short    *srcPtr,*destPtr,*srcPtr2;
Ptr            srcStartPtr,destStartPtr,srcStartPtr2;

          /* GET LEFT GWORLD INFO */

   pixMapL = GetGWorldPixMap(gStereoWorldLeft);
   LockPixels(pixMapL);
   pixAddrL = (unsigned short *)GetPixBaseAddr(pixMapL);
   rowBytesL = (unsigned long)(**pixMapL).rowBytes & 0x3fff;

          /* GET RIGHT GWORLD INFO */

   pixMapR = GetGWorldPixMap(gStereoWorldRight);
   LockPixels(pixMapR);
   pixAddrR = (unsigned short *)GetPixBaseAddr(pixMapR);
   rowBytesR = (unsigned long)(**pixMapR).rowBytes & 0x3fff;

          /* GET CENTER GWORLD INFO */

   pixMapC = GetGWorldPixMap(gStereoWorldCenter);
   LockPixels(pixMapC);
   pixAddrC = (unsigned short *)GetPixBaseAddr(pixMapC);
   rowBytesC = (unsigned long)(**pixMapC).rowBytes & 0x3fff;

          /* CALC SIZE OF GWORLD */

   r = gStereoWorldLeft->portRect;
   width = (r.right-r.left);
   height = (r.bottom-r.top);

            /* COPY LEFT & RIGHT TO CENTER */
```

```
    srcStartPtr = (Ptr)pixAddrL;
    srcStartPtr2 = (Ptr)pixAddrR;
    destStartPtr = (Ptr)pixAddrC;

    for (y = 0; y < height; y++)
    {
      srcPtr = (unsigned short *)srcStartPtr;
      srcPtr2 = (unsigned short *)srcStartPtr2;
      destPtr = (unsigned short *)destStartPtr;
      for (x = 0; x < width; x++)
      {
        lc = *srcPtr++;              // get left colors
        lr = (lc&(0x1f<<10))>>10;
        lb = lc&0x1f;

        rc = *srcPtr2++;            // get right colors
        rr = (rc&(0x1f<<10))>>10;
        rb = rc&0x1f;

        if (lr > lb)                // see which channel is brightest
          lc = lr;
        else
          lc = lb;

        if (rr > rb)
          rc = rr;
        else
          rc = rb;

        *destPtr++ = (lc<<10)|rc;
      }
      srcStartPtr += rowBytesL;
      srcStartPtr2 += rowBytesR;
      destStartPtr += rowBytesC;
    }
}
```

This approach to doing 3-D stereoscopic effects works relatively well for QuickDraw 3D applications running in small windows on machines without hardware acceleration. Gerbils can run full-screen 640 x 480 at over 30 frames per second with hardware acceleration, but with the 3-D filtering turned on, it slows to about three frames per second. In contrast, Gerbils runs at about ten frames per second in a 320 x 200 window on an unaccelerated Power Mac 6100/60, and only slows to about four frames per second in 3-D mode. What I'm pointing out is that doing stereoscopic filtering with hardware-accelerated systems completely kills your rendering speed, but it only causes a 60 percent performance hit on software-only systems.

All about Meshes

Of all of QuickDraw 3D's built-in geometric objects (boxes, lines, spheres, etc.), none is used more often than the *mesh*. Meshes are the most flexible and efficient of all the geometry types. A mesh is simply a collection of vertices and a collection of faces that use those vertices. What makes meshes so efficient is that one vertex in a mesh can be shared by many different polygons, thus reducing the number of calculations needed to draw complex models (see figure 9.12).

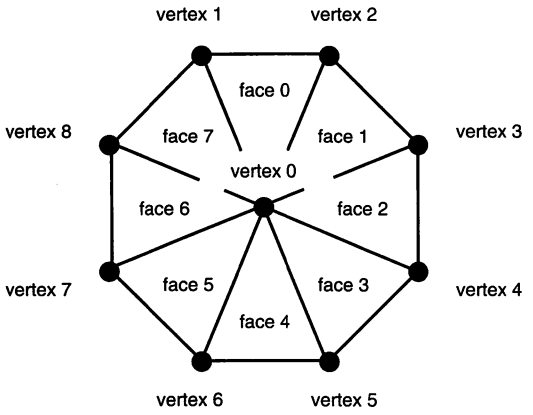


Figure 9.12 *Every triangular polygon face in this mesh shares its vertices with another polygon. Vertex 0 is shared by all the faces.*

Building the Mesh

Remember, a mesh object is your best friend in QuickDraw 3D. It's a good idea to get to know as much as you can about them because you will be using them a lot. Creating a mesh involves two fairly simple steps:

1. Add a bunch of vertices to the mesh.
2. Add a bunch of faces to the mesh.

Figure 9.13 shows us the simple nine-polygon mesh which we are going to create in the following listing.

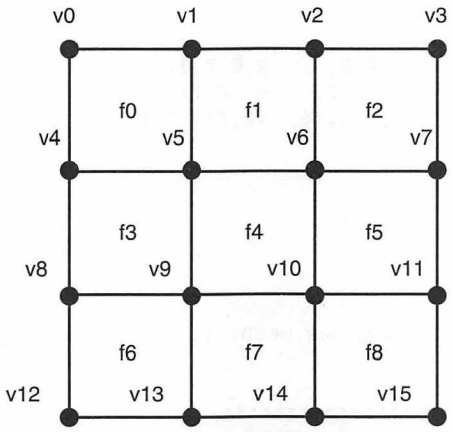


Figure 9.13 *A mesh with 16 vertices and nine faces. Each face is a square polygon.*

```
/****************** CREATE MY MESH ******************/
//
// Creates a mesh geometric object.
//
// INPUT  : none
// OUTPUT : none
//

void CreateMyMesh(void)
{
long              i,faceNum,vertexNum;
float             ambient = 1.0;
TQ3ColorRGB       color;
TQ3AttributeSet   vertexAttribs;
TQ3Vertex3D       vertex;
TQ3MeshFace       face;
TQ3MeshVertex     meshVertexIDList[16];
TQ3MeshVertex     faceVerts[4];
TQ3GeometryObject meshObj;

TQ3Point3D     meshVertexCoords[16] =
            {
               -10,0,-10,   -5,0,-10,    5,0,-10,     10,0,-10,
               -10,0,-5,    -5,0,-5,     5,0,-5,      10,0,-5,
               -10,0,5,     -5,0,5,      5,0,5,       10,0,5,
               -10,0,10,    -5,0,10,     5,0,10,      10,0,10
            };
```

```
long        meshFaceVertices[9*4] =
            {
              0,4,5,1,     1,5,6,2,     2,6,7,3,
              4,8,9,5,     5,9,10,6,    6,10,11,7,
              8,12,13,9,   9,13,14,10,  10,14,15,11
            };


      /* CREATE NEW MESH OBJECT */

  meshObj = Q3Mesh_New();
  if (meshObj == nil)
    DoFatalAlert("\pError making new mesh!");


      /**********************************/
      /* CREATE THE VERTICES OF THE MESH */
      /**********************************/

  for (vertexNum = 0; vertexNum < 16; vertexNum++)
  {
      /* MAKE ATTRIBUTES FOR THIS VERTEX */

    vertexAttribs = Q3AttributeSet_New();                    // make attribs
    if (vertexAttribs == nil)
      DoFatalAlert("\pError making new vertex attribs.");

    color.r = (float)1/((Random()&0xf)+1);                   // make random RGB color
    color.g = (float)1/((Random()&0xf)+1);                   // for each vertex
    color.b = (float)1/((Random()&0xf)+1);

    Q3AttributeSet_Add(vertexAttribs,                        // set color
         kQ3AttributeTypeDiffuseColor, &color);

    Q3AttributeSet_Add(vertexAttribs,                        // set ambient
         kQ3AttributeTypeAmbientCoefficient, &ambient);


         /* SET COORDS & ATTRIBS FOR VERTEX */

    vertex.point.x = meshVertexCoords[vertexNum].x; // copy x,y,z coords
    vertex.point.y = meshVertexCoords[vertexNum].y;
    vertex.point.z = meshVertexCoords[vertexNum].z;
    vertex.attributeSet = vertexAttribs;                     // attach attribs


      /* CREATE A NEW VERTEX & KEEP THE RETURNED VERTEX ID */

    meshVertexIDList[vertexNum] = Q3Mesh_VertexNew(meshObj, &vertex);
```

```
        /* DISPOSE OF OUR REFERENCE TO THE ATTRIBUTE SET */

    Q3Object_Dispose(vertexAttribs);
}

        /*****************************/
        /* CREATE MESH POLYGON FACES */
        /*****************************/

for (i = 0, faceNum=0; faceNum < 9; faceNum++)
{
        /* GET VERTEX ID'S FROM TABLE */

    faceVerts[0] = meshVertexIDList[meshFaceVertices[i++]];      // copy vertex
    faceVerts[1] = meshVertexIDList[meshFaceVertices[i++]];      // IDs into local
    faceVerts[2] = meshVertexIDList[meshFaceVertices[i++]];      // array for easy
    faceVerts[3] = meshVertexIDList[meshFaceVertices[i++]];      // use.


        /* MAKE FACE WITH NO ATTRIBUTES */

    face = Q3Mesh_FaceNew(meshObj,4,&faceVerts[0],nil);          // make the face
    if (face == nil)
        DoFatalAlert("\pError adding face to mesh object!");
}

gMyMeshObject = meshObj;
}
```

First, notice that the variables declaration contains two tables: `meshVertexCoords` and `meshFaceVertices`. The `meshVertexCoords` table simply contains a list of 16 coordinate points that define all of the 16 vertices of the mesh in figure 9.13. The next table, `meshFaceVertices`, is a list of vertex numbers. Each polygon in the mesh has four vertices, and this table refers to the vertices in `meshVertexCoords`. The first polygon, for example, uses vertices 0, 4, 5, and 1, in that order. Remember that the vertices must be ordered counterclockwise so that the lighting and backface removal will be correct.

After creating an empty mesh object with the `Q3Mesh_New()` call, we need to add our vertices to the mesh. Adding vertices is as simple as setting a `TQ3Vertex3D` structure that contains a reference to an attribute set and the coordinates of a vertex, and then passing it to `Q3Mesh_VertexNew()`.

Gouraud Shading with Color Attributes

We're going to demonstrate two things at once with this mesh code. Not only are we learning how to build a mesh, but we are going to see how to perform color interpolation shading (often called Gouraud shading) on our polygons. The code that creates the attribute set for each vertex is generating a random RGB color value:

```
color.r = (float)1/((Random()&0xf)+1);          // make random RGB color
color.g = (float)1/((Random()&0xf)+1);          // for each vertex
color.b = (float)1/((Random()&0xf)+1);
```

Since each vertex of a polygon will have a different color attribute, the colors across the face of the polygon will be interpolated from vertex to vertex. In other words, our polygon will be smoothly shaded with a range of colors, giving it a very attractive appearance. Also note that since we are applying a color attribute to every vertex, any color attributes assigned to the faces or the entire geometric object will be overridden due to the inheritance rule discussed earlier. And of course we dispose of our reference to the attribute object when we are done with it.

Adding the vertices into the mesh as we have just done does not create any polygons. If drawn, this mesh would be blank. Without faces, the vertices are meaningless points in space. Creating faces is even easier than creating the vertices.

Making Polygon Faces from Vertices

When we created each vertex, we were returned a vertex ID number from the `Q3Mesh_VertexNew()` call. When we create faces, we use these returned values to identify which vertices each face is comprised of. The best way to do this is to create an array of four `TQ3MeshVertex` types and set them to the returned vertex ID numbers. This array, along with a vertex count (four for each of these faces) and an attribute reference (NIL in our example), is passed to `Q3Mesh_FaceNew()`, which creates the face in the mesh.

Caution

I should point out that QuickDraw 3D does not check to see if you are adding duplicate vertices into a mesh. If you accidentally add two identical vertices into a mesh, everything will appear correctly, but internally, it is processing both copies of the same vertex, which causes unnecessary processor overhead. When creating your mesh objects, be very careful to have as many vertices shared by as many faces as possible. This will greatly improve the performance and memory requirements of your application.

Applying Transforms to an Object

At this point, our mesh object is complete. We've added the vertices and built polygons out of them, but rather than return our mesh object to the calling routine, we will add the mesh object into a group object. In a real QuickDraw 3D application, almost everything will be included in a group. Drawing individual geometric objects is very rare, mainly because to perform any kind of rotation, you must include a rotation transformation object in the group

of the geometric object you wish to rotate. You cannot directly apply rotation to a geometry; rather, you must apply the rotation to the group in which the geometry exists.

Rotating the Mesh Object

Let's try to add rotation to the mesh object that we just created in the code sample above. We'll change the end of the routine to look like this:


```
        /*********************/
        /*  DO THE ROTATION  */
        /*********************/


    /* CREATE A GROUP WITH ROTATION TRANSFORMS */

  groupObject = MakeMyRotationGroup(0.1,0.1,-0.4);


        /* ADD MESH TO THE GROUP */

  myGroupPosition = Q3Group_AddObject(groupObject, meshObj);
  if ( myGroupPosition == nil )
    DoFatalAlert("\pQ3Group_AddObject failed!");


      /* DISPOSE OF OUR REFERENCE TO THE MESH OBJECT */

  Q3Object_Dispose(meshObj);

  gMyMeshGroup = groupObject;
}
```

The new ending of `CreateMyMesh()` calls a new routine `MakeMyRotationGroup()`. This routine, which is listed below, returns a group object that contains three rotate transform objects: an x-axis, y-axis, and z-axis rotate. The mesh object is then attached to the end of the group, after the transform objects.

```
/***************** MAKE MY ROTATION GROUP ********************/
//
// Creates a rotate transform object for all three axes, then combines them
// into a single rotation group object which is returned to the caller.
//
// INPUT: x,y,z = radian degrees to rotate on each axis
//

TQ3GroupObject MakeMyRotationGroup(float x, float y, float z)
{
TQ3GroupObject      rotGroup;
TQ3RotateTransformData xformData;
```

```
TQ3TransformObject     rotObject;
TQ3GroupPosition    myGroupPosition;

            /* CREATE A GROUP */

  rotGroup= Q3OrderedDisplayGroup_New();
  if ( rotGroup== nil )
    DoFatalAlert("\pCouldnt make rotation group!");


       /* CREATE AN X-AXIS ROTATE TRANSFORMATION */

  xformData.axis = kQ3AxisX;
  xformData.radians = x;
  rotObject = Q3RotateTransform_New(&xformData);          // create new xform object

  myGroupPosition = Q3Group_AddObject(rotGroup, rotObject);// add to group
  if ( myGroupPosition == 0 )
    DoFatalAlert("\pError adding x-axis rotation object to rot group!");
  Q3Object_Dispose(rotObject);                            // dispose of our reference


        /* CREATE A Y-AXIS ROTATE XFORM */

  xformData.axis = kQ3AxisY;
  xformData.radians = y;
  rotObject = Q3RotateTransform_New(&xformData);          // create new xform object

  myGroupPosition = Q3Group_AddObject(rotGroup, rotObject);// add to group
  if ( myGroupPosition == 0 )
    DoFatalAlert("\pError adding y-axis rotation object to rot group!");
  Q3Object_Dispose(rotObject);                            // dispose of our reference


        /* CREATE A Z-AXIS ROTATE XFORM */

  xformData.axis = kQ3AxisZ;
  xformData.radians = z;
  rotObject = Q3RotateTransform_New(&xformData);          // create new xform object

  myGroupPosition = Q3Group_AddObject(rotGroup, rotObject);// add to group
  if ( myGroupPosition == 0 )
    DoFatalAlert("\pError adding z-axis rotation object to rot group!");
  Q3Object_Dispose(rotObject);                            // dispose of our reference


             /* RETURN THE GROUP */

  return(rotGroup);
}
```

This routine creates a new group which will hold our rotate transform objects. The `Q3RotateTransform()` routine creates each rotate transform object—one for each axis. When this group is submitted for drawing, it will cause any geometric objects also contained within the group to be rotated by the indicated values x, y, and z. Remember that QuickDraw 3D uses radians, not degrees. A full circle is 2π radians (or approximately 6.28 radians), not 360 degrees. Half a circle is π (or approximately 3.14) radians (see figure 9.14).

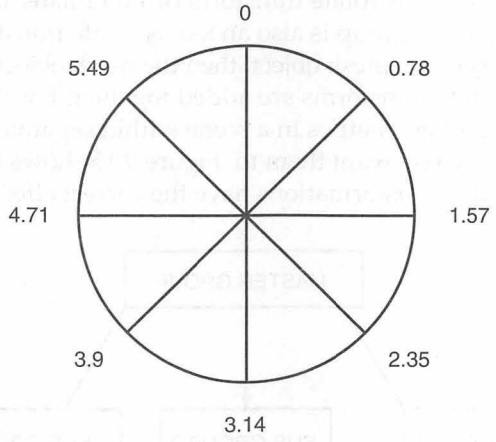


Figure 9.14 *Radian values along a circle.*

When this group is passed back to `CreateMyMesh()`, the mesh object is attached to the group. The end result is a single group object that contains three rotate transforms and a mesh. Since we are no longer handling the mesh object directly, we need to make one small change to the draw code. Previously, we used the following code to draw the mesh object:

```
        /* DRAW THE MESH GEOMETRIC OBJECT */

    myStatus = Q3Geometry_Submit(gMyMeshObject,gMyViewObject);
```

If you attempt to use `Q3Geometry_Submit()` to draw a group object, nothing will happen and you'll be hacking at code for hours trying to figure out where the mesh has vanished to. It is very important to remember that in order to draw a group object, you must use `Q3DisplayGroup_Submit()`.

```
        /* DRAW THE MESH GROUP OBJECT */

myStatus = Q3DisplayGroup_Submit(gMyMeshGroup,gMyViewObject);
```

Transformation Scope

When you draw the mesh object with a call to `Q3DisplayGroup_Submit()`, QuickDraw 3D pushes the current transform information onto a stack and then resets the current transform state. The old state is restored when the group has completed drawing.

All transform operations are additive in the sense that they add to any existing transform state. Say your group has an x-axis rotate transform of 1.0 radians as the first object. Then suppose the second object in the group is also an x-axis rotate transform, but with a value of -2.1 radians. If the third object is a mesh object, then the mesh object will be rotated by -1.1 radians because the two rotate transforms are added together. For this reason, it is extremely important that you enclose all geometries in a scene within separate groups so that your transforms come out the way you want them to. Figure 9.15 shows how objects might be ordered in groups so that the transformations have the correct effect.

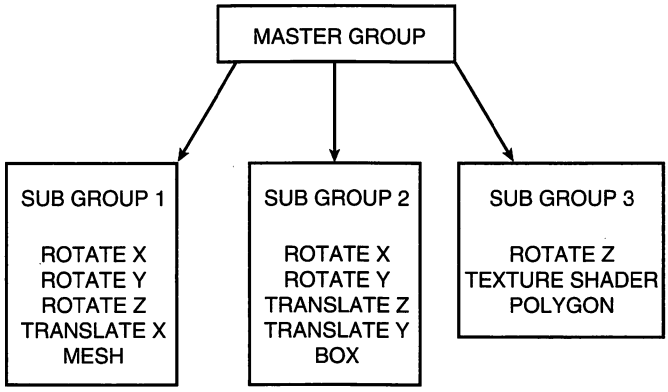


Figure 9.15 *Sample group hierarchy for correct transformations.*

Other Transform Objects

It should be noted that there are many different kinds of transformation objects in addition to rotation transformations:

- *Translate Transforms*—Move an object in 3-D space.
- *Scale Transforms*—Scale an object's dimensions along the x, y, and/or z axes.
- *Matrix Transforms*—Do specific matrix transformations. For the advanced user who understands the concept of matrices and the details of 3-D mathematics. It can be a more powerful and efficient way of performing all of the other transformations since all the other transformations eventually get converted into a matrix representation.

- *Rotate-about-Point Transforms*—Rotate an object around a point in space rather than its own origin.
- *Rotate-about-Axis Transforms*—Rotate an object around an arbitrary axis in 3-D space.
- *Quaternion Transformations*—Rotate an object using four-dimensional spatial mathematics. For people who really know what they're doing.

Each of these transform objects works exactly like the rotate transform that we demonstrated previously. You simply create the transform object, add it to a transform group if you wish, then add it to the geometry's group.

Using Translation and Rotation in Game Design

The rotate and translate transforms are the two most commonly used transform objects. You'll need the translate transform to move your geometric objects in space. Until now, we have simply forced the coordinates of our objects by setting their vertices to the actual 3-D space coordinates where we want the object to appear. But when we want to move one of those objects through space, we need to apply translation to it, because generating a new geometry with new vertex coordinates would be incredibly slow and inefficient.

All geometric objects should be built as though they are centered around coordinates (0, 0, 0). If you actually want the object to appear at (100, -30, 19), then you should translate the object to that coordinate rather than build it at those coordinates. The main reason for this is that applying a standard rotate transform to an object built around (100, -30, 19) would cause it to rotate around the world's origin at (0, 0, 0), not around the object's origin at (100, -30, 19).

The other, more practical reason for building all objects around (0, 0, 0) is that you will most likely be placing many copies of the same object at different coordinates in 3-D space. For example, suppose you create an airplane. There might be ten of these objects in the scene at one time. You will create only one original mesh object which represents the airplane, but you will create a group which contains a rotate transform, a translate transform, and a reference to the master mesh object for each airplane in the scene. This way, you can set ten different rotation values and ten different coordinate locations for each airplane, while using only one source object.

Texture Maps

Although it causes a fairly noticeable performance hit, you will undoubtedly want to use at least *some* texture-mapped objects in your games. QuickDraw 3D certainly does an excellent job of flat and Gouraud shading, but no modern video game looks up to spec without texture mapping. A clever game designer will try to combine flat, Gouraud, and textured polygons to get optimal performance while maintaining the best appearance.

(u,v) Coordinates

The basic idea behind texture mapping is simple. First, you have a bitmap of a texture in memory. This could be a picture of wood, clouds, metal, fabric, and so on. For every vertex of a polygon which you are going to texture map, you assign what are called "(u,v)" coordinates (also known as parametric coordinates). A (u,v) coordinate is a 2-D (x,y) coordinate relative to the dimensions of the texture map that ranges from 0.0 to 1.0. In other words, the (u,v) coordinate for the lower left pixel of a texture map is always (0, 0). The coordinate for the upper right pixel of a texture map is (1, 1). You can think of the (u,v)'s as percentages from 0.0 percent to 100.0 percent of the total width or height of the bitmap. The u coordinate for the horizontal center of the bitmap is 0.5 or 50 percent.

Note

QuickDraw 3D defies the norm in that its v coordinate goes 0.0 to 1.0 from bottom to top, not top to bottom as in most 3-D graphics systems. If you are used to (u,v) coordinates which have (0, 0) in the upper left corner, you may find that you have to pay closer attention to what you are doing, or your texture maps will be mapped upside down. Most 3-D systems that I have worked with have the v's going from top to bottom. Certain modeling tools such as 3D Studio also output (u,v) coordinates with the v's starting at the top.

The reason that (u,v)s are treated as percentages and not physical coordinates on the bitmap is so bitmaps can be changed without having to reset the (u,v)s for every polygon that uses the bitmap. It keeps the mapping of the texture consistent even if the texture changes. Suppose you texture map a square polygon with a bitmap with dimensions of 400 x 400 pixels. The (u,v)s for the four vertices of the polygon would appear as they do in figure 9.16.

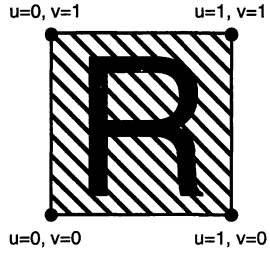


Figure 9.16 *The (u,v) coordinates for a square polygon.*

What if we wanted to replace that 400 x 400 bitmap with a 100 x 100 bitmap? No problem, because our (u,v) coordinates do not change, they remain exactly as they were. Even if our bitmap is 45 x 20, our (u,v) coordinates work just fine to display the entire bitmap in the polygon. (0, 1) is still the upper left corner and (1, 0) is still the lower right corner.

Normally, replacing a texture map would be a rarely used feature in an application, but games can make great use of this ability because what we can achieve are animated textures. Animating textures is one of the easiest and most efficient ways to add spectacular animation to your game. Suppose you have a 3-D landscape that you've been bombarding from an airplane. You could add animating flames to your impact areas quickly and easily. Each time a bomb explodes on the terrain, create one single square polygon which displays an animating texture map of an explosion or flames. The polygons need to be set only once with their (u,v) coordinates, and all you have to do is play a PICS movie or even a QuickTime movie into the polygon's bitmap to cause all of the polygons to animate. However, this may not be possible with the initial release of QD 3D.

There are many different ways to assign texture maps to objects, so once again I will refer you to *Inside Macintosh: QuickDraw 3D* for all the gritty details. For my little texture mapping demonstration, I will show one method for applying a texture map to a simple polygon object and setting the (u,v) coordinates for the vertices of the polygon.

Texture Shader Objects

One thing common to all texture mapping methods in QuickDraw 3D is the texture shader object. This object contains all of the information needed to define a texture map such as the pixel map, height, width, color depth, and so on. Creating a texture shader object is a somewhat tedious process, but once you've done it a few times, it won't bother you too much.

```
/************ GET MY TEXTURE MAP FROM 'PICT' ******************/
//
// Loads a 'PICT' resource and returns a shader object which is
// based on the 'PICT' converted to a texture map.
//
// INPUT: textureRezID = resource ID of texture 'PICT' to get.
//
// OUTPUT: TQ3ShaderObject = shader object for texture map.
//

TQ3ShaderObject   GetMyTextureMapFromPICT(long textureRezID)
{
PicHandle           picture;
TQ3StoragePixmap    pixmap;
TQ3TextureObject    texture;
TQ3ShaderObject     shader;


          /* LOAD 'PICT' RESOURCE */

  picture = GetPicture(textureRezID);
  if (picture == nil)
    DoFatalAlert("\pUnable to load texture PICT resource");


       /* SET PIXMAP DATA FOR 'PICT' */
```

```
    CreateMyTexturePixmapData(picture,&pixmap);


        /* MAKE NEW PIXMAP TEXTURE OBJECT */

    texture = Q3PixmapTexture_New (&pixmap);
    if (texture == nil)
      DoFatalAlert("\pError calling Q3PixmapTexture_New!");

    shader = Q3TextureShader_New (texture);
    if (shader == nil)
      DoFatalAlert("\pError calling Q3TextureShader_New!");


        /* DISPOSE OF UNNEEDED THINGS */

    Q3Object_Dispose (texture);
    Q3Object_Dispose (pixmap.image); // Disposes of extra reference to storage obj
                                     // from CreateMyTexturePixmapData()

    ReleaseResource((Handle)picture);

    return(shader);
}
```

The above routine shows the basic flow of events required to make a texture shader object. The first few lines of code are good old resource manager routines that load a 'PICT' resource from the current resource file. We pass the picture to the `CreateMyTexturePixmapData()` routine, which is listed below. `CreateMyTexturePixmapData()` sets up a `TQ3Pixmap` structure which is used to create the `TQ3TextureObject`. In turn, the `TQ3TextureObject` object is passed to `Q3TextureShader_New()` to create the final `TQ3ShaderObject`. Make sense? Probably not. Like I said, it's a somewhat tedious process, so the best way to understand it is to look closely at the code, which makes more sense than any verbal explanation could.

```
/************* CREATE MY TEXTURE PIXMAP DATA ********************/
//
// This routine takes a 'PICT', draws it into a GWorld, then uses the GWorld
// to define the parameters for a storage pixmap object. The object is not made
// here, but the data is passed back -- the calling routine uses it to create
// the TQ3StoragePixmap object.
//
// INPUT  : pict = 'PICT' resource handle
//
// OUTPUT : pixmapData = pointer to TQ3StoragePixmap data structure
//

void CreateMyTexturePixmapData(PicHandle pict, TQ3StoragePixmap *pixmapData)
{
```

```
Ptr             pixAddr;
Rect            rectGW;
GWorldPtr       pGWorld,oldGW;
PixMapHandle    hPixMap;
OSErr           myErr;
GDHandle        oldGD;
unsigned long   size,mapSizeX,mapSizeY,rowBytes;
TQ3StorageObject storageObj;


        /* CALC SIZE OF 'PICT' */

  mapSizeX =  (**pict).picFrame.right - (**pict).picFrame.left;
  mapSizeY =  (**pict).picFrame.bottom - (**pict).picFrame.top,


      /* CREATE A GWORLD TO DRAW 'PICT' */

  SetRect(&rectGW, 0, 0, mapSizeX, mapSizeY);              // set dimensions
  myErr = NewGWorld(&pGWorld, 32, &rectGW, 0, 0, 0L);      // make 32-bit GWorld ➥
(must be 32-bit!)
  if (myErr)
    DoFatalAlert("\pError making texture GWorld!");


    /* CALC GWORLD'S PIXMAP ADDR & ROWBYTES */

  hPixMap = GetGWorldPixMap(pGWorld);
  pixAddr = GetPixBaseAddr(hPixMap);
    // get addr of pixels
  rowBytes = (unsigned long)(**hPixMap).rowBytes & 0x3fff; // calc rowbytes


      /* DRAW PICTURE INTO GWORLD */

  GetGWorld(&oldGW, &oldGD);                  // save current port
  SetGWorld(pGWorld, nil);                    // set port to GWorld
  LockPixels(hPixMap);                        // lock pixels
  EraseRect(&rectGW);                         // clear the GWorld
  DrawPicture(pict, &rectGW);                 // draw 'PICT'
  SetGWorld(oldGW, oldGD);                    // restore port


    /* CREATE A MEMORY STORAGE OBJECT */

  size = rowBytes * mapSizeY;                        // memory = rowBytes * #rows
  storageObj = Q3MemoryStorage_New(pixAddr, size);  // make new storage object
```

```
        /* SET PIXMAP DATA FIELDS */

    pixmapData->image    = storageObj;                  // image = storage object
    pixmapData->width    = mapSizeX;                    // set pixel width
    pixmapData->height     = mapSizeY;                  // set pixel height
    pixmapData->rowBytes   = rowBytes;                  // set rowBytes
    pixmapData->pixelSize  = 32;                        // 32-bit depth
    pixmapData->pixelType  = kQ3PixelTypeRGB32;         // ditto
    pixmapData->bitOrder   = kQ3EndianBig;              // Mac bit & byte order
    pixmapData->byteOrder  = kQ3EndianBig;
}
```

Here's the basic outline of what is happening here:

1. We calculate the size of the 'PICT'.
2. We create a GWorld in which to draw the 'PICT'.
3. We draw the 'PICT' into the GWorld.
4. We create a memory storage object which is set to the GWorld.
5. Fields of the `pixmapData` structure are filled in and passed back.

We create a temporary GWorld so that we have somewhere to draw the 'PICT' that is passed into the routine. Because QuickDraw 3D is about to directly access the memory occupied by the GWorld, we need to calculate the GWorld's pixmap memory address and its `rowBytes` value. QuickDraw 3D doesn't know about GWorlds; it knows only about pixmaps. Because of this, we must tell QuickDraw 3D all of the pixmap-specific information about the GWorld so it knows how to work with it. The pixel address and `rowBytes` value are all QuickDraw 3D needs to know.

Now we only need to set the fields of `pixmapData`. First, we assign a reference to the memory storage in the `image` field, then we set the pixmap's dimensions and `rowBytes` values. The next two fields, `pixelSize` and `pixelType`, set the color depth of the pixel data, which we know is 32 bits. In QuickDraw 3D 1.0, texture maps *must* be 32 bits. The remaining two fields are for cross-platform compatibility and you probably won't ever have to worry about them. They determine the byte order of the pixel data, and since we are using Macintoshes, these values should always be set to `kQ3Endian_Big`.

Here's a trick you can play with the `image` field of the `TQ3Pixmap` structure. Suppose you set it to point to the screen address instead of to the address of an off-screen buffer. When you draw a polygon with a texture shader made from that `TQ3Pixmap`, the polygon will have the image of the entire screen on it. This would cause the infinite-mirror effect, since each polygon would have on it a picture of the previous frame of animation, which would have a polygon with a picture of the previous frame of animation, etc., etc., etc. I don't know what you could ever use this effect for, but it's pretty cool anyway! Kinda psychedelic.

There's one important note to the animating texture theory. QuickDraw 3D will cache your texture into its own private memory or the accelerator card. When this happens, QuickDraw 3D no longer looks at the memory you designated for the image, so putting a new 'PICT' into our GWorld will not change the image which is texture-mapped onto an object. In order to tell QuickDraw 3D to update its cache with a new pixel map, all we need to do is tell QuickDraw 3D that the data has changed. The best way to do this is as follows:

```
Q3TextureShader_GetTexture(shader,&texture);      // get texture object
Q3PixmapTexture_GetPixmap(texture, &pixmap);      // get pixmap from texture

DrawNewPICTIntoOldGWorld(thePict,&pixmap);        // draw new data into GWorld

Q3PixmapTexture_SetPixmap(texture,&pixmap);       // let QD 3D know it changed
```

Calling `Q3PixmapTexture_SetPixmap()` will notify QuickDraw 3D that it needs to flush its texture caches and load the new pixel data.

Mapping a Polygon

We now have a texture shader object, so let's see how to use it. We're going to make a simple polygon object and the texture map will be stretched over its entire surface.

```
/***************** MAKE TEXTURED POLYGON GROUP *****************/
//
// Creates a simple four-sided polygon object which has
// (u,v) attributes for the four vertices. The polygon and
// the texture shader object are placed into a new group object.
//

void MakeTexturedPolygonGroup(void)
{
long              i;
TQ3GeometryObject polygonObj;
TQ3GroupPosition  myGroupPosition;
TQ3PolygonData    polygonData;
TQ3Vertex3D       vertices[4] = {-20,0,20,nil,
                                 20,0,20,nil,
                                 20,0,-20,nil,
                                 -20,0,-20,nil};
TQ3AttributeSet   attribs[4];
float             ambient = 1.0;
TQ3Param2D        uv[4] = {0,0,1,0,1,1,0,1};


      /* CREATE A GROUP */

  gMyPolygonGroup = Q3OrderedDisplayGroup_New();
  if (gMyPolygonGroup == nil)
    DoFatalAlert("\pCouldnt make the group!");
```

```
        /* ADD TEXTURE SHADER OBJECT TO GROUP */

    myGroupPosition = Q3Group_AddObject(gMyPolygonGroup, gMyTextureShaderObject);
    if ( myGroupPosition == nil )
      DoFatalAlert("\pQ3Group_AddObject failed!");


        /* CREATE VERTICES OF POLYGON & SET UV'S */

    for (i=0; i < 4;i++)
    {
      attribs[i] = Q3AttributeSet_New();              // make new attribute set
      Q3AttributeSet_Add(attribs[i],                  // add uv attribute
            kQ3AttributeTypeShadingUV, &uv[i]);
      vertices[i].attributeSet = attribs[i];
    }

        /* CREATE NEW POLYGON OBJECT */

    polygonData.numVertices = 4;                      // four-sided polygon
    polygonData.vertices = vertices;                  // point to our array of vertices
    polygonData.polygonAttributeSet = nil;            // polygon itself has no attribs
    polygonObj = Q3Polygon_New(&polygonData);         // create it


      /* DISPOSE OF OTHER REFERENCES TO ATTRIBUTES */

    for (i=0; i < 4; i++)
      Q3Object_Dispose(attribs[i]);

          /* ADD POLYGON INTO GROUP */

    myGroupPosition = Q3Group_AddObject(gMyPolygonGroup, polygonObj);
    if ( myGroupPosition == nil )
      DoFatalAlert("\pQ3Group_AddObject failed!");

        /* DISPOSE OF REFERENCE TO POLYGON */

    Q3Object_Dispose(polygonObj);
}
```

We create the polygon object just like any other object in QuickDraw 3D, but the main difference here is that we needed to create (u,v) attributes. (u,v) attributes are created with a `Q3AttributeSet_Add()` call just like any other attribute. The only input parameter is a `TQ3Param2D` structure that contains the (u,v) values. Note that each of the four vertices of the polygon have different (u,v)s that correspond to the four corners of the texture map.

At the end of the routine lies the only other bit of different code. Before adding the polygon geometric object to the new group object, we add our texture shader object. This way, when we draw the group object, the shader object will be "submitted" first; thus, any subsequent drawing within the group will use that shader. Refer back to figure 9.15 for more information on group hierarchy.

You should now have a pretty good understanding of QuickDraw 3D mechanics. Like I've said dozens of times in this chapter, QuickDraw 3D is massive, and to fully understand its complete potential, you'll need to read *Inside Macintosh: QuickDraw 3D* several times from front cover to back. I've gone over the areas that are most important to game developers, but without a doubt you will end up using many of the other functions that QuickDraw 3D supplies. The 3-D Math Utilities are indispensable. There are routines for performing 3-D mathematical operations to do just about everything. If your game uses the mouse for 3-D interaction, then you will very likely end up using pick objects for doing collision detection between the cursor and geometries.

As Power Macintoshes become more and more prevalent, I'm sure that you will begin to see a bounty of applications which make use of QuickDraw 3D. And with the hardware acceleration, I would expect to see some amazing visual effects in what would otherwise be mundane applications. With a little patience and a lot of creativity, you'll be making 3-D games in no time. What you once thought would take years will now take weeks with QuickDraw 3D.

Profile: Phil Simon

Phil is a programmer with Drew Pictures who was heavily involved in the development of Iron Helix, one of the early CD-ROM games.

Q: Okay, so how big was the Iron Helix project? How long did it take and how many people were involved?

A: I think it took 13 months of full-scale production, not counting the preproduction work. There were about six people, and there were contractors that did other parts of the game.

Q: You talk about preproduction—what exactly was that?

A: Preproduction involved designing the ship and its interior rooms. It also involved working out game play and doing video prototypes to make sure the compression techniques we used were going to work and play correctly off of CD-ROM.

continued

There wasn't any QuickTime in Iron Helix; it was all PACO. At the time, QuickTime 1.0 had just come out, and it could do only 160 x 120-pixel windows. It had a really limited frame range, and the graphics compressor that was available at the time wasn't too good. Cinepack wasn't out yet, so the digital video performance was pretty lame. So we decided to use PACO. That was a typical design decision.

Essentially, the graphics window and navigation window, where you're actually streaming data off of the CD-ROM, was scaled accordingly. We kept paring down the size of these windows until it could play off a 68020-based system with a single-speed (150K/sec) CD-ROM drive. And for a 386-based 25 MHz PC.

Basically, you do various tasks and you burn one copy off. Then you run off this platter and test it to see what happens. It came down to just burning a lot of CDs. Crunching the window down to different sizes and seeing what would play on that type of machine. That essentially was the governing design factor in that game, because technically most of the game play was about streaming data off of the CD-ROM. You just had to make sure that your window's size was small enough.

Q: Did you do any arrangement of the data on the platter to help with the data streaming?

A: That was another thing. There was the naming convention of the files, which allows them to be laid out in the closest possible manner so that the head seek times are minimized on the CD-ROM. There are also redundant copies of those files that you use a lot. For example, there are certain door openings and things that happen all over the place, so they are duplicated. The drive head doesn't move as much, since it doesn't have to do a seek to the directory. I think actually Jay did some stuff to the directory too so that drive didn't have to seek to the directory. So, there's only one seek—the head just went to where it needed to go. Usually that file was right ahead or right behind where the head was currently located. When the head parks, you would be wherever you were. And the naming convention was set up so the head would just have to move forward or backward.

Q: So it was carefully arranged so that wherever you were in the ship layout, the next files (say, if you backed up or went forward) were physically adjacent to one another on the platter.

A: Exactly. And that's done through the naming convention. Because when you burn the CD-ROM—that is, you transfer the files to it—they're going to be laid down based on the names. So depending on how you name the file, that's how it gets all arranged on the CD. I don't know exactly what Jay's scheme was, but Jay's the guy that programmed it.

Q: What part of the program did you do?

A: I did all the videos. I did the Sega conversion, and I did animations for the game ending.

Q: How did you do these videos?

A: Those were all shot on High-8. They were digitized, at the time, with Radius's VideoVision capture board, which had just come out. The captured video was edited in Adobe Premiere. All the special effects were done in Premiere, such as that green grid overlay and static. They were taken out of QuickTime at six frames per second, and then they were converted to PACO.

Q: How difficult was that? Did you get an actor, make him or her up, have light men fiddle with the illumination, aim the camera at them, and then rehearse? Did you make a couple of tries and then splice everything together?

A: Well, in Iron Helix, the only real actor was Admiral Arbock. For everybody else, we just got friends and people we knew to be in it. So it wasn't really a professional thing in terms of the talent. It was shot in a studio in a week. We just banged the whole thing out. Out of all the production elements within that game, that one was pretty straightforward. You go in there, shoot the video, and then you edit it. There was really nothing complex about that at all.

In the game we're working on now, we have built sets, we have real actors—we took it a step further. The only technically interesting thing we did on Iron Helix was shoot it with all blue screens. We had a blue screen background used to key in various backgrounds of the ship. We would just set the person down in front of the blue screen and key up the background using TrueVision's NuVista+ display board, so that we could see the actor's live video and the ship's background at the same time. We'd adjust the lighting and adjust the camera angles to rough it in. Then we'd record and do the blue screen composite on that background later.

Q: So the blue screen was so you could do the digital chroma key and merge the video of the background with the video of the actor?

A: Yes. We'd merge the background with the actor. Because of all that green screen special effect, the procedure was really forgiving. We didn't have to worry about blue spill or the actor's appearance being problematic. Since everything was converted to grayscale and then tinted green, it was a very forgiving process. You really didn't have to worry too much about the keys. On the game we've got going now, we have full-on blue screen and we have much more problematic keys that we're pulling, so we have to get more sophisticated with our methods.

continued

Q: Now, a little bit on Iron Helix. On the game play, you did some testing. Was there anything you found out, such as having to reorganize the ship layout, or making the defense robot more vicious?

A: Somewhat. They did that with a focus group. Spectrum HoloByte brought in people and gave them the game for ten days. Then they brought the people back and, using a two-way mirror arrangement, they had a conference where they asked them questions. One of the things that was changed was you needed the robot arm to open all the doors. People really hated that, because you had to pull out that arm every time you wanted to go through a door. So they eliminated that and just made it a keyboard command.

They made the changes that they could make. You have to take all that stuff with a certain grain of salt, because if players say they hate the game totally, then you're in big trouble. But if people are actually enthusiastic about the game, that's different. There were some things that we could change, and some things that we couldn't. And the things that we could change, we did just that. For example, there were some control issues with the robot arm that were changed. I think there were some mapping issues, like the way the maps worked, that they modified. They modified where clues were, so it's easier to find certain clues on lower levels of the game. They didn't really modify the ship or its layout. They tweaked things they could tweak, like how vicious the robot was, where clues were, how the interface works, things like that.

Q: You said there were certain things that couldn't be changed. What was an example of one of those?

A: The layout of the ship is one. That's something that you simply say, tough. Or, they wanted the video window bigger. Or, I wish it was faster. You just have to say that there's nothing we can do. Some things are inherent in the design. For the video window, there was talk about pixel doubling that image. For the Sega CD version, essentially that's what happens. But on the Mac and PC versions, there was a lot of debate as to whether we should do it or not. The graphic quality was the issue that kept us at a smaller window size. With pixel doubling, while the image is twice as big, it gets twice as crunchy, and they wanted the graphic quality to remain high. So we didn't pixel double it.

Q: That's understandable. You've mentioned your working on another game right now. Is that something you can talk about?

A: Yeah, it's called Bad Mojo. It's a game where you are transported into the body of a roach and you go on a quest through this apartment building. You move up from the basement, through the various floors, to the top floor, in quest of a way to turn yourself back into a human. The whole game takes place from a roach-eye view, so perspectives are very strange, and your perception of the world is as a bug. You encounter other insects, spiders, goop, weird sticky

traps, and poison. All of the things that a roach would encounter on its journey through a building.

Q: What about 3-D stuff? What can you say about that?

A: All the games I've worked on were games where 3-D graphics were prerendered.

Q: In other words, the graphics were created offline, and all you get are bitmaps of the 3-D images. That's what actually gets employed in the game.

A: Yeah. Bad Mojo is a screen-based game and there's a thousand screens. They're all prerendered and all tiled together. These prerendered images that we make with Electric Image become our backgrounds. Or in other cases, like Iron Helix, it was prerendered animations that we were spooling off the disc.

Q: Such as the defense robot approaching and shooting at you...

A: Or just the change in scenery as you move from one place to another. All of that animation is prerendered. The plus side of that approach is that you can get high graphic quality. You can get nicely rendered objects with good texture maps, which is something you can't do when you're generating them on the fly. For that, you're working at a lower resolution with a lower polygon count, and your texture maps are limited.

Q: Is there a threshold or a break point where you would say, "For 3-D imagery, let's just prerender it, versus generating it on the fly"? Or was this something you would recommend for the sake of aesthetics? And perhaps performance?

A: Well, it really depends on the type of game you want to make. If the stuff is prerendered, there's certain types of things you can and can't do. I think we're probably going to do a combination of both techniques. When we did Iron Helix, it used off-the-shelf software. If you get into real-time 3-D, you are going to buy a real-time 3-D engine. There are such engines available, but they're not as good as the cutting-edge stuff. Or, you have to devote time and energy to developing your own 3-D engine. That type of work is time-consuming and can get costly. If you want to do the real-time stuff, you're in a technology race. If you want to do prerendered stuff, there's off-the-shelf stuff that you can use right now. To enter that market, starting off doing the prerendered stuff is easier.

The Bad Mojo game combines a 3-D sprite that's generated on the fly with backgrounds that are prerendered. So we're combining the two in a way. In the future, we're going to be doing more and more 3-D engine stuff. We're developing a 3-D engine right now. That's going to generate more possibilities where we can do 3-D on the fly and combine that with prerendered images. That's probably the way we're going to go, and you see other games doing the same thing.

continued

Q: You talk about 3-D engines. What's your take on QuickDraw 3D?

A: I don't know too much about it, but there's one thing I understand. It's going to set an architecture for displaying real-time 3-D graphics for people to build around. I think that's a step in the right direction, because QuickTime is like an architecture for time-based media. QuickTime has been readily adopted by hundreds of developers. Since QuickTime came out, there's been a lot of really interesting and innovative tools that have been released that utilize its file format. You look at stuff like QuickTime VR, tools like COSA's After Effects, and other special effects programs. Look at Adobe Premiere. The QuickTime architecture allowed those tools to happen.

If Apple does a good job putting together a 3-D architecture that's crossplatform and also maybe gets into the consumer market, such as the PlayStations and the Saturn, then I think that's really good, if that format allows more portability between applications. Because right now with 3-D, when you're working in an application, you can get models from one place to another, but you can't get data information like animation, key frames, texture maps, stuff like that. So there's no way to go, unlike QuickTime, which allows graphics and time-based media to move from application to application. And it's supported the same way in each application. If they could do something like that with 3-D, where you can move 3-D files from Soft Mod, Electric Image, so forth and so on, and retain all the data in those files, that would be really, really, good. That would make a lot of people's lives a lot easier. It would create an architecture people could develop around and set standards around.

Q: What do you think of Simon & Schuster's Star Trek Interactive Manual *CD-ROM?*

A: It's pretty cool. I looked at some of the files and how they are doing stuff, and it's pretty interesting. It's interesting the way they are created, and especially if some of the 3-D applications support it. That was all done with photography. They have these tools, and one's called the Stitcher because it stitches all the images together to make one big bitmap. Hopefully, some of the 3-D applications will support that capability, so you can build a model, build an environment, and export that environment in a format that would be already stitched together for you.

Right now the performance level that you need to make that stuff run adequately is pretty high. Most of the stakes are higher because we're not making games for Mac LCIIs anymore. Technology is quickly putting PowerPCs and Pentiums into desktop computers. Since vendors ship so many of these systems, it's getting to the point where you can develop with those machines and not sacrifice too much of your market. QuickTime VR requires a lot of horsepower, which is its down side, but it's pretty cool.

The only thing that is crazy right now is that insane licensing and royalty structure. It's at a price point where developers are going to say "Well, I don't know

if I'm going to do this." Microsoft has its own version of the technology that doesn't appear to be—from what I've heard—as good, but it's free. For Apple, in the case of QuickTime, they charge you a royalty of $1500 a year for a title. It doesn't matter how many units you sell, you just paid the license straight out. But if they're charging 80 cents a disc for QuickTime VR, or even 25 cents a disc, nobody would be using it. When you're just barely making money, 25 cents a disc is a lot of money for a developer. Especially when you're only making $2 a disc.

Q: They changed that recently. The licensing fee was reduced for commercial use to between 3 to 8 cents per unit. There's no licensing fee for noncommerical use. It has been as high as 80 cents per unit. Also, the price of the QuickTime VR SDK was reduced from $1995 to $495. I think a lot of that had to do with the simple existence of the Microsoft Surround Video product, which had no licensing fees. Apple realized that it needed to back off on the fees or it will lose developers.

A: I think the Mac could be viable; it's always had innovative software. And QuickTime is so much better than Video for Windows. I think they shouldn't look at this as a money-making operation. They should charge a licensing fee that allows them to keep it viable and not make it a big profit center. Because it's what's going to allow people to develop on the Mac. It enables developers to make stuff for the Mac market, and make stuff for any of the player technologies Apple's working on, such as Pippin.

But you know, if they're going to charge a lot of money.... That's 80 cents a disk. If I look at selling 40,000 units, how much does that cost me, and could I develop something like this myself? Or, if I did it with the Microsoft product, I would make X amount more money, and how much better would it be? Would I sell more units if I made it with QuickTime, and if I sold more units, would those units be more than the difference in the licensing fees? It gets dicey at best. The technology is pretty neat—I'm looking forward to trying to get out on a project.

Q: What do you think of the game market in general now? Obviously stuff like Marathon, Doom II, Iron Helix, and the JourneyMan project have kind of raised the bar on expectations. How do you see the game market, and is there a place for an aspiring Mac programmer?

A: On the Mac side, its more of a like a boutique type of thing where people have been making smaller, experimental projects like the JourneyMan project and Freak show and stuff like that. But the bar has definitely been raised. People just can't put out schlock. In terms of the game market, you've got to go to the PC side to make any money. The PC side has the very sophisticated gaming community, and there's more games out on the PC, like Doom or Dark Forces from LucasArts. Technologically, these engines are really sophisticated.

continued

Right now, though, a lot of the content is kind of dumb. Doom is where you see something and you whack it. I mean, it's cool and it's fun, but there's not a lot of story going on. And I don't necessarily know if that's a bad thing, but I think games, in order to move beyond just being games to being more entertainment, are going to need to have a little more depth to the content. Not to say that you can't whack things and blow stuff up, but you're going to need something else to hold your attention. Because those shoot-'em-up games are just going to get tiring. It's okay. You can appeal to that 15- and 20-year-old crowd. But I think to move beyond that, you're to going to have to be a little more sophisticated. Iron Helix was essentially pretty simplistic in its concept. But there was some strategy involved in it. I think some of these other games are going to get a little more sophisticated in content and also in terms of execution. There's a lot of junk out there right now, and I think that stuff is just going to die because it costs a lot of money to produce these things. And even selling 100,000 units, you're just breaking even.

Q: What about shareware? Do you think there's any point in somebody going that route, or is it all in the hands of bigger outfits such as yourself and LucasArts?

A: Well, we're not LucasArts. We're not on that scale at all. You look at something that Id Software did, that started out as a Wolfenstein. That was distributed on the net free. It was a statement: "Check it out, here's some free stuff, and if you want other levels you have to buy it." That was an interesting way to get that thing on the market, to just give it away, and give people a taste of how cool it was, and then sell a real deal. That worked out really well for those guys—they sold a million units.

But in terms of shareware, I don't know. It's one thing to market a product by giving a portion of it away over the net, or however you do it, in hopes of building a buzz and then sell a real product afterwards. But you're going to need a distribution network. You're going to need somebody to get your product on the shelves, because with more and more products coming out, a person has seven feet of shelf space in his store, and he's only going to put X amount of titles in that seven feet. If your product isn't there, it's not there and no one's going to buy it. It doesn't matter how great it is. You can look at some titles that have just been critically acclaimed but have sold terribly because they haven't gotten the penetration in the market. You need to be distributed in the superstores to make money.

And what that means is that you don't necessarily have to be a big guy, you don't have to be huge, but you need to be affiliated with somebody, whether it's an affiliate label deal like the guys at Pop Rocket are doing with Electronic Arts, whether it's a straight publishing deal, or whatever. You need to be hooked up with somebody. When you get into that business, it's a "who knows who" and "who's the biggest guy on the block" situation. You're up against Virgin and Time-Warner and these huge game companies, and you've got to have some in-roads into the distribution network.

Chapter 10
Networking

In the last few years, networking has been one of the strongest areas of growth within the computer industry. As more networks are put into place, game programmers have begun to take advantage of the possibilities offered by networks. Games such as Spectre, Bolo, MazeWars, Marathon, and Doom have demonstrated that networking not only enhances the playability of existing games, but also makes possible games of unprecedented imagination and just plain fun.

In this chapter, we'll look at the protocols and technology you need to add networking capabilities to your game or application. You will be pleased and perhaps even surprised at how easy it is to add such features to your game.

As a "real live" example of a network game, on the CD-ROM accompanying this book, you'll find a copy of the source code for a game called Novel Netwar. This game demonstrates the basic ideas discussed on the pages that follow, and also shows how to implement these ideas in a practical application. The game was written using Metrowerk's CodeWarrior C compiler, but the code should compile under Symantec's Think C with only minor modifications.

Figure 10.1 shows a screen snapshot of the Novel Netwar game. As you can see, it's a classic arcade-style game in which players pilot spacecraft around a sun while trying to blow each other up with photon torpedoes. The game is fairly basic as I write this, though I'll keep hacking at it as the publication deadline looms nearer, so check out the source code on the CD-ROM for any last-minute enhancements to the game. The source code is also available from my home page, `http://www.io.org/~rrwood`, and the most recent version will always be available there.

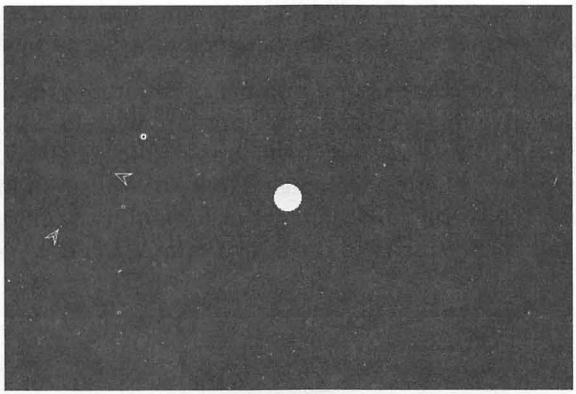


Figure 10.1 *Novel Netwar.*

In this chapter, we'll look specifically at the following networking protocols:

- AppleTalk, the networking programming protocol "built in" on all Macintoshes
- Serial connections via the Printer or Modem port
- Outland's Connection Library
- Open Transport, Apple's new "standards-based" network programming interface

So, without further ado, let's get on to the good stuff...

AppleTalk

The single most common networking protocol available for the Macintosh is unquestionably AppleTalk. There are literally thousands of AppleTalk networks connecting hundreds of thousands of Macintoshes all over the world. In fact, AppleTalk has proven so popular that its use has spread to other platforms, such as the 80x86 PC and Unix-based systems. AppleTalk's popularity has increased even more in the past few years with the introduction of AppleTalk Remote Access (ARA), a low-level link access protocol that allows AppleTalk to run over a serial connection such as a modem, and the introduction of similar low-level link access protocols to support Ethernet, Token Ring, and Fiber Distributed Data Interface networks.

Note

Note that Ethernet, Token Ring, FDDI, and ARA are integrated transparently by the AppleTalk Manager. As a programmer, you do not have to worry about the low-level link access protocol; you just program to the AppleTalk Manager interface and everything is taken care of automatically.

The reason behind AppleTalk's popularity with the average user is undoubtedly its ease of use: Connecting to an AppleTalk network can be as simple as plugging a LocalTalk cable into the Macintosh's printer port and turning on AppleTalk from within the Chooser. The reasons behind AppleTalk's popularity with programmers include the range of services it provides and its relatively straightforward applications programming interface (API).

The AppleTalk Manager API encompasses a number of different areas, not all of which are especially relevant or useful to the average games programmer. In this section, we'll look at the AppleTalk Name Binding Protocol (NBP), the AppleTalk Transaction Protocol (ATP), and the AppleTalk Datagram Delivery Protocol (DDP). For information about the other protocols available under the AppleTalk Manager or for a more detailed reference to the protocols discussed in this section, refer to the book *Inside Macintosh: Networking*

(ISBN: 0-201-62269-6), published by Addison Wesley or available via anonymous FTP from `ftp.info.apple.com` as `Apple.Support.Area/Developer_Services/Technical_Documentation/Inside_Macintosh/Inside_Macintosh/Networking/Networking.sit.hqx` (the accompanying index is in the same directory but is called `Networking.idx.sit.hqx`). The electronic version of the book available from `ftp.info.apple.com` is in DocViewer format, so if you're going to FTP it, make sure you have a copy of Apple's DocViewer or else grab a copy from `Apple.Support.Area/Developer_Services/Utilities/Apple_DocViewer__v1.1` while you're there.

Now before we get into the nitty-gritty details of the AppleTalk API, we first need to clarify a few fundamentals and a lot of terminology. As always, we look at things from the perspective of a games programmer, so we'll omit or at least simplify most of the more esoteric aspects of AppleTalk in particular and networking theory in general.

AppleTalk Networking Concepts

The logical view of an AppleTalk network is shown in figure 10.2.

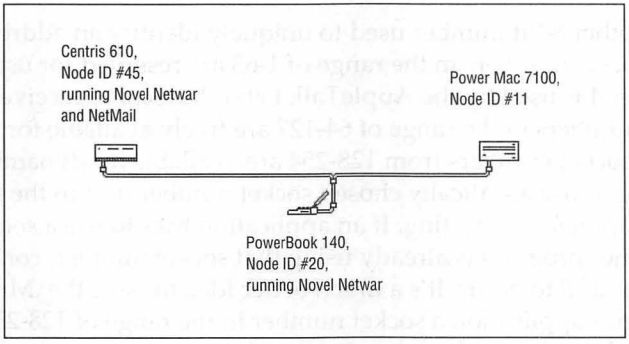


Figure 10.2 *Logical view of an AppleTalk network.*

If you examine figure 10.2 carefully, you'll note that each computer connected to the network is uniquely identified by a number which is referred to as its *node ID*. For example, the Centris 610 has a node ID of 45, the PowerBook 140 has a node ID of 20, and the Power Mac 7100 has a node ID of 11.

A computer's node ID is determined dynamically when the AppleTalk driver (.MPP) is initialized. At this time, the .MPP driver looks up the node ID that was used the last time it connected to the network, and broadcasts a network message indicating its intent to use the node ID again. If another computer sends a reply indicating that it is currently using that node ID, the .MPP driver randomly selects another ID and again broadcasts a network message indicating its intent to use the new node ID. This process is repeated until no protests from other machines are received.

Each node ID is an 8-bit number in the range of 1-254, with node IDs from 1-127 available for user nodes and node IDs from 128-254 reserved for server nodes. The distinction between user nodes and server nodes is made under the assumption that a user-node computer is likely to start up and shut down often, whereas a server-node computer (such as the fileserver running on the Power Mac 7100) is likely to remain up and running most of the time. The .MPP driver waits longer for replies when it tries to register a node ID on the server range, so there is less chance that two machines accidentally register on the network with the same node ID (which would obviously be a Bad Thing—sort of like the telephone company assigning the same phone number to two different people).

Messages can be sent from one machine to another on the network using the node IDs as destination addresses. However, if only node IDs are used to identify network addresses, problems would arise if more than one network-aware program were running on each computer. For example, the Centris 610 in figure 10.2 is running a copy of Novel Netwar and a copy of NetMail (a hypothetical electronic mail application). When an incoming network message is received by the Centris 610, the .MPP driver needs to have some way of determining which application should receive the message—this is where socket numbers come in to play.

A socket number is another 8-bit number used to uniquely identify an address on an AppleTalk network. Socket numbers in the range of 1-63 are reserved for use by Apple; for example, socket number 4 is used by the AppleTalk Echo Protocol to receive and reply to echo requests. Socket numbers in the range of 64-127 are freely available for use in development and testing, and socket numbers from 128-254 are available for dynamic assignment. Note that it's a bad idea to use a statically chosen socket number (i.e., in the range of 64-127) for anything but development and testing. If an application tries to use a socket number in this range and some other program is already using that socket number, conflicts and system crashes are obviously bound to occur. It's a much better idea to have the .MPP driver dynamically allocate your application a socket number in the range of 128-254 and thus avoid any potential conflict over preferred socket numbers.

Combining a node ID and a socket number makes it simple to uniquely identify the destination address and application for a network message. For example, the Novel Netwar game running on the Centris 610 is identified as 45:64, since it is located at node ID number 45 and socket number 64. Likewise, the NetMail program running on the same machine is identified as 45:200, and the Novel Netwar game running on the PowerBook 140 is 20:64.

All of the above is fairly straightforward within the context of the network in figure 10.2; however, if we now consider the possibility of an extended AppleTalk network such as the one shown in figure 10.3, it quickly becomes apparent that we have to consider one more level of network addressing.

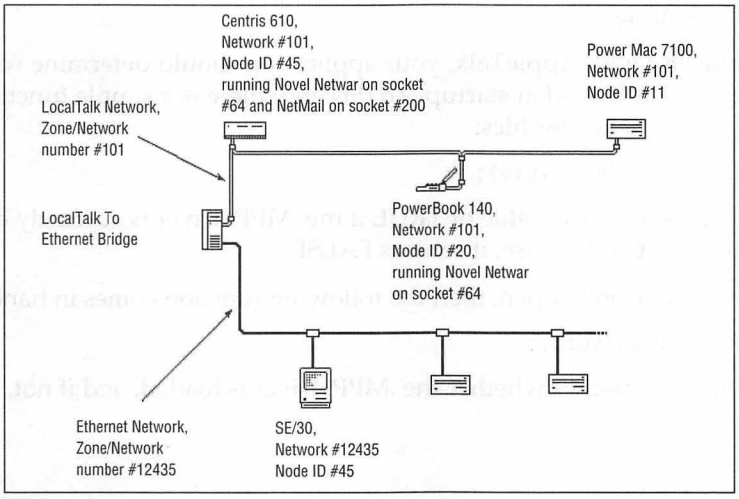


Figure 10.3 *An extended AppleTalk network.*

If you examine figure 10.3 carefully, you'll note that one of the new machines has the same node ID as the Centris 610 in the original network. Since the SE/30 in the new network is also registered as node ID number 45, it would seem that the two machines no longer have unique network addresses, and therefore it has become impossible to specifically address just the Centris 610 or the SE/30. If you look at the figure a little more closely, however, you'll see that the solution to this problem is fairly simple: The new machines belong to a network with a network number, 12435, that is different from the original network's network number, 101. The 16-bit network number of each network forms the final level of network addressing for AppleTalk.

An extended network or "AppleTalk Internet" (not to be confused with *the* Internet) such as the one shown in figure 10.3 is connected by some sort of a router or bridge which listens to network traffic from both networks, forwarding data packets from one network to the other as appropriate. Each of the AppleTalk networks in an AppleTalk Internet is identified by a zone name, which is the human-readable name listed in the Chooser or other network selection dialog. Also, a network number of zero is interpreted by the .MPP driver as a reference to the local network.

Bearing this in mind, it's now obvious that a complete, unique AppleTalk network address is comprised of a 16-bit network number, an 8-bit node ID, and an 8-bit socket number. For example, the network address of the Novel Netwar game running on the Centris 610 is 101:45:64 (network number 101, node ID number 45, and socket number 64), and the corresponding address of the Novel Netwar game running on the SE/30 is 12435:45:64. And now that we understand the general concepts of AppleTalk addressing, it's time to examine the API that allows you to register your program as an AppleTalk network entity.

Opening the AppleTalk .MPP Driver

Before doing anything else with AppleTalk, your application should determine whether the AppleTalk .MPP driver was loaded at startup. To this end, there is a simple function defined in the AppleTalk.h universal header files:

```
extern pascal Boolean IsMPPOpen(void);
```

The `IsMPPOpen()` function returns a value of TRUE if the .MPP driver is currently open and loaded into the system heap; otherwise, it returns FALSE.

If the .MPP driver is not currently open, then the following function comes in handy:

```
extern pascal OSErr MPPOpen(void);
```

The `MPPOpen()` function determines whether the .MPP driver is loaded, and if not, it loads and initializes the driver.

Note

You can also use the Device Manager `OpenDriver()` routine to open the .MPP driver. In fact, you must make an extended `Open()` call to the Device Manager if you want to allocate a node ID in the server range 128-254. Refer to Inside Macintosh: Networking, *pp. 2.7-2.8 for more information about this.*

The Novel Netwar game uses the preceding functions in its `appletalkingStartup()` routine as follows:

```
// Excerpted from the appletalking.c file…

OSErr     errCode;


   if (IsMPPOpen() == false)
   {
      errCode = MPPOpen();

      if (errCode != noErr)
      {
         sprintf(errorMessage,"appletalkingStartup: MPPOpen() error %d, can't open ➥
the AppleTalk MPP driver!",(int) errCode);

         ErrorAlert(errorMessage);

         goto EXITPOINT;
      }
   }
```

This source code shows that the `appletalkingStartup()` routine uses `IsMPPOpen()` to check whether the .MPP driver has already been loaded. If .MPP has not been loaded, a call to `MPPOpen()` is made, and the returned error code is checked. If an error occurs in trying to open .MPP, the global character string `errorMessage` is set and then displayed using my `ErrorAlert()` function, and then a jump is made to `appletalkingStartup`'s exit point. Incidentally, I prefer to handle errors in this fashion, despite the taboo associated with `goto`s in structured languages.

Note

Speaking of structured languages and taboos, keep in mind the following:

"C... all the power of assembly language programming with all the ease of assembly language programming."

It's also worth noting that a quick call to `Gestalt()` allows you to determine if AppleTalk is installed, and if so, what version of AppleTalk is installed. This is demonstrated by the following excerpt from the `globalsInit()` routine in globals.c:

```
// Excerpted from globals.c

long      theAppleTalkVersion;
char      hasAppleTalk;


hasAppleTalk = false;

if (Gestalt(gestaltAppleTalkVersion,&theAppleTalkVersion) == noErr)
{
   if (theAppleTalkVersion != 0)
   {
      hasAppleTalk = true;
   }
}
```

Determining Your Network Address

Once the .MPP driver is open, you next need to look up the network number and node ID of your machine. This is most easily accomplished by calling `GetNodeAddress()`, which is declared as follows:

```
extern pascal OSErr GetNodeAddress(short *myNode, short *myNet);
```

The `GetNodeAddress()` function returns the network number and node ID of the machine in `myNet` and `myNode`, respectively. For a nonextended LocalTalk network, the network number is always zero.

The Novel Netwar game uses the `GetNodeAddress()` function in its `appletalkingStartup()` routine as follows:

```
// Excerpted from appletalking.c

OSErr     errCode;
short     myNodeNum,myNetNum;

  errCode = GetNodeAddress(&myNodeNum,&myNetNum);

  if (errCode != noErr)
  {
    sprintf(errorMessage,"appletalkingStartup: GetNodeAddress() error %d, can't ➥
look up AppleTalk node/net address!",(int) errCode);

    ErrorAlert(errorMessage);

    goto EXITPOINT;
  }
```

Note that `GetNodeAddress()` returns your network number and node ID, but not your socket number. The socket number is determined when you open it, as described later in this chapter when we discuss ATP and DDP.

Self-Sending

Normally, AppleTalk is configured so that network messages are sent only to other nodes, and not to other programs running on the same machine; however, you may wish to allow AppleTalk-based communication between two applications running on the same machine, which is when the `PSetSelfSend()` call is useful:

```
extern pascal OSErr PSetSelfSend(MPPPBPtr thePBptr, Boolean async);
```

The `PSetSelfSend()` function allows you turn on or off the self-send feature of AppleTalk. When self-sending is on, network messages can be sent to and received from other sockets on the same node; when self-sending is off, this intranode delivery capability is disabled.

`PSetSelfSend()` takes a pointer to an `MPPParamBlock` as its first parameter, and a Boolean flag as its second parameter. The `MPPParamBlock` is a complicated, multipurpose structure that we'll be seeing a lot of in the pages that follow, but for `PSetSelfSend()` we're mostly interested in its `newSelfFlag` and `oldSelfFlag` fields. The former field is used to indicate the desired new status of self-sending, and the latter field returns the previous status of self-sending. If I use the notation of *Inside Macintosh*, these fields are defined as:

→	`newSelfFlag`	Byte	A flag that indicates the desired new status of intranode delivery
←	`oldSelfFlag`	Byte	A flag that reports the previous status of intranode delivery

Note
Inside Macintosh *uses the convention that a right-pointing arrow, "→", indicates an input parameter, a left-pointing arrow, "←", indicates an output parameter or returned value, and a double-pointed arrow, "↔", indicates a parameter used for both input and output.*

The `asynch` parameter is used to indicate whether the call should be done synchronously or asynchronously. If you pass in a value of FALSE for this parameter, the call is made synchronously, so everything is completed before the function completes and control returns to your program. If you pass in a value of TRUE for this function, the call returns immediately and the actual work of setting the self-send flag is done more or less concurrently as your program continues execution. We'll talk more about asynchronous routines later in this chapter, but for now, just accept it on faith that there's really not much point to asynchronously setting the self-send flag.

The Novel Netwar game doesn't use the `PSetSelfSend()` function, but you may wish to use it yourself someday if, for example, you wish to test an AppleTalk game on one machine running multiple copies of the program. `PSetSelfSend()` can be used as shown below:

```
MPPParamBlock     myMPPPB;

   myMPPPB.SETSELF.newSelfFlag = true;

   errCode = PSetSelfSend(&myMPPPB,false);

   if (errCode != noErr)
   {
      sprintf(errorMessage,"PSetSelfSend() error %d",(int) errCode);

      ErrorAlert(errorMessage);
   }
```

AppleTalk Name Binding Protocol Services

Now that we've seen how to open the AppleTalk .MPP driver and determine our own network address, the next logical step is to see how we can determine the network address of another computer on the network. This is fairly easy to do using the services provided by the AppleTalk Name Binding Protocol (NBP). Basically, NBP allows you to register your application's presence on the network and also look up other network entities. If you've ever used the Chooser to select a networked LaserWriter or to connect to an AppleShare fileserver, you've used NBP's services.

NBP is not essential to the sending and receiving of network messages; all you really need to know to send an AppleTalk message is the network number, node ID, and socket number of the message's destination. The advantage of NBP is that it substitutes meaningful names in place of the obscure numbers used by the lowest levels of AppleTalk. For example, the copy of Novel Netwar running on the Centris 610 in figure 10.3 could be registered with NBP as Roy:NetWar@LocalTalkNet, which is much more meaningful to us humans than a network address of 101:45:64.

An NBP address or entity name is made up of three parts: an object, a type, and a zone name. Each part of the name is a string of characters which can be no longer than 31 characters. The zone name is the name of the entity's AppleTalk zone; the type identifies entities of a common nature, such as copies of a particular game or a group of LaserWriters; the object name differentiates the members of a type of entity, such as the individual players playing a game. A complete NBP entity name is usually written as object:type@zone, which is in the reverse order of the network:node:socket form used for numeric addresses.

NBP supports three wildcard characters in NBP name descriptors: An "=" represents all possible values for an object or type, a "≈" (Option-X) represents zero or more characters in an object or type, and a * represents the local zone name.

Registering as an NBP Entity

To register as an NBP entity, a program must create a special data structure called a Names Table Entry and then pass a pointer to this data structure to the `PRegisterName()` function. For example, the following code snippet shows how a program could register as Roy:NetWar@LocalTalkNet:

```
NamesTableEntry   myNamesTableEntry;
MPPParamBlock  myMPPPB;
short        errCode;


   NBPSetNTE((Ptr) &myNamesTableEntry, "\pRoy", "\pNetWar", "\pLocalTalkNet", 64);

   myMPPPB.NBP.ioCompletion = nil;
   myMPPPB.NBP.interval = 0x07;
   myMPPPB.NBP.count = 0x03;
   myMPPPB.NBP.nbpPtrs.entityPtr = (Ptr) &myNamesTableEntry;
   myMPPPB.NBP.parm.verifyFlag = true;

   errCode = PRegisterName(&myMPPPB,true);
```

The `NBPSetNTE()` procedure sets the `objectStr`, `typeStr`, and `zoneStr` fields of a `NamesTableEntry` data structure, which would be difficult to do otherwise because they are stored in a complex format (packed). The procedure is declared as shown below. Note that the string parameters must all be Pascal-style strings.

```
extern pascal void NBPSetNTE(Ptr ntePtr, ConstStr32Param nbpObject,➥
ConstStr32Param nbpType, ConstStr32Param nbpZone, short socket);
```

`ntePtr` is a pointer to a `NamesTableEntry` data structure.

`nbpObject` is the object part of the NBP name.

`nbpType` is the type part of the NBP name.

`nbpZone` is the zone part of the NBP name. This is normally just set to *, which is a special wildcard character that NBP interprets as a reference to the local zone.

`socket` is the socket number of the entity.

The `PRegisterName()` function takes the NBP name from the `NamesTableEntry` data structure and adds it to a list of names maintained internally by .MPP. Later, when other computers send out an NBP lookup request, .MPP will automatically respond with the appropriate names from its cached list. `PRegisterName()` is declared as shown below. As with `PSetSelf()`, the function's first parameter is a pointer to `MPPParamBlock`, the relevant fields of which are also described.


```
extern pascal OSErr PRegisterName(MPPPBPtr thePBptr, Boolean async);
```

Parameters:

→	`ioCompletion`	ProcPtr	For asynchronous calls to `PRegisterName()`, this is a pointer to a completion routine that is called when name registration is complete; for no completion routine call, this should be NIL.
←	`ioResult`	OSErr	The return code or function result.
→	`ioRefNum`	Short	The .MPP driver reference number; this is automatically set.
→	`csCode`	Short	This is automatically set to `registername`.
→	`interval`	Byte	The retry interval measured in 8-Tick units (see what follows).
→	`count`	Byte	The retry count (see what follows).
→	`entityPtr`	Ptr	A pointer to a Names Table Entry.
→	`verifyFlag`	Byte	A flag to indicate whether NBP should attempt to verify the uniqueness of the supplied name.

Note

The Names Table Entry pointed to by `entityPtr` is used by .MPP as long as the NBP name remains registered. You should either allocate memory for this using `NewPtr` or declare it as a static or global variable. Do not use a temporary or local variable for this or your program and system will crash.

`PRegisterName()` takes the supplied NBP name and broadcasts a request to use that name. If another network entity is already using the same name, then the other entity responds with a denial message, and `PRegisterName()` returns an error. The `interval` and `count` parameters are used to indicate, respectively, how often and how many times `PRegisterName()` should attempt to verify the uniqueness of the NBP name. This verification process can be disabled by setting the `verifyFlag` to FALSE.

Since the validation of uniqueness can take a few seconds, you may wish to call `PRegisterName()` asynchronously and allow the impatient user a chance to cancel the operation. If your program calls `PRegisterName()` with a second parameter of TRUE, the function returns almost immediately and verification proceeds concurrently. If you call `PRegisterName()` in this manner, `ioResult` is set to a value of 1 until the verification is complete. Your program can poll `ioResult` while it is doing something else, such as watching to see if the user presses a cancel button in a dialog. This is what Novel Netwar does, as shown by the following snippet:

```
// from appletalking.c

OSErr          errCode;
MPPParamBlock  myMPPPB,myCancelMPPPB;
char           tempString[256];
DialogPtr      theDPtr,tempDPtr;
short          itemHit,type;
handle         theItem;
Rect           tempRect;
EventRecord    theEvent;
handle         theMachineName;


// Try to look up machine name as shown in Chooser dialog,
// otherwise make a unique name from net/node/socket numbers

   theMachineName = GetResource('STR ',-16413);

   if (theMachineName != nil && *theMachineName != nil && **theMachineName != 0)
   {
      HLock(theMachineName);
      mystrncpy((char *) *theMachineName,tempString,255);
      HUnlock(theMachineName);
      ReleaseResource(theMachineName);
   }

   else
   {
      sprintf(tempString,"Netwar@%d:%d:%d",(int) myNetNum,(int) myNodeNum,(int) ➥
myATPSocketNum);
   }
```

```
// Allow user to accept default NBP name or supply his/her own choice

   errCode = GetInput("Choose your network name:",tempString);

   if (errCode != noErr)
   {
      errCode = NONETNAME;

      sprintf(errorMessage,"appletalkingStartup: no network name specified!");

      ErrorAlert(errorMessage);

      goto EXITPOINT;
   }


// Save a permanent Pascal-string copy of the name in myNBPName

   mystrncpy((char *) &myNBPName,tempString,sizeof(myNBPName));
   CtoPstr((char *) myNBPName);


// Allocate storage for a NamesTableEntry

   myNamesTableEntry = NewPtr(sizeof(NamesTableEntry));

   errCode = MemError();

   if (errCode != noErr || myNamesTableEntry == nil)
   {
      sprintf(errorMessage,"appletalkingStartup: NewPtr() error %d, cannot register ➥
NBP name!",(int) errCode);

      ErrorAlert(errorMessage);

      goto EXITPOINT;
   }


// Show a dialog with a Cancel button

   theDPtr = GetNewDialog(INFOCANCELDLOG,nil,(WindowPtr) -1L);

   if (theDPtr != nil)
   {
      SetPort(theDPtr);

      ((DialogPeek) theDPtr)->aDefItem = -1;

      GetDItem(theDPtr, 2, &type, &theItem, &tempRect);
      SetIText(theItem,"\pPlease wait-- registering network name....");
```

```
    CenterWindow(theDPtr);
    ShowWindow(theDPtr);
    DrawDialog(theDPtr);
  }


// Set up the NamesTableEntry; myATPSocketNum has already been set --
// see the section on sending and receiving messages using ATP below

  NBPSetNTE(myNamesTableEntry, myNBPName, myNBPNetType, "\p*", myATPSocketNum);


// And register the entity

  myMPPPB.NBP.ioCompletion = nil;
  myMPPPB.NBP.interval = 0x0F;
  myMPPPB.NBP.count = 0x03;
  myMPPPB.NBP.nbpPtrs.entityPtr = myNamesTableEntry;
  myMPPPB.NBP.parm.verifyFlag = true;

  errCode = PRegisterName(&myMPPPB,true);

  if (errCode != noErr)
  {
    sprintf(errorMessage,"appletalkingStartup: PRegisterName() error %d at ➥
initial call, can't register NBP name!",(int) errCode);

    ErrorAlert(errorMessage);

    goto EXITPOINT;
  }


// Wait for PRegisterName() to complete or user to press Cancel

  do
  {
    itemHit = 0;

    if (GetNextEvent(everyEvent,&theEvent) == true && IsDialogEvent(&theEvent) == ➥
true)
    {
      // Call my ProcessDialogCmdKey() to handle ESC, etc.

      if (ProcessDialogCmdKey(theDPtr,&theEvent,&itemHit) == true)
      {

      }


      else if (DialogSelect(&theEvent,&tempDPtr,&itemHit) == true)
      {
```

```
            if (tempDPtr != theDPtr)
               itemHit = 0;
         }
         else
         {
            itemHit = 0;
         }
      }

   } while (itemHit != 1 && myMPPPB.NBP.ioResult == 1);


// Check if user pressed Cancel button

   if (itemHit == 1)
   {
      myCancelMPPPB.NBPKILL.nKillQEl = (Ptr) &myMPPPB;

      PKillNBP(&myCancelMPPPB,false);
   }
   else
   {
      if (myMPPPB.NBP.ioResult == noErr)
      {
         nbpRegistered = true;
      }
      else
      {
         sprintf(errorMessage,"appletalkingStartup: PRegisterName() error %d after ➥
asynch completion, can't register NBP name!",(int) errCode);

         ErrorAlert(errorMessage);

         goto EXITPOINT;
      }
   }
```

In the code listing above, you'll notice that the `PRegisterName()` is canceled if the user presses the Cancel button in the dialog. This is accomplished by a call to `PKillNBP()`, which is declared as:

```
extern pascal OSErr PKillNBP(MPPPBPtr thePBptr, Boolean async);
```

Parameters:

→	`nKillQEl`	Ptr	A pointer to the `MPPParamBlock` for the asynch NBP call to kill.

The only other thing you need to remember about registering as an NBP entity is that you must unregister before your program exits. Recall that .MPP uses the `NamesTableEntry` you pass in to `PRegisterName()` until you tell it otherwise. If your application exits without removing this entry from .MPP's list of known NBP names, the system may crash. Removing the NBP entity name is perfectly straightforward:

```
NamesTableEntry   tempNamesTableEntry;
MPPParamBlock     myMPPPB;
short             errCode;

  NBPSetEntity((Ptr) &tempNamesTableEntry, myNBPName, myNBPNetType, "\p*");

  myMPPPB.NBP.nbpPtrs.entityPtr = (Ptr) &tempNamesTableEntry;

  errCode = PRemoveName(&myMPPPB,false);
```

`PRemoveName()` is declared as shown below:

```
extern pascal OSErr PRemoveName(MPPPBPtr thePBptr, Boolean async);
```

The most important parameter to `PRemoveName()` is the `entityPtr` parameter, which points to a `NamesTableEntry` structure set up by `NBPSetEntity()`. Do not use `NBPSetNTE()` to set up this structure—the format it creates is incompatible with `PRemoveName()`!

Looking Up Other NBP Entities

Once you've registered yourself as an NBP entity, you'll probably want to look up the other NBP entities on the network. Naturally, there are simple routines for doing this, as illustrated by the following code:

```
OSErr          errCode;
MPPParamBlock  myMPPPB;
AddrBlock      theEntityAddress;
Ptr            nbpNamesBuffer;
EntityName     theEntityName;
int            i;


// Allocate storage for NBP lookup

  nbpNamesBuffer = NewPtr(MAXNUMNBPNAMES * NBPNAMESIZE);

  errCode = MemError();

  if (errCode != noErr || nbpNamesBuffer == nil)
  {
    sprintf(errorMessage,"NewPtr() error %d, can't allocate nbpNamesBuffer for ➥
use in looking up NBP names!",(int) errCode);

    goto EXITPOINT;
  }
```

```
// Set the match criteria for the NBP lookup

   NBPSetEntity((Ptr) &theEntityName,"\p=",myNBPNetType,"\p*");


// Set the MPPParamBlock parameters

   myMPPPB.NBP.ioCompletion = nil;
   myMPPPB.NBP.interval = 0x0F;
   myMPPPB.NBP.count = 0x01;
   myMPPPB.NBP.nbpPtrs.entityPtr = (Ptr) &theEntityName;
   myMPPPB.NBP.parm.Lookup.retBuffPtr = nbpNamesBuffer;
   myMPPPB.NBP.parm.Lookup.retBuffSize = MAXNUMNBPNAMES * NBPNAMESIZE;
   myMPPPB.NBP.parm.Lookup.maxToGet = MAXNUMNBPNAMES;


// Look up the NBP names

   errCode = PLookupName(&myMPPPB,false);

   if (errCode != noErr)
   {
      sprintf(errorMessage,"PLookupName() error %d, can't look up enemy player's ➥
NBP names!",(int) errCode);

      goto EXITPOINT;
   }


// Extract the names from the returned data

   for (i=0;i < myMPPPB.NBP.parm.Lookup.numGotten;i++)
   {
      errCode = NBPExtract(nbpNamesBuffer, myMPPPB.NBP.parm.Lookup.numGotten, i+1, ➥
&theEntityName, &theEntityAddress);

      if (errCode != noErr)
      {
         sprintf(errorMessage,"NBPExtract() error %d, can't extract an NBP ➥
name!",(int) errCode);

         goto EXITPOINT;
      }
   }
```

The first routine used in looking up NBP names is `PLookupName()`, which is defined as:

```
extern pascal OSErr PLookupName(MPPPBPtr thePBptr, Boolean async);
```

Parameters:

→	`ioCompletion`	ProcPtr	A completion routine pointer for asynchronous calls.
←	`ioResult`	OSErr	The return code or function result.
→	`ioRefNum`	Short	Automatically set to the .MPP driver reference number.
→	`csCode`	Short	Automatically set to `lookupName`.
→	`interval`	Byte	The retry interval.
→	`count`	Byte	The retry count.
→	`entityPtr`	Ptr	A pointer to an entity name.
→	`retBuffPtr`	Ptr	A pointer to a buffer for returned NBP names.
→	`retBuffSize`	Short	The length of return buffer.
→	`maxToGet`	Short	The maximum number of names to return.
←	`numGotten`	Short	The actual number of NBP names returned.

`PLookupName()` sends out a request to other computers on the network, asking for replies that match the form of the names specified by `entityPtr`. The replies received are stored in the return buffer, and the number of names returned is indicated in `numGotten`. These names are stored in a packed form and must be extracted using `NBPExtract()`.

Note

*`PLookupName()` recognizes the three NBP wildcard characters in the `EntityName` descriptor passed in by `entityPtr`. Recall that = represents all possible values for an object or type, ≈ (Option-X) represents zero or more characters in an object or type, and * represents the local zone name.*

`NBPExtract()` is defined in this way:

```
extern pascal OSErr NBPExtract(Ptr theBuffer, short numInBuf, short whichOne, ➥
EntityName *abEntity, AddrBlock *address);
```

The parameters to `NBPExtract()` are described as follows:

`theBuffer` is the pointer to the buffer returned by `PLookupName()`.

`numInBuf` is the number of names in `theBuffer`.

`whichOne` is the index number of the name to extract.

`abEntity` is the extracted NBP name.

`address` is the net:node:socket of the extracted NBP name.

The most important part of the extracted data is the network address which is returned in a data structure of type `AddrBlock`:

```
struct AddrBlock {
   UInt16      aNet;
   UInt8       aNode;
   UInt8       aSocket;
};
```

Also, the object name is returned at the start of the `abEntity` structure and can be extracted like this:

```
char    playername[32];

   PtoCstr((unsigned char *) &theEntityName);
   mystrncpy(playerName,(char *) &theEntityName,sizeof(playername));
```


As with most AppleTalk routines, `PLookupName()` can be called asynchronously, which is what Novel Netwar does, thus allowing the user to cancel the procedure at any time. As with `PRegisterName()`, an outstanding `PLookupName()` call can be canceled using `PKillNBP()`.

```
MPPParamBlock     myCancelMPPPB;

   myCancelMPPPB.NBPKILL.nKillQEl = (Ptr) &myMPPPB;

   errCode = PKillNBP(&myCancelMPPPB,false);
```

Before calling `PKillNBP()` to cancel an outstanding `PLookupName()` request, you should probably check the `ioResult` field to see if the lookup request has already completed. As well, you should check the result code returned by `PKillNBP()` to see if it returns an error of `cbNotFound`, which indicates that the request has already been completed.

If you wish to look up NBP names for entities in other AppleTalk zones, you'll have to use the AppleTalk Zone Information Protocol (ZIP) routines to get a list of these other zones. Refer to *Inside Macintosh: Networking* for more information about ZIP.

Sending and Receiving Network Messages

Now that we've seen how to open the AppleTalk .MPP driver and determine our own address and the address of other network entities, it's time to examine the process of sending and receiving network messages. The two methods we'll look at are the AppleTalk Transaction Protocol (ATP) and the Datagram Delivery Protocol (DDP). Both protocols are used for sending and receiving network messages, though they each handle things a little differently and therefore are best suited to different types of applications.

AppleTalk Transaction Protocol (ATP)

The AppleTalk Transaction Protocol is a reliable delivery protocol, in that when you use it to send a message to another network entity, it will periodically resend the message until it receives confirmation of receipt from the destination node. This reliable delivery service entails a certain amount of overhead, which means that ATP is a little slower to use than a protocol like DDP, which does not retry message sends.

A complete ATP transaction consists of a request message being sent from a source entity to a destination entity, and a reply which is sent back from the destination entity to the source entity. A request message can consist of up to 578 bytes of data, and a reply can consist of up to eight 578-byte packets, or 4,624 bytes in total. If you want to transfer more than 4,624 bytes, you'll have to do it through multiple request-reply transactions.

As part of providing guaranteed delivery of data packets, ATP will resend a packet if it does not receive a reply within a specific timeframe. This means that it is possible for the destination entity to receive duplicate requests, which may be undesirable. To solve this problem, ATP supports at-least-once and exactly-once transactions. At-least-once transactions allow for the possibility of duplicate requests, while exactly-once transactions cause ATP to automatically screen out duplicate requests. Obviously there is extra processing involved in implementing exactly-once transactions, which means even slower system response; however, in some circumstances, this may be tolerable.

Opening an ATP Socket

Before your program can send or receive ATP data packets, it must open an ATP socket using `POpenATPSkt()` as follows:

```
OSErr          errCode;
ATPParamBlock  myATPPB;
UInt8          myATPSocketNum;


// Ask for a dynamically allocated socket number

   myATPPB.ATP.atpSocket = 0;

// Accept requests from anyone

   myATPPB.ATP.addrBlock.aNet = 0;
   myATPPB.ATP.addrBlock.aNode = 0;
   myATPPB.ATP.addrBlock.aSocket = 0;

   errCode = POpenATPSkt(&myATPPB,false);

   if (errCode == noErr)
   {
      myATPSocketNum = myATPPB.ATP.atpSocket;
   }
```

The `POpenATPSkt()` function is declared as:

```
extern pascal OSErr POpenATPSkt(ATPPBPtr thePBptr, Boolean async);
```

Parameters:

→	`ioCompletion`	ProcPtr	A completion routine pointer for asynchronous calls.
←	`ioResult`	OSErr	The return code or function result.
→	`csCode`	Short	Automatically set to `openATPSkt`.
↔	`atpSocket`	Byte	The desired ATP socket number or zero for a dynamically allocated socket number.
→	`addrBlock`	AddrBlock	The address from which ATP requests will be accepted; specify zeroes to accept requests from all networks, nodes, or sockets.

Of course, your program must also close the ATP socket before it exits. This is done using the `PCloseATPSkt()` function:

```
OSErr          errCode;
ATPParamBlock  myATPPB;

   myATPPB.ATP.atpSocket = myATPSocketNum;

   errCode = PCloseATPSkt(&myATPPB,false);
```

Sending an ATP Request

Once your program has opened an ATP socket and has used NBP to look up the address of another network entity, you can send a request to that other entity like this:

```
OSErr          errCode;
ATPParamBlock  myATPPB;
AddrBlock      otherPlayerAddr;
char           atpMessage[256],atpResponseBuffer[256];
short          numBDSBuffers;
BDSType        theBDS;


// Set up data structure for replies from other entity

   numBDSBuffers = BuildBDS((Ptr) atpResponseBuffer, (Ptr) &theBDS, 256);


// We can send anything we want as a message

   strcpy(atpMessage,"Hi there!");
```

```
// Set up the ATPParamBlock and then send the ATP request

  myATPPB.ATP.ioCompletion = nil;
  myATPPB.ATP.atpFlags = 0;

// Assume that otherPlayerAddr is set from a previous NBP lookup

  myATPPB.ATP.addrBlock.aNet = otherPlayerAddr.aNet;
  myATPPB.ATP.addrBlock.aNode = otherPlayerAddr.aNode;
  myATPPB.ATP.addrBlock.aSocket = otherPlayerAddr.aSocket;
  myATPPB.ATP.reqLength = strlen(atpMessage) + 1;
  myATPPB.ATP.reqPointer = (Ptr) atpMessage;
  myATPPB.OTH1.u.numOfBuffs = numBDSBuffers;
  myATPPB.ATP.bdsPointer = (Ptr) &theBDS;
  myATPPB.SREQ.timeOutVal = 3;
  myATPPB.SREQ.retryCount = 3;

  errCode = PSendRequest(&myATPPB,false);

  if (errCode == noErr)
  {
//   Reply is in atpResponseBuffer
  }
```

Before sending an ATP request to another entity, you must set up a Buffer Data Structure (BDS) to indicate where the response from the other entity should be stored. This is accomplished by calling `BuildBDS()` as shown above. `BuildBDS()` is declared as:

```
extern pascal short BuildBDS(Ptr buffPtr, Ptr bdsPtr, short buffSize);
```

Parameters:

`buffPtr`	A pointer to the buffer where the response packet data will be stored.
`bdsPtr`	A pointer to the BDS data structure to set up.
`buffSize`	The size of the response buffer.

Next, the `PSendRequest()` function is declared as:

```
extern pascal OSErr PSendRequest(ATPPBPtr thePBPtr, Boolean async);
```

Parameters:

→	`ioCompletion`	ProcPtr	A completion routine pointer for asynchronous calls.
←	`ioResult`	OSErr	The return code or function result.
→	`userData`	long	For your use.
←	`reqTID`	short	The transaction ID number.

→	csCode	Short	Automatically set to sendRequest.
←	currBitMap	Byte	A bitmap showing which reply packets were received.
↔	atpFlags	Byte	Control flags (see *Inside Macintosh: Networking*).
→	addrBlock	AddrBlock	The destination address.
→	reqLength	Short	The length of the request packet.
→	reqPointer	Ptr	A pointer to the request message.
→	bdsPointer	Ptr	A pointer to the reply BDS.
→	numOfBuffs	Byte	Number of reply packets expected.
→	timeOutVal	Byte	The maximum length of time (in seconds) that ATP should wait for a reply.
←	numOfResps	Byte	Number of response packets received.
↔	retryCount	Byte	The number of retries.
→	TRelTime	Byte	The release timer for exactly-once requests.

As described previously, you must build a BDS structure to indicate to `PSendRequest()` where the response packets from the destination entity should be stored. The response data can consist of up to eight 578-byte packets, for a total of 4,624 bytes of response data. The response data and the request data may contain any information you care to exchange. For example, the previous listing sends a text string, "`Hi there!`" to the destination entity, and Novel Netwar uses ATP packets to determine the status of a player (either INITIALIZING, WAITING, PLAYING, or CLOSING).

Because of the overhead involved in guaranteed delivery of ATP packets, it tends to be a little too slow for fast-action games; however, this same feature of ATP makes it an excellent choice for slower-paced strategy games, such as a networked version of chess. Regardless of the application, it's a good idea to call `PSendRequest()` asynchronously and then periodically check the `ioResult` field to determine when the call completes. While your program is waiting for this to happen, you can deal with user actions, animation tasks, and so on. And if you do call `PSendRequest()` asynchronously and later wish to abort an incomplete ATP request, just use `PKillSendReq()` like this:

```
ATPParamBlock  CancelATPPB;

  CancelATPPB.KILL.aKillQEl = (Ptr) &myATPPB;

  errCode = PKillSendReq(&myCancelATPPB,false);
```

Receiving ATP Requests

The last ATP topic we need to look at is the receiving of and replying to ATP requests. For the ATP request-reply mechanism to work properly, your program must be listening for an ATP request before the request is sent by the other network entity. Also, your program must send back an ATP reply packet in a reasonable time. This process could be as simple as the following:

```
OSErr           errCode;
ATPParamBlock   requestATPPB,replyATPPB;
char            atpReceiveBuffer[256],atpReplyBuffer[256];
short           numBDSBuffers;
BDSType         theBDS;


// Set up for and get ATP request packet

   requestATPPB.ATP.ioCompletion = nil;
   requestATPPB.ATP.atpSocket = myATPSocketNum;
   requestATPPB.ATP.reqLength = 256;
   requestATPPB.ATP.reqPointer = atpReceiveBuffer;

   errCode = PGetRequest(&requestATPPB,false);

   if (errCode == noErr)
   {
      strcpy(atpReplyBuffer,"Message received!");

      numBDSBuffers = BuildBDS((Ptr) atpReplyBuffer,(Ptr) &theBDS, strlen ➥
(replyBuffer) + 1);


      replyATPPB.ATP.ioCompletion = nil;
      replyATPPB.ATP.atpSocket = myATPSocketNum;
      replyATPPB.ATP.atpFlags = atpEOMvalue;
      replyATPPB.ATP.addrBlock = requestATPPB.ATP.addrBlock;
      replyATPPB.ATP.bdsPointer = (Ptr) &asynchATPReplyBDS;
      replyATPPB.OTH1.u.numOfBuffs = 1;
      replyATPPB.OTH2.bdsSize = numBDSBuffers;
      replyATPPB.OTH2.transID = requestATPPB.ATP.reqTID;

      errCode = PSendResponse(&replyATPPB,true);
   }
```

The `PGetRequest()` function is the complement of the `PSendRequest()` function and is declared as:

```
extern pascal OSErr PGetRequest(ATPPBPtr thePBPtr, Boolean async);
```

Parameters:

→	`ioCompletion`	ProcPtr	A completion routine pointer for asynchronous calls.
←	`ioResult`	OSErr	The return code or function result.
←	`userData`	long	User data sent by source entity.
←	`reqTID`	short	The transaction ID number.
→	`csCode`	Short	Automatically set to `getRequest`.
→	`atpSocket`	Byte	The ATP socket to receive with.
←	`atpFlags`	Byte	Control flags (see *Inside Macintosh: Networking*).
←	`addrBlock`	AddrBlock	The source address.
↔	`reqLength`	Short	On entry, this is the length of the request buffer; on return, it is the actual length of the received data.
→	`reqPointer`	Ptr	A pointer to the request buffer.
←	`bitMap`	Byte	A bitmap for the transaction.

`PGetRequest()` waits for an ATP request to arrive at the specified socket, at which point it returns to the calling program (assuming it was called synchronously). At that point, the request message packet is stored in the buffer pointed to by `reqPointer`. If the received data packet is larger than the size indicated by `reqLength`, the data will be truncated and `reqLength` will be set to indicate the nontruncated size of the received data packet.

Once `PGetRequest()` returns and indicates that a request has been received, your program should call `PSendResponse()` to send a response packet. `PSendResponse()` is declared as:

```
extern pascal OSErr PSendResponse(ATPPBPtr thePBPtr, Boolean async);
```

Parameters:

→	`ioCompletion`	ProcPtr	A completion routine pointer for asynchronous calls.
←	`ioResult`	OSErr	The return code or function result.
→	`userData`	long	User data to send back to source.
→	`csCode`	Short	Automatically set to `sendResponse`.
→	`atpSocket`	Byte	The ATP socket to reply with.
→	`atpFlags`	Byte	Control flags (see *Inside Macintosh: Networking*).
→	`addrBlock`	AddrBlock	The address to which the reply should be sent (i.e., the source of the request).

→	`bdsPointer`	Ptr	A pointer to the reply BDS.
→	`numOfBuffs`	Byte	The number of response packets.
→	`bdsSize`	Byte	The number of elements in the reply BDS.
→	`transID`	short	The transaction ID number, as given in the `reqTID` of the request `ATPParamBlock`.

`PSendResponse()` sends response data back to the source of the received ATP request. As mentioned previously, the data may be of any nature, so long as it is no more than 4,624 bytes in length.

If the response data is 578 bytes or less, only one response packet is sent; if it is longer than 578 bytes, multiple response packets are sent, increasing the overall time involved. Once again, this reliable delivery service is nice, but slower than a high-performance application typically requires—particularly if done synchronously. Novel Netwar uses ATP for communicating the status of the player, not for the actual exchange of information during game play.

Asynchronous ATP Transactions

The sluggishness of ATP transactions can be reduced by making use of completion routines, though this entails more tricky programming—including some assembly language "glue" routines. Novel Netwar does just this, resulting in increased responsiveness to ATP requests and automatic handling of ATP requests at the interrupt level. The latter means that it becomes unnecessary to continually check the `ioResult` field of a pending `PGetRequest()` call; instead, when a request is received, the completion routine is called to send the reply, and then execution of the main program continues. This is an elegant way to handle ATP transactions, though it's also a good way for a careless programmer to crash the machine.

The first difficulty to contend with when writing a completion routine is the fact that completion routines are called at an interrupt level, so the value of register A5 is unreliable. If we wish to access our application global variables from within our completion routine, we need to temporarily alter the value of A5. In order to do this, we will use the following data type, which is simply an `ATPParamBlock` with an extra four-byte field prepended to it.

```
struct ATPParamBlockPlusA5
{
  unsigned long  A5;
  ATPParamBlock  theATPParamBlock;
};
typedef struct ATPParamBlockPlusA5 ATPParamBlockPlusA5;
```

When a completion routine is called, one of the few parameters available to it is a pointer to the `ATPParamBlock` for the `PGetRequest()` call. Thus, if we ensure that the correct value of A5 is always stored just before the `ATPParamBlock`, we can always set up A5 to access our application globals. The setting up of the `ATPParamBlockPlusA5` and asynchronous call to `PGetRequest()` are done like this:

```
// global vars

static ATPParamBlockPlusA5  myAsynchReceiveATPPB;
static UInt8                myATPSocketNum;
static Ptr                  myATPAsynchReceiveBuffer;


// local vars

OSErr          errCode;


// Get current value of A5

   myAsynchReceiveATPPB.A5 = SetCurrentA5();

   myAsynchReceiveATPPB.theATPParamBlock.ATP.ioCompletion = (ATPCompletionUPP) ➥
ATPReceiveCompletionRoutineASM;
   myAsynchReceiveATPPB.theATPParamBlock.ATP.atpSocket = myATPSocketNum;
   myAsynchReceiveATPPB.theATPParamBlock.ATP.reqLength = MAXATPPACKETSIZE;
   myAsynchReceiveATPPB.theATPParamBlock.ATP.reqPointer = myATPAsynchReceiveBuffer;

   errCode = PGetRequest(&(myAsynchReceiveATPPB.theATPParamBlock), true);
```

Unlike the previous example involving `PGetRequest()`, this time it is called with `asynch = true`. Also notice that the `ioCompletion` field is set to point to the completion routine, `ATPReceiveCompletionRoutineASM()`; that `myATPSocketNum` has already been opened; and that `myATPAsynchReceiveBuffer` has previously been allocated.

Note

Note that the first parameter to `PGetRequest()` is a pointer to the `ATPParamBlock` part of the `ATPParamBlockPlusA5` structure, not *a pointer to the whole `ATPParamBlockPlusA5` structure.*

When the asynchronous call to `PGetRequest()` finishes, the completion routine, `ATPReceiveCompletionRoutineASM()`, is called. The parameter passed to this routine is contained in the A0 register, and it consists of a pointer to the `ATPParamBlock` for the original call to `PGetRequest()`. In our case, we have stored the value of A5 four bytes before the `ATPParamBlock`, so it can easily be accessed from `ATPReceiveCompletionRoutineASM()`:

```
static asm void ATPReceiveCompletionRoutineASM(void)
{
// Save registers other than D0-D2 and A0-A1

   movem.l   D3-D7/A2-A6,-(A7)


// Get our A5 back
```

```
    movea.l    -4(A0),A5

// Set up the parameters and call the C completion routine

    movem.l    A0,-(A7)

    movem.l    A5,-(A7)

    jsr        ATPReceiveCompletionC

// Clean up the stack

    addq.l     #0x08,A7

// Restore registers and return

    movem.l    (A7)+,D3-D7/A2-A6

    rts
}
```

Note that the completion routines listed in this chapter are written using some 68K assembly language, so they will obviously not compile as native-mode PowerPC code. Stuart Cheshire offers the following advice and comments on the topic of completion routines and the PowerPC:

The way I handle Universal Procedure pointers at the moment is to have a global that is initialized at startup time:

```
// Global variable:
ATPCompletionUPP ATPReceiveCompletionRoutineASM_UPP;

// In the initialization code:
ATPReceiveCompletion_UPP = NewATPCompletionProc(ATPReceiveCompletionRoutineASM);

// And then your code says:
myAsynchReplyATPPB.theATPParamBlock.ATP.ioCompletion = ATPReceiveCompletion_UPP;
```

I'm not saying that this will guarantee that your program will work when compiled native for PowerPC—as far as I know the System software still doesn't support native completion routines—but one day it probably will, and then your code should work correctly with a native compiler.

Of course, there is still the problem of the 68K assembly language, but most of the code listed here could be rewritten in C if Apple were to provide a high-level language interface for these interrupts.

As you can see, very little work is done in the assembler completion routine. It simply sets the value of A5 so that application globals are accessible, and then it calls the C completion routine, `ATPReceiveCompletionC()`.

```
// Global vars

static int                  appletalkingStatus;
static UInt8                myATPSocketNum;
static char                 doAsynchATP;
static Ptr                  myATPAsynchReceiveBuffer, myATPAsynchReplyBuffer;
static BDSType              asynchATPReplyBDS;
static AddrBlock            asynchATPReplyAddrBlock;
static short                asynchATPReplyTransID;
static ATPParamBlockPlusA5  myAsynchReceiveATPPB, myAsynchReplyATPPB;


static void ATPReceiveCompletionC(unsigned long A5,ATPParamBlock *theRequestATPPB)
{
short       numBDSBuffers;
OSErr       errCode;

// Check to make sure we're still supposed to do asynch ATP stuff

  if (doAsynchATP == true)
  {
//   Return a human-readable status string

    switch (appletalkingStatus)
    {
      case INITIALIZING:
        strcpy((char *) myATPAsynchReplyBuffer, "INITIALIZING");
        break;

      case WAITING:
        strcpy((char *) myATPAsynchReplyBuffer, "WAITING");
        break;

      case PLAYING:
        strcpy((char *) myATPAsynchReplyBuffer, "PLAYING");
        break;

      case CLOSING:
        strcpy((char *) myATPAsynchReplyBuffer, "CLOSING");
        break;

      default:
        strcpy((char *) myATPAsynchReplyBuffer, "UNKNOWN");
        break;
    }


//   Note the address and transID for use in killing later

    asynchATPReplyAddrBlock = myAsynchReceiveATPPB.theATPParamBlock.ATP.addrBlock;

    asynchATPReplyTransID = myAsynchReceiveATPPB.theATPParamBlock.OTH2.transID;
```

```
//   Set up a reply BDS and send it off

     numBDSBuffers = BuildBDS(myATPAsynchReplyBuffer,(Ptr) ➥
&asynchATPReplyBDS,strlen((char *) myATPAsynchReplyBuffer) + 1);


     myAsynchReplyATPPB.A5 = A5;

//   See appletalking.c for ATPReplyCompletionRoutineASM

     myAsynchReplyATPPB.theATPParamBlock.ATP.ioCompletion = (ATPCompletionUPP) ➥
ATPReplyCompletionRoutineASM;
     myAsynchReplyATPPB.theATPParamBlock.ATP.atpSocket = myATPSocketNum;
     myAsynchReplyATPPB.theATPParamBlock.ATP.atpFlags = atpEOMvalue;
     myAsynchReplyATPPB.theATPParamBlock.ATP.addrBlock = ➥
myAsynchReceiveATPPB.theATPParamBlock.ATP.addrBlock;
     myAsynchReplyATPPB.theATPParamBlock.ATP.bdsPointer = (Ptr) &asynchATPReplyBDS;
     myAsynchReplyATPPB.theATPParamBlock.OTH1.u.numOfBuffs = 1;
     myAsynchReplyATPPB.theATPParamBlock.OTH2.bdsSize = numBDSBuffers;
     myAsynchReplyATPPB.theATPParamBlock.OTH2.transID = ➥
myAsynchReceiveATPPB.theATPParamBlock.ATP.reqTID ;

     errCode = PSendResponse(&(myAsynchReplyATPPB.theATPParamBlock), true);

     if (errCode != noErr)
     {
//      Should probably post a Notification Manager Event…
     }
   }
}
```

`ATPReceiveCompletionC()` does all the real work of sending the ATP reply, though most of what it does should make sense. The only subtle part of the procedure is that it sets up an asynchronous `PSendResponse()` call that, as part of its completion routine, eventually results in another asynchronous `PGetRequest()` call being made. This circular process is necessary to ensure that the program is ready to receive requests as soon as it has finished replying to the previously received one. It would, of course, be possible to set up several asynchronous calls to `PGetRequest()`, though that would require more buffers and `ATPParamBlocks`. A more high-performance application would do things that way, but for Novel Netwar, it's not worth worrying about a missed request or two while a reply is being sent.

Note

Since `ATPReceiveCompletionC()` is called at interrupt time, it must not make any system calls that allocate or move memory.

Finally, if you are using asynchronous ATP routines, it's necessary to kill them before closing the socket and exiting your application. Novel Netwar does this as follows:

```
// Global vars

static char                 doAsynchATP;
static UInt8                myATPSocketNum;
static AddrBlock            asynchATPReplyAddrBlock;
static short                asynchATPReplyTransID;
static ATPParamBlockPlusA5  myAsynchReceiveATPPB, myAsynchReplyATPPB;


// Local vars

OSErr          errCode;
ATPParamBlock  myATPPB;


// Tell asynch routines to stop aysnching!

   doAsynchATP = false;


// Check if an asynch receive is outstanding

   if (myAsynchReceiveATPPB.theATPParamBlock.ATP.ioResult == 1)
   {
      myATPPB.KILL.aKillQEl = (Ptr) &(myAsynchReceiveATPPB.theATPParamBlock);

      errCode = PKillGetReq(&myATPPB,false);

   }


// Check if an asynch reply is outstanding

   if (myAsynchReplyATPPB.theATPParamBlock.ATP.ioResult == 1)
   {
      myATPPB.ATP.atpSocket = myATPSocketNum;
      myATPPB.ATP.addrBlock = asynchATPReplyAddrBlock
      myATPPB.OTH2.transID = asynchATPReplyTransID;

      errCode = PRelRspCB(&myATPPB,false);

   }
```

Note that the `doAsynchATP = false` is important, since the completion routines could very well interrupt the whole process of trying to kill outstanding asynchronous calls. Interrupts are tricky, and it can never hurt to be very careful when dealing with them.

Datagram Delivery Protocol (DDP)

In contrast to ATP, the Datagram Delivery Protocol is not a reliable delivery protocol; rather, it is a best-effort delivery protocol, which means it will try to send the message, but will not retry or attempt to confirm its receipt. This form of service is less reliable than ATP, but requires far less overhead and is thus much faster than ATP. For an application such as a game like Novel Netwar, DDP is a much better choice than ATP, since an occasional dropped or lost packet doesn't matter much.

As with ATP, DDP packets are limited in size, though DDP packets may be up to 586 bytes long, and only one packet may be sent at a time. Naturally, the data sent in a DDP packet may be of any nature that the programmer sees fit to send. In the case of Novel Netwar, each copy of the game uses DDP to broadcast the position and orientation of the local player, as well as the locations of the player's photon torpedoes, which amount to only 26 bytes of data.

Opening a DDP Socket

As with ATP, your program must open a DDP socket before it begins to send or receive network data. This is done using the `POpenSkt()` function:

```
#define MYGAMEDDPSOCKETNUMBER    64


// Global vars

static UInt8   myDDPSocketNum;


// Local vars

OSErr            errCode;
ATPParamBlock    myATPPB;
MPPParamBlock    myMPPPB;


   initMyDDPSocketListener();


   myMPPPB.DDP.socket = MYGAMEDDPSOCKETNUMBER;
   myMPPPB.DDP.u.listener = (DDPSocketListenerUPP) myDDPSocketListener;

   errCode = POpenSkt(&myMPPPB,false);

   if (errCode == noErr)
   {
      ddpSocketOpened = true;

      myDDPSocketNum = myMPPPB.DDP.socket;
   }
```

`POpenSkt()` is declared as follows:

```
extern pascal OSErr POpenSkt(MPPPBPtr thePBptr, Boolean async);
```

Parameters:

→	`ioCompletion`	ProcPtr	Completion routine pointer.
←	`ioResult`	OSErr	The return code or function result.
→	`csCode`	Short	Automatically set to `openSkt`.
↔	`socket`	Byte	The desired socket number or zero for a dynamically allocated socket number.
→	`listener`	Ptr	A pointer to a socket listener (see next section for more information).

If you examine the Novel Netwar source, you'll notice that it opens DDP socket number 64 rather than dynamically allocating a socket by setting `socket` to zero before calling `POpenSkt()`. Apple advises against doing this, since there will be problems if another application is already using that socket number; however, I chose to do it anyway. By choosing a hard-coded socket number, I ensure that all copies of Novel Netwar are easily located on the network. As you'll see shortly, by fixing the network socket address of game participants, it becomes possible to broadcast information to all players at once, rather than having to send packets to individual addresses. This process will be explained more thoroughly in the next few sections of this chapter.

After you've opened a DDP socket, you must be sure to close it again before your application exits. This is done using the `PCloseSkt()` function:

```
OSErr           errCode;
MPPParamBlock   myMPPPB;

  myMPPPB.DDP.socket = myDDPSocketNum;

  errCode = PCloseSkt(&myMPPPB,false);
```

DDP Socket Listeners

If you were reading carefully, you noticed the following line in the source code listing that showed how to open a DDP socket:

```
myMPPPB.DDP.u.listener = (DDPSocketListenerUPP) myDDPSocketListener;
```

The `listener` parameter is a key parameter for the `POpenSkt()` call, since it must contain a pointer to a routine that will handle incoming DDP packets. This routine is called a socket listener, and it is the heart of the DDP packet-handling process. If you wish to use DDP services in your application, you must provide a socket listener. If you pass NIL for the listener parameter, your application will crash, probably taking the rest of the system with it.

A socket listener is similar to a completion routine in that it is called at the interrupt level and thus must be implemented using some assembly language. The Novel Netwar socket handler is fairly simple to understand, since only a short portion of it is written in assembler:

```
// Declare prototypes first

static asm void initMyDDPSocketListener(void);
static asm void myDDPSocketListener(void);
static asm void myDDPSocketListenerASMCode(void);


static asm void myDDPSocketListenerASMCode(void)
{
// Safeguard accidental calls to myDDPSocketListenerASMCode()

   rts


// Storage for A5

myA5:   dc.l 0


// Initialize listener--i.e. save A5 in myA5 for later use (note
// that this is *not* self-modifying code!)

ENTRY   static  initMyDDPSocketListener

   lea     myA5,A3
   movem.l A5,(A3)
   rts


// The actual socket listener

ENTRY   static  myDDPSocketListener

// Set up A5 so we can access application globals

   lea     myA5,A3
   movem.l (A3),A5


// Read the rest of the DDP packet into my buffer; beware errors

   lea     myDDPReadBuffer.packetData,A3
   move.l    D1,D3
   jsr     2(A4)
   bne     @EXIT


// Save registers
```

```
    movem.l D4-D7/A5-A6,-(A7)

// Call to ReadRest may have altered A5, so restore it

    lea     myA5,A3
    movem.l (A3),A5


// Get the source address and store it in myDDPReadBuffer (this
// could be done faster with a couple of move.l instructions…)

    lea     myDDPReadBuffer,A3

    move.b  struct (DDPDataPacket.destNodeID) + toRHA (A2),D2
    move.b  D2,struct (DDPDataPacket.destNodeID) (A3)

    move.b  struct (DDPDataPacket.srcNodeID) + toRHA (A2),D2
    move.b  D2,struct (DDPDataPacket.srcNodeID) (A3)

    move.b  struct (DDPDataPacket.ddpHeaderType) + toRHA (A2),D2
    move.b  D2,struct (DDPDataPacket.ddpHeaderType) (A3)

    move.b  struct (DDPDataPacket.datagramLengthMSB) + toRHA (A2),D2
    move.b  D2,struct (DDPDataPacket.datagramLengthMSB) (A3)

    move.b  struct (DDPDataPacket.datagramLengthLSB) + toRHA (A2),D2
    move.b  D2,struct (DDPDataPacket.datagramLengthLSB) (A3)

    move.b  struct (DDPDataPacket.destSocketNumber) + toRHA (A2),D2
    move.b  D2,struct (DDPDataPacket.destSocketNumber) (A3)

    move.b  struct (DDPDataPacket.srcSocketNumber) + toRHA (A2),D2
    move.b  D2,struct (DDPDataPacket.srcSocketNumber) (A3)

    move.b  struct (DDPDataPacket.ddpProtocolType) + toRHA (A2),D2
    move.b  D2,struct (DDPDataPacket.ddpProtocolType) (A3)


// Call the C code to process the packet

    jsr     processDDPPacket

// Restore registers before returning…

    movem.l (A7)+,D4-D7/A5-A6

@EXIT:  rts
}
```

As you can see, the socket listener simply reads in the DDP packet and header information, then calls the C routine `processDDPPacket()`, which deals with it. This socket listener is extremely simplistic, and it does not deal with issues such as checksums, multiple input buffers, and so on. For a more detailed treatment of socket listeners, refer to *Inside Macintosh: Networking*, pp. 7.20-7.32.

The actual C routine, `processDDPPacket()`, is fairly short, since it simply checks to make sure that the received DDP packet came from a socket of the correct number, that the packet is a broadcast packet, and that the packet did not originate at the local node (just in case self-sending was enabled). If all of these conditions are met, the received packet is handed off to the `gameReceiveBroadcastData()` routine, which deals with the actual information contained in the packet. The source code for `processDDPPacket()` is

```
void processDDPPacket(void)
{
  if (myDDPReadBuffer.ddpHeaderType == 1 &&
      myDDPReadBuffer.srcSocketNumber == MYGAMEDDPSOCKETNUMBER &&
      myDDPReadBuffer.destNodeID == DDPBROADCASTNODE &&
      myDDPReadBuffer.srcNodeID != myNodeNum)
  {
    gameReceiveBroadcastData((Ptr) &myDDPReadBuffer,(long) sizeof(DDPDataPacket));
  }
}
```

Of course, your application can do any sort of processing of the received packet data you choose. The socket listener listed above can be easily transplanted into your application and used as is, with the real modifications being done to the C routine, `processDDPPacket()`.

Note

Since the socket listener is called at the interrupt level, it must not allocate or move memory. Also be aware that very subtle errors can occur when the socket listener modifies the application globals. Refer to the gameplay.c file for examples of ways to deal with this.

Sending DDP Packets

Once you have opened your DDP socket and installed your DDP socket listener, you will undoubtedly want to send packets to other network entities. This is accomplished much as it is for ATP, though in this case the relevant routines are `BuildDDPwds()` and `PWriteDDP()`:

```
#define MYDDPPACKETTYPE        0xAA


// Global vars

AddrBlock      destDDPAddress;
```

```
UInt8           myDDPSocketNum;

// Local vars

MPPParamBlock     theDDPWriteMPPPB;
AddrBlock         destDDPAddress;
char              myDDPwds[14],myDDPheader[18];
char              theMessage[256];


   strcpy(theMessage,"Hi There!");

   BuildDDPwds(myDDPwds,myDDPheader,(Ptr) theMessage, destDDPAddress, (short) ➥
MYDDPPACKETTYPE, strlen(theMessage)+1);


   theDDPWriteMPPPB.DDP.ioCompletion = nil;
   theDDPWriteMPPPB.DDP.socket = myDDPSocketNum;
   theDDPWriteMPPPB.DDP.checksumFlag = 0;
   theDDPWriteMPPPB.DDP.u.wdsPointer = (Ptr) &myDDPwds;

   errCode = PWriteDDP(&theDDPWriteMPPPB,false);
```


`BuildDDPwds()` is used to build a DDP write data structure and is declared as:

```
extern pascal void BuildDDPwds(Ptr wdsPtr, Ptr headerPtr, Ptr dataPtr, AddrBlock ➥
netAddr, short ddpType, short dataLen);
```

`wdsPtr` is a pointer to the buffer that will contain the write data structure; this buffer must be at least 14 bytes long.

`headerPtr` is a pointer to a buffer that will contain the DDP packet header; this must be at least 17 bytes long.

`dataPtr` is a pointer to the data to be sent.

`netAddr` is the destination net address.

`ddpType` is the DDP packet protocol type; types 1-15 are reserved by Apple, all others are available for your use.

`dataLen` is the length of the data packet to be sent.

The DDP packet type, specified in the `ddpType` parameter to `BuildDDPwds()`, is used to distinguish between different types of DDP data packets. This value is chosen by the programmer and can be used to distinguish between different kinds of exchanged data, such as a player's location and a text message. Novel Netwar uses only one type of packet, `0xAA`, chosen for no particular reason other than `0xAA` is my favorite bit pattern.

`PWriteDDP()` does the actual work of sending the DDP packet, and is declared as:

```
extern pascal OSErr PWriteDDP(MPPPBPtr thePBptr, Boolean async);
```

Parameters:

→	`ioCompletion`	ProcPtr	Completion routine pointer.
←	`ioResult`	OSErr	The return code or function result.
→	`csCode`	Short	Automatically set to `writeDDP`.
→	`socket`	Byte	The DDP socket to send from.
→	`checksumFlag`	Byte	If this flag is nonzero, a checksum will be calculated and sent with the DDP packet.
→	`wdsPointer`	Ptr	A pointer to the write data structure created by `BuildDDPwds()`.

As with ATP, you may wish to perform the call to `PWriteDDP()` asynchronously, with or without a completion routine. Novel Netwar makes its calls to `PWriteDDP()` asynchronously and also uses a completion routine. If you look at the code for this in appletalking.c, you'll note that the completion routine is almost identical to the ATP completion routines we looked at earlier in this chapter, and that all the routine really does is call the function `gameplayHandleBroadcastCompletion()`. This latter function just sets a flag indicating that the last broadcast message has been sent, so it once again makes sense to attempt another update broadcast of the player's position.

DDP interprets a destination node ID of 255 to indicate a special broadcast packet that will be received by all DDP listeners on the local network so long as they are listening at the specified socket. Novel Netwar uses this broadcast service, rather than sending updates to specific entities looked up using NBP. This technique works well for nonextended networks, but not extended networks (bridges and routers do not usually forward broadcast packets). Other network games establish a logical ring of players, and pass a block of information around the logical ring, having each member of the ring update the portion of the data that relates to the local player. The two approaches have their respective advantages and disadvantages, and if you really want to decide which is best for your application, you'll need to consider interrupt overhead, processing times, and coding complications. Perhaps the best approach would be to perform some timing tests of the two methods and make an empirically motivated decision....

If you are contemplating implementing a logical ring of players for your game, then consider the following comments from Stuart Cheshire, the author of Bolo:

Actually, simple broadcast is the easiest and most common way that network games are written. The old NetTrek for the Mac by Randy Carr did broadcast, as did the first version of Doom for the PC. Some games such as SGI's dogfight use IP Multicast, which is effectively just a slightly more subtle kind of broadcast under a different name.

The next most common technique is client/server, such as X Windows NetTrek and XTank.

As far as I know, Bolo and Spectre are the only games to use a logical ring, and it is a complete coincidence. I didn't even know Steve Newman when he was writing Spectre. We met at a party at Stanford and the following exchange took place: "What do you do?" "Oh, I'm working on a networked multiplayer tank battle game for the Mac" "Oh, I just wrote one of those." "Yeah, sure you did."

I'd say the jury is still out on which method is best. I like the ring, but it is so fraught with difficulties (joining, leaving, crashes, network failures, etc.) that I'd hesitate to recommend it wholeheartedly to someone beginning a new program.

Another method a programmer may wish to consider is the client/server approach, in which one machine controls the list of players and sends messages to all other players at once.

Serial Communication

After AppleTalk, the next most common communications method used on the Macintosh is undoubtedly serial communication. Most Macs have at least two serial ports, the Printer port and Modem port, that support asynchronous serial communications at up to 57,600 bits per second. Typically, these ports are (as the names indicate) used to connect to serial printers and modems, but they can also easily be used to directly connect two Macintoshes.

Incidentally, if you enable the use of serial connections in an application, you have automatically enabled the use of a modem connection as well. The serial port drivers deal only with sending data to and receiving data from the physical port; the drivers have no way of knowing whether there is a direct-connect cable, a modem, or anything else plugged into the port. Although it is possible to build in plenty of extra support for modem configuration and dialing scripts, the only functionality required for modem connections is simple serial port support. If you examine Novel Netwar's serial.c file, you'll see that as part of the `serialConnectToOpponents()` procedure, it implements a simple miniterminal to allow the user to interact with a modem, perhaps setting modem parameters or dialing up another player; however, this is done only as a convenience for the user.

In this section, we'll look at how you can set up your program to allow bidirectional communication using the serial ports. As always, we'll concentrate on the essential information of concern to a game programmer, and the examples will be largely taken from the Novel Netwar game. If you want more information about using the serial ports and the Macintosh Serial Driver, consult *Inside Macintosh: Devices*, available from Addison Wesley (ISBN: 0-201-62271-8) or available via anonymous FTP from `ftp.info.apple.com` as `Apple.Support.Area/Developer_Services/Technical_Documentation/Inside_Macintosh/Inside_Macintosh/Devices/Devices.sit.hqx` (the accompanying index is in the same directory but is called `Devices.idx.sit.hqx`). The electronic version of the book available from `ftp.info.apple.com` is in DocViewer format, so if you're going to FTP it, make sure you have a copy of Apple's DocViewer or else grab a copy from `Apple.Support.Area/Developer_Services/Utilities/Apple_DocViewer__v1.1` while you're there.

Opening the Serial Port Device Drivers

Before your program can send or receive information over the serial ports, it must first open the serial ports using the `OpenDriver()` Device Manager routine. This is demonstrated by the following code:

```
short     inputRefNum,outputRefNum;
OSErr     errCode;


// Open Modem ports for input and output

   errCode = OpenDriver("\p.AOut",&outputRefNum);

   if (errCode != noErr)
   {
//    Should deal with errors such as portInUse…
   }

   errCode = OpenDriver("\p.AIn",&inputRefNum);

   if (errCode != noErr)
   {
//    Again, deal with errors…
   }
```

The example shown opens the Modem port for output and input separately, since the serial driver implements read/write access via distinctly different drivers. The device name for the Modem port input side is `.AIn` and the device name for the output side is `.AOut`. Similarly, the device driver names for the Printer port are `.BIn` and `.BOut`.

The `OpenDriver()` routine is declared as:

```
extern pascal OSErr OpenDriver(ConstStr255Param name, short *drvrRefNum);
```

The parameters to `OpenDriver()` are as follows:

`name` is pointer to the Pascal-style name of the device driver to open.

`drvrRefNum` is a pointer to a short integer, in which will be returned the driver reference number that you must use for subsequent accesses to the driver.

The most likely error to be returned by a call to open the serial ports is `portInUse`, which indicates that the port has already been opened for use by some other process. For example, because AppleTalk usually uses the Printer serial port, if your application attempts to open the Printer port while AppleTalk is turned on, a `portInUse` error is returned.

Note
Inside Macintosh: Devices *says that the output driver should always be opened first, since it initializes the local variables for both the input and output side. Opening just the input driver will load the driver, but will not initialize it.*

Once the serial drivers have been loaded and opened, your program may set the input buffer and configure the speed and handshaking settings of the port.

Configuring the Serial Port

When the serial port is initially opened, it is, by default, set up to communicate at 9,600 bits per second, with 8 data bits per character, no parity, and 2 stop bits. Flow control is CTS/DTR hardware handshaking, and the input buffer is 64 bytes in size. All of these settings can be changed using the `SerReset()`, `Control()`, and `SerSetBuf()` system calls. In many cases, the default port settings are acceptable, but you may wish to alter them to achieve higher performance and greater control over serial communications options.

Novel Netwar allows the user to select speeds of 2,400, 9,600, or 19,200 bits per second, but always uses 8 data bits, no parity, and 1 stop bit.

Setting Serial Port Speed, Data Bits, Parity, and Stop Bits

As mentioned previously, when the serial port is initially opened, it is, by default, set up to communicate at 9,600 bits per second, with 8 data bits per character, no parity, and 2 stop bits. Each of these terms is discussed briefly below before the `SerReset()` system call is detailed.

The serial port speed, or baud rate, is the speed at which data is transferred over the serial connection; obviously, you want this to be as fast as possible, but not so fast that incoming data arrives more rapidly than the computer can handle. For a typical game, most of the processing load is due to graphics, and only a small fraction of the total CPU time should be spent handling communications between machines. The serial hardware on the Macintosh buffers a small number of incoming data bytes, and the serial drivers implement a much larger software buffer, so there is little chance of data being lost, unless your game does not periodically process buffered data. As long as you are careful to deal with incoming data, you should be safe in setting the bit rate as high as possible. The only overriding concern comes into play if the user is using a modem as part of the serial connection, in which case your program should not attempt to drive the serial ports faster than the modem can handle. It's also worth noting that you should *never* use modem compression with a game, since it absolutely kills responsiveness. Games don't send a lot of data, but they do want it delivered quickly. Waiting half a second for the modem to compress your data is the last thing you want.

The number of data bits per character is an obvious setting, with 8 data bits being the most sensible value for most applications. The reasons for allowing fewer than 8 data bits per character are largely historical, since certain hardware from the early days of computer telecommunications sometimes operated with 7 data bits.

The parity is a term to describe an optional bit that can be sent along with each character sent over the serial connection. If the parity is set to odd, then the value sent for the parity bit is chosen so that there will be an odd number of 1's sent for each character. Conversely, if the parity is set to even, then the value sent for the parity bit is chosen so that there will be an even number of 1's sent for each character. If the parity is set to none, then no additional parity bit will be sent. The use of parity is a simple form of error checking, which may be desirable for unreliable connections, such as noisy phone lines; however, such error control is not particularly effective, and just adds transmission overhead, thus slowing connection throughput. It is difficult to imagine a situation wherein anyone would really need to use parity checking for a game, especially since most modems now have transparent error correction protocols built in.

The number of stop bits is an indication of the spacing of outgoing serial data. After a character is sent over the serial connection, additional stop bits are sent, thus separating the transmitted characters. Fewer stop bits means that there is less time wasted between consecutive characters sent, but also allows less chance for the connection at the other end to resynchronize with the transmitted bitstream if errors occur.

The serial port speed, data bits per character, parity, and stop bits are set using the `SerReset()` function. This function is declared as:

```
extern pascal OSErr SerReset(short refNum, short serConfig);
```

`refNum` is the driver reference number returned by `OpenDriver()`.

`serConfig` is a 16-bit value that specifies the port settings.

The `serConfig` parameter is set using the following constants:

```
baud300     = 380
baud600     = 189
baud1200    = 94
baud1800    = 62
baud2400    = 46
baud3600    = 30
baud4800    = 22
baud7200    = 14
baud9600    = 10
baud19200   = 4
baud38400   = 1
baud57600   = 0
stop10      = 16384
stop15      = -32768
stop20      = -16384
noParity    = 0
```

```
oddParity    = 4096
evenParity   = 12288
data5        = 0
data6        = 2048
data7        = 1024
data8        = 3072
```

To select the desired speed, data bits, parity, and stop bits, add (or logically OR) the constants listed and pass in the result as the `serConfig` parameter to `SerReset()`. For example, to set the port for 19,200 bits per second, 8 data bits, no parity, and 2 stop bits, you could do this:

```
errCode = SerReset(outputRefNum,baud19200+data8+noParity+stop20);
```

To configure the serial ports, always call `SerReset()` with the driver reference number of the output serial port. The input port is always configured the same as the output port, so it is not necessary to configure both input and output ports.

Setting Serial Port Handshaking

As mentioned previously, when the serial port is initially opened, flow control is set to CTS/DTR hardware handshaking. To understand what this means, consider what happens if incoming data arrives faster than the receiving program can handle it: Incoming data will be ignored and lost. The simplest solution to such a problem is to simply ignore the incoming data until the program is again ready to deal with incoming serial data; this is also the least desirable solution to the problem, since all the ignored data will be lost and the sender will have to resend it. A much better solution is to tell the sender to temporarily stop sending data, thus allowing the receiver to catch up and avoiding the complications of dealing with lost data. This is what flow control or handshaking is all about, and it is implemented on the Macintosh in two forms: hardware handshaking or XON/XOFF flow control.

Hardware handshaking consists of asserting the DTR control line (pin 1 of the hardware port) to indicate to the other end of the connection that the Macintosh is ready to receive data, or dropping DTR to indicate that the other end of the connection should stop sending data. Similarly, the other end of the connection asserts or drops the signal on CTS (pin 2) to control data flow from the Macintosh. Because of its use of control lines, hardware handshaking requires a special cable and hardware support from both ends of the connection (most high-speed modems require hardware handshaking).

Flow control can also be handled in software by using an agreed-upon set of characters as start and stop signals. The Serial Driver supports XON/XOFF flow control, which typically assigns the Control-Q (0x11) character as the start signal and the Control-S (0x13) character as the stop signal, although you can choose different characters as suits your application.

To configure the handshaking options for the serial ports, the Device Manager `Control()` function is called. There is a related function called `SerHShake()` that you can use instead, but it does not give you complete control over all handshaking options. As an example usage of

`Control()` to set handshaking options, consider the following code snippet, which turns on hardware handshaking and disables XON/XOFF flow control:

```
SerShk     mySerShk;
OSErr      errCode;

   mySerShk.fXOn  = false;
   mySerShk.fCTS  = true;
   mySerShk.xOn   = 0;
   mySerShk.xOff  = 0;
   mySerShk.errs  = 0;
   mySerShk.evts  = 0;
   mySerShk.fInX  = false;
   mySerShk.fDTR  = true;

   errCode = Control(outputRefNum,14,&mySerShk);
```

Similarly, the following code snippet shows how XON/XOFF flow control could be turned on along with hardware handshaking (an odd combination, but someone somewhere might want to do this):

```
SerShk     mySerShk;
OSErr      errCode;

   mySerShk.fXOn  = true;
   mySerShk.fCTS  = true;
   mySerShk.xOn   = 0x11;
   mySerShk.xOff  = 0x13;
   mySerShk.errs  = 0;
   mySerShk.evts  = 0;
   mySerShk.fInX  = true;
   mySerShk.fDTR  = true;

   errCode = Control(outputRefNum,14,&mySerShk);
```

The `SerShk` data structure which is passed as a parameter to `Control()` is defined as follows:

```
struct SerShk {
   Byte      fXOn;        // XON/XOFF output flow control flag
   Byte      fCTS;        // CTS output flow control flag
   unsigned char  xOn;    // XOn character
   unsigned char  xOff;   // XOff character
   Byte      errs;        // errors mask bits
   Byte      evts;        // event enable mask bits
   Byte      fInX;        // XON/XOFF input flow control flag
   Byte      fDTR;        // DTR input flow control flag
};
typedef struct SerShk SerShk;
```

The errs field of the SerShk data structure is used to control the behavior of the serial driver in response to parity errors, buffer overruns, and framing errors. The evts field is used to control the behavior of the serial driver in response to break signals. For simple applications, you can disable and ignore these features, but if you want more information on the use of these features, consult *Inside Macintosh: Devices*.

Setting the Serial Port Input Buffer

When the serial port is initially opened, the default input buffer is set to the driver's internal 64-byte buffer. If large amounts of data are going to be received over the serial connection, then it is necessary to instruct the serial driver to use a user-specified buffer. The SerSetBuf() function allows an application's programmer to specify a buffer for use by the input serial driver. SerSetBuf() is declared as:

```
extern pascal OSErr SerSetBuf(short refNum, Ptr serBPtr, short serBLen);
```


The parameters to SerSetBuf() are described as follows:

refNum is the driver reference number returned by OpenDriver().

serBPtr is a pointer to a user-allocated buffer.

serBLen is the length of the user-allocated buffer.

As an example of the use of SerSetBuf(), consider the following:

```
Ptr       myBuffer;
OSErr     errCode;


  myBuffer = NewPtr(1024);

  if (myBuffer != nil)
  {
     errCode = SerSetBuf(inputRefNum,myBuffer,1024);
  }
```

Note

Before your application exits, it is essential that you restore the original serial buffer by calling SerSetBuf() with a zero value for serBLen. If you do not reset the serial input buffer, Bad Things will happen if serial input data is received later. For example, the system will probably crash if incoming data is stored in a buffer whose address now corresponds to the CODE segment of your word processor.

Sending Serial Data

Once the serial port is opened and configured, your application may send data out the serial port using the `PBWrite()` Device Manager call as shown below:

```
OSErr          errCode;
ParamBlockRec  myWritePB;
char           myMessage[256];


  strcpy(myMessage,"Hi There!");

  myWritePB.ioParam.ioCompletion = nil;
  myWritePB.ioParam.ioRefNum = outputRefNum;
  myWritePB.ioParam.ioBuffer = (Ptr) myMessage;
  myWritePB.ioParam.ioReqCount = strlen(myMessage) + 1;

  errCode = PBWrite(&myWritePB,false);
```

`PBWrite()` is declared as:

```
extern pascal OSErr PBWrite(ParmBlkPtr paramBlock, Boolean async);
```

Parameters:

→	`ioCompletion`	ProcPtr	Completion routine pointer.
←	`ioResult`	OSErr	The return code or function result.
→	`ioRefNum`	Short	The driver reference number returned by `OpenDriver()`.
→	`ioBuffer`	Ptr	A pointer to the data to be written.
→	`ioReqCount`	long	The length of the data to be written.
←	`ioActCount`	long	The actual number of bytes written.

As with many of the calls to the AppleTalk manager, you may wish to call `PBWrite()` asynchronously, with or without a completion routine. The `serialSendData()` routine in Novel Netwar's serial.c file calls `PBWrite()` asynchronously with a completion routine, much as `appleTalkingSendData()` in appletalking.c did.

Receiving Serial Data

To read incoming data from the serial ports, the `SerGetBuf()` and `PBRead()` functions are used:

```
OSErr          errCode;
ParamBlockRec  myReadPB;
char           incomingMessage[256];
long           numCharsInBuff;
```

```
// Find out how many bytes are in input buffer

   errCode = SerGetBuf(inputRefNum,&numCharsInBuff);

   if (errCode == noErr && numCharsInBuff > 0)
   {
//    We have room for only 256 bytes…

      if (numCharsInBuff > 256)
      {
         numCharsInBuff = 256;
      }

//    Set up and read in data from input buffer

      myReadPB.ioParam.ioCompletion = nil;
      myReadPB.ioParam.ioRefNum = inputRefNum;
      myReadPB.ioParam.ioBuffer = (Ptr) incomingMessage;
      myReadPB.ioParam.ioReqCount = numCharsInBuff;

      errCode = PBRead(&myReadPB,false);
   }
```


The `SerGetBuf()` function is used to determine the number of bytes currently in the input buffer, and is declared as:

```
extern pascal OSErr SerGetBuf(short refNum, long *count);
```

The parameters to `SerGetBuf()` are defined as follows:

`refNum` is the driver reference number returned by `OpenDriver()`.

`count` is a pointer to a long integer, in which will be returned the number of characters currently available in the input buffer.

Once your program determines that there is data available in the input buffer, `PBRead()` is used to remove the data from the input buffer and copy it to a user-specified location. `PBRead()` is declared as:

```
extern pascal OSErr PBRead(ParmBlkPtr paramBlock, Boolean async);
```

Parameters:

→	`ioCompletion`	ProcPtr	Completion routine pointer.
←	`ioResult`	OSErr	The return code or function result.
→	`ioRefNum`	Short	The driver reference number returned by `OpenDriver()`.

→	`ioBuffer`	Ptr	A pointer to the buffer where the data will be transferred.
→	`ioReqCount`	long	The number of bytes to transfer to the buffer pointed to by `ioBuffer`.
←	`ioActCount`	long	The actual number of bytes read.

Once again, an asynchronous call to `PBRead()` is generally desirable—particularly if you choose to attempt to read more data than is currently available in the input buffer. For many applications, particularly games, the data that is transferred is of fixed size, and it makes sense to issue an asynchronous read call asking for a fixed-size packet of data. Then, when the requested amount of data is received, a completion routine handles the received packet and queues up a request for another packet of data. Novel Netwar takes this approach to handling incoming serial data, and the results are much better than would be achieved by constantly polling and reading in a few bytes at a time.

As always, if you wish to use asynchronous calls with completion routines, you must write at least a glue routine in assembler, taking care to set up register A5 if you wish to access application globals, and also taking care not to permanently modify the values of registers D3-D7 and A2-A6. The Novel Netwar completion routine shown below handles most of these details and passes control to the C routine `SerialReceiveCompletionC()`. If you use this routine in your own programs, be sure to call the initialization entry point, `initSerialReadCompletionRoutine()`, to save the value of A5 for later use in the actual completion routine.

```
static asm void mySerialCompletionASMCode(void)
{
// Just in case somebody tries to call mySerialCompletionASMCode()

     rts


// Storage for A5

myA5:   dc.l 0


// Init routine to save current A5 for later use

ENTRY   static  initSerialReadCompletionRoutine

     lea        myA5,A3
     movem.l    A5,(A3)
     rts


// Actual completion routine

ENTRY   static  SerialReadCompletionRoutineASM
```

```
// Save registers

     movem.l   D3-D7/A2-A6,-(A7)

// Set up A5

     lea       myA5,A3
     movem.l   (A3),A5


// Call C routine to handle completion of read

     jsr       SerialReceiveCompletionC


// Restore saved registers

     movem.l   (A7)+,D3-D7/A2-A6

@EXIT: rts
}
```


Synchronizing Serial Data Streams

AppleTalk is described as a packet-oriented communications protocol, whereas a serial connection is a streamed protocol. The difference between these two protocols is that packet-oriented protocols send and receive data in fixed packets, whereas a streamed connection sends and receives data as a continuous stream of bytes. The streamed nature of a serial connection means that if a receiving end just reads a fixed number of bytes until it believes it has read a complete message, it may have, in fact, read the end of one message and the start of the following message. To ensure that the receiver and sender are synchronized with respect to the start and ends of messages, the receiving end must have some way of determining when a message starts and stops.

Novel Netwar deals with the problem of synchronizing the data stream by appending a unique end-of-message character to every packet of data which is sent over the connection. This sounds like a simple solution until you realize that the data being sent could contain that end-of-message character, too. To contend with this complication, the `serialSendData()` routine makes a copy of the outgoing data, inserting a second end-of-message character after any occurrence of the character in the data. Thus, when the receiving routine reads incoming serial data, it interprets an occurrence of two consecutive end-of-message characters as a single byte with the same value, and a single end-of-message character as a true end-of-message marker.

Since the receiving end understands the meaning of the end-of-message characters and knows the expected size of the incoming data packets, this simple strategy works. If the receiver receives a single end-of-message character before it has read a complete data packet,

it discards all the data read up to that point. Things would be more complicated if variable-sized packets were used, but fortunately, that is not the case for Novel Netwar or most games of similar nature, because they use fixed-length packets.

The end-of-message detection algorithm implemented for Novel Netwar is far from perfect, since unusual conditions can still cause problems for it; however, it is conceptually simple, and by default it just discards damaged or incomplete packets.

The `serialSendData()` routine is listed below:

```
#define ENDPACKETCHAR    0xFF

// Global vars

static Ptr               serialOutputBufferPtr;
static ParamBlockRec     myWritePB;


void serialSendData(Ptr theDataPtr,long theDataLength)
{
OSErr                    errCode;
register unsigned char   *srcBytePtr,*destBytePtr;
register long            srcbyteCount,destByteCount;


// Make sure last asynch write is completed

   if (myWritePB.ioParam.ioResult != 1)
   {
//    Deal with errors from last write

      errCode = myWritePB.ioParam.ioResult;

      if (errCode != noErr)
      {
        sprintf(errorMessage,"serialSendData: PBWrite() error %d, asynch send of ➥
serial data was unsuccessful!",(int) errCode);

        ErrorAlert(errorMessage);
      }


//    Copy data, doubling occurrences of ENDPACKETCHAR

      srcBytePtr = (unsigned char *) theDataPtr;
      destBytePtr = (unsigned char *) serialOutputBufferPtr;
      srcbyteCount = 0;
      destByteCount = 0;

      while (srcbyteCount < theDataLength)
      {
        if (*srcBytePtr == ENDPACKETCHAR)
```

```
        {
          *destBytePtr++ = ENDPACKETCHAR;
          destByteCount++;
        }

        *destBytePtr++ = *srcBytePtr++;
        srcbyteCount++;
        destByteCount++;
      }

//    Append an extra ENDPACKETCHAR

      *destBytePtr = ENDPACKETCHAR;
      destByteCount++;


//    Set up paramblock and write data asynchronously

      myWritePB.ioParam.ioCompletion = nil;
      myWritePB.ioParam.ioRefNum = outputRefNum;
      myWritePB.ioParam.ioBuffer = serialOutputBufferPtr;
      myWritePB.ioParam.ioReqCount = (short) destByteCount;


//    Deal with errors

      errCode = PBWrite(&myWritePB,true);

      if (errCode != noErr)
      {
        sprintf(errorMessage,"serialSendData: PBWrite() error %d, can't start ➥
asynch send of serial data!",(int) errCode);

        ErrorAlert(errorMessage);
      }
    }
}
```

The complementary routine to `serialSendData()` is the read completion routine, `SerialReceiveCompletionC()`, which is listed below:

```
#define ENDPACKETCHAR     0xFF

// Global vars

static Ptr                serialInputBufferPtr,serialPacketBuffer;
static ParamBlockRec      myReadPB;
static long               numBytesInPacketBuff;
```

```
static void SerialReceiveCompletionC(void)
{
OSErr                   errCode;
register unsigned char  *srcBytePtr,*destBytePtr;
register long           numBytesToProcess;


// Figure out where incoming data was stored

  srcBytePtr = (unsigned char *) serialPacketBuffer + numBytesInPacketBuff;
  destBytePtr = (unsigned char *) serialPacketBuffer + numBytesInPacketBuff;


// Figure out how many bytes were actually read

  numBytesToProcess = myReadPB.ioParam.ioActCount;


// Process all bytes read

  while (numBytesToProcess > 0)
  {
//   Deal with ENDPACKETCHAR markers

    if (*srcBytePtr == ENDPACKETCHAR)
    {
//    Skip over it

      srcBytePtr++;
      numBytesToProcess--  ;


//    See if we're done processing

      if (numBytesToProcess == 0)
      {
        break;
      }


//    Next byte is *not* a second ENDPACKETCHAR, so it
//     really was an end-of-packet marker, and we should
//     discard all previous data

      else if (*srcBytePtr != ENDPACKETCHAR)
      {
        numBytesInPacketBuff = 0;
        destBytePtr = (unsigned char *) serialPacketBuffer;
      }
    }


//  Copy bytes (only necessary because we may have deleted one
```

```
//    of a pair of duplicate ENDPACKETCHAR characters)

      *destBytePtr++ = *srcBytePtr++;
      numBytesToProcess--;
      numBytesInPacketBuff++;
   }


// Check to see if we've read in a complete packet

   while (numBytesInPacketBuff >= sizeof(GameData))
   {
//    Call the routine to handle incoming game packets

      gameReceiveBroadcastData(serialPacketBuffer,(long) sizeof(GameData));

//    Shift the buffer to get rid of the processed data

      BlockMove(serialPacketBuffer + ➥
sizeof(GameData),serialPacketBuffer,numBytesInPacketBuff - sizeof(GameData));

      numBytesInPacketBuff -= sizeof(GameData);
   }


// Check if we should keep reading data

   if (doAsynchReceives == true)
   {
//    Queue up the next asynch read and try to get a full packet
//     or complete a partially read packet already in the buffer.
//     We could try to read extra bytes in case the next packet
//      contains multiple ENDPACKETCHAR characters, but then
//      again maybe it won't contain any and we'd be stuck
//       waiting for unnecessary bytes…

      myReadPB.ioParam.ioCompletion = (IOCompletionUPP) ➥
SerialReadCompletionRoutineASM;
      myReadPB.ioParam.ioRefNum = inputRefNum;
      myReadPB.ioParam.ioBuffer = serialPacketBuffer + numBytesInPacketBuff;
      myReadPB.ioParam.ioReqCount = sizeof(GameData) + 1 - numBytesInPacketBuff;

      errCode = PBRead(&myReadPB,true);

      if (errCode != noErr)
      {
//       Should probably post a Notification Manager event…
      }
   }
}
```

Outland

Unless you've been living under a rock for the past few years, you've undoubtedly heard of the much-hyped Information Superhighway or the Internet. In general terms, the Internet is the network of networks that interconnects computers around the world, making possible such things as global email, Usenet news, Internet Relay Chat, the World Wide Web, and FTP archive sites. Given the recent explosive growth of the Internet, it seems only natural that someone should come up with a way of using it to do something really important, like playing interactive games. This is exactly what Outland, Inc. has done.

To quote from the *Outland Development Guide*:

Outland is an affordable online service where people can meet to play Macintosh games. It is accessible through the Internet and from dialup sites across the country. By bringing players together, Outland provides game designers with a perfect medium for implementing multiplayer games. It is the fastest and easiest way to enter the online gaming marketplace.

Outland was designed to be utilized by Macintosh developers. Understanding that your spare time is scarce, we provide tools, code, and our personal assistance to help make your games available on our wide-area network.

We envision Outland as a network where the enjoyment of many Macintosh games is enhanced by the enthusiasm of real people. Through our combined efforts, we can turn the vast potential of online Macintosh gaming into a reality.

As claimed, Outland provides excellent support for their developers, including plenty of example code and thorough documentation. Additionally, the financial potential is excellent, since Outland offers royalties between 4 percent and 14 percent, depending on the originality, complexity, and required bandwidth of each game.

In this section, we'll look at the Outland Connection Library, the communications library used to develop games that work with the Outland wide-area network. If you wish to learn more about Outland, you can contact:

Outland
405 El Camino Real, Suite 224
Menlo Park, CA, 94025
1-800-752-9688
`info@outland.com`

The Connection Library

The Outland connection library is designed to provide a simple interface to the Outland connection protocols. The library allows you to create and debug a multiplayer game on a local AppleTalk network or even a single Macintosh. When the program is finished, it is automatically ready to work with the full-blown Outland wide-area network, with no need for source code changes or recompiling. The Outland connection library provides a

bidirectional, error-free byte stream, so it is conceptually quite similar to the Serial Driver we looked at in the previous section.

Because the Outland network is a wide-area network, there are appreciable delays in sending and receiving data. Outland suggests that the lag time may be as much as 200 ms coast to coast in the U.S., which, although it may not seem like much, means that real-time interaction is sluggish under worst-case conditions. Given the propagation delays inherent in the network, it is best suited to slower-paced, strategic games, rather than mindless shoot-'em-up arcade games.

Outland games must also be set up to run as a client/server applications, with the client being the game run on each player's machine and the server being the "referee" program run on the server at Outland. All player actions are routed through the server, which in turn informs the other participants of each player's actions. There is no direct communication between players, which is very different from the peer-to-peer interaction of Novel Netwar.

The Connection Library includes functions for initializing the library, closing the library, setting up connections, reading data from and writing data to the connection, and also performing maintenance functions on the connection. The individual functions are described in the following sections.

Initializing the Connection Library

To initialize the connection library, your program should call the `clInit()` function after all the usual Macintosh Toolbox managers have been initialized. The following code shows how this could be done:

```
OSErr     errCode;

   InitGraf((Ptr) &(qd.thePort));
   InitFonts();
   InitWindows();
   InitMenus();
   TEInit();
   InitDialogs(nil);
   InitCursor();


// Initialize Outland ConnLib for a client connection

   errCode = clInit(0);
```

The `clInit()` function is declared as:

```
OSErr clInit(unsigned long flags);
```

The lone parameter to `clInit()` is flags, which is a bitfield of which only 1 bit is currently defined; pass in zero if you are initializing a connection for a client program; pass in `kClIsServer` if you are initializing for a server.

The `kClIsServer` and other constants are defined in the ConnLib.h file that comes with the Outland Developer's Kit.

Allocating and Opening a Connection Record

Once the Outland Connection Library has been initialized, your program is free to allocate a connection record. A connection record contains internal information that the library uses to manage the connection. After the connection record has been allocated, your program should set the properties of the record and then open the connection. The following example shows how this can be done for a client application:

```
// Useful constants

#define SERVERSIGNATURE  "MyGame Server"
#define PLAYERSIGNATURE  "MyGame Client"


// Prototypes for relevant functions

static void connectionMade(connRef cr, ub4 refcon);
static void connectionBroken(connRef cr, ub4 refcon);


// Global vars

static connRef    connectionRef;
static char       connectionErrorMessage[256];


// Local vars

OSErr     errCode;


// Allocate a connection record

  errCode = clConnAlloc(kClLocalPort, 0, connectionMade, connectionBroken, ➥
&connectionRef);

  if (errCode != noErr)
  {
    sprintf(connectionErrorMessage,"clConnAlloc() error %d",(int) errCode);

    FatalError(connectionErrorMessage);
  }


// Set the name of the desired server connection

  errCode = clConnSetProp('dnam', SERVERSIGNATURE, connectionRef);
```

```
    if (errCode != noErr)
    {
        sprintf (connectionErrorMessage,"clConnSetProp('dnam') error %d",(int) errCode);

        FatalError(connectionErrorMessage);
    }


// Set the client name

    errCode = clConnSetProp('pnam', connectionName, connectionRef);

    if (errCode != noErr)
    {
        sprintf (connectionErrorMessage,"clConnSetProp('pnam') error %d",(int) errCode);

        FatalError(connectionErrorMessage);
    }


// Open the connection

    errCode = clConnOpen(connectionRef);

    if (errCode != noErr)
    {
        sprintf (connectionErrorMessage,"clConnOpen() error %d",(int) errCode);

        FatalError(connectionErrorMessage);
    }


// Activate the connection

    errCode = clConnAllow(connectionRef);

    if (errCode != noErr && errCode != destPortErr)
    {
        sprintf (connectionErrorMessage,"clConnAllow() error %d",(int) errCode);

        FatalError(connectionErrorMessage);
    }
```


The first step in setting up a connection record is the allocation of the record using `clConnAlloc()`. This function is declared as follows:

```
OSErr clConnAlloc(unsigned long flags, unsigned long refcon, connCallback onOpen, ➥
connCallback onClose, connRef *cref);
```

The parameters to `clConnAlloc()` are defined as follows:

`flags` is a bitfield of which only 1 bit is currently defined: Pass in `kClLocalPort` for a client to indicate that the connection will be visible on the local Macintosh only; pass in zero for a server to indicate that the connection is widely visible.

`refcon` is a user-defined value.

`onOpen` is a callback routine that points to a routine that is called when the connection is ready to begin sending or receiving data; the callback is made at noninterrupt time.

`onClose` is a callback routine that points to a routine that is called when the connection is terminated unexpectedly; the callback is also made at noninterrupt time.

The `clConnAlloc()` function allocates a connection record, which is the fundamental data type used by the Connection Library. A connection record must be allocated before any of the other useful library function calls can be made. The two callback functions are of type `void (*connCallback) (connRef cr, ub4 refcon)`, so a typical definition for the `onOpen` callback would be

```
static void connectionMade(connRef cr,ub4 refcon)
{
// Respond to make-connection event…
}
```

For the `connectionMade()` callback routine, the `cr` parameter is a pointer to the connection record that has connected, and `refcon` is the user-defined constant passed in as the second parameter to `clConnAlloc()`. Note that a server application allocates many connection records and its `connectionMade()` callback procedure must determine which one has connected before dealing with it appropriately, whereas a client application will typically have allocated only a single connection record, so there is no question which connection record should be dealt with.

After a connection record is allocated, additional properties of the record may be set using the `clConnSetProp()` function. This function is declared as follows:

```
OSErr clConnSetProp(unsigned long whichProperty, void *value, connRef cr);
```

The parameters to `clConnSetProp()` are defined as follows:

`whichProperty` is the property to set; currently defined values are `'wtra'` to control dumping of connection write data to a file, `'rtra'` to control dumping of connection read data to a file, `'pnam'` to set the name of the connection port, and `'dnam'` to set the name of the port to which the connection should connect (this last one is used only in client applications and should be the name of the server's `'pnam'` port).

`value` is a pointer to the new value for the property indicated by `whichProperty`; if value is NIL and `whichProperty` is `'wtra'` or `'rtra'`, then dumping of read/write data is stopped.

`cr` is a pointer to the connection record whose property is to be set.

Once the connection record is allocated and configured, the connection record can be opened using `clConnOpen()`. This function is declared as follows:

```
OSErr clConnOpen(connRef cr);
```

The single parameter to `clConnOpen()` is `cr`, which is a pointer to the connection record for which a port should be created.

And finally, once the connection has been allocated, configured, and opened, a connection can be established using `clConnAllow()`:

```
OSErr clConnAllow(connRef cr);
```

The single parameter to `clConnAllow()` is `cr`, which is a pointer to the connection record for which a connection should be established. Notification of the actual connection is made via the callback routine given as the `onOpen` parameter to `clConnAlloc()`.

The `clConnOpen()` function behaves differently, depending on whether it is called by the server or client application. On the client side, `clConnOpen()` actively initiates the connection to the server. On the server side, `clConnOpen()` passively waits for a client to connect. Particularly in the case of the server, `clConnOpen()` returns immediately rather than waiting for the connection to be established asynchronously, so your application will not block on a call to `clConnOpen()`. Note that a successful connection will result in a call to the `onOpen` callback routine specified as the second parameter to `clConnAlloc()`. This callback is made during execution of the `clConnIdle()` routine (see the section coming up about the `clConnIdle()` procedure).

Reading and Writing Connection Data

Once the Outland Connection Library is initialized and a connection record is allocated, configured, and activated, your program can read data from and write data to the connection. As mentioned previously, the Outland Connection Library implements a streamed connection, so data is sent and received as a stream of bytes rather than as discrete packets of data. Consequently, use of the Connection Library is more similar to the use of the serial device driver than the use of the AppleTalk Manager.

The Connection Library routines for reading and writing data are `clConnRead()` and `clConnWrite()`. These functions are declared as:

```
OSErr clConnRead(unsigned char *buffPtr, unsigned long *buffSize, connRef cr);
```

`buffPtr` is a pointer to the buffer where received data should be stored.

`buffSize` is the maximum number of bytes `clConnRead()` should try to read into the buffer. When `clConnRead()` returns, this contains the actual number of bytes copied to the buffer.

`cr` is a pointer to the connection record for which data should be read.

```
OSErr clConnWrite(unsigned char *buffPtr, unsigned long *buffSize, connRef cr);
```

The parameters to `clConnWrite()` are similar to the parameters for `clConnRead()`:

`buffPtr` is a pointer to the buffer containing data to be sent out the connection.

`buffSize` is the maximum number of bytes that `clConnWrite()` should try to send out the connection; the actual number of bytes sent is returned in this parameter upon exit from `clConnWrite()`.

`cr` is the pointer to the connection record through which data should be written.

Unlike a simple serial connection, a Connection Library connection is error-free (Outland's Connection Library is implemented using TCP/IP), so your application does not have to worry about lost data; however, it is still necessary to impose some sort of packet alignment on the data stream, perhaps by inserting end-of-message markers as Novel Netwar does for serial connections.

Idle Routines

Because much of the Connection Library's data transfer takes place asynchronously, the library needs to periodically perform internal maintenance and processing. To support this, your application must call the `clIdle()` routine regularly. Typically, this would be done in the main event loop of your program.

In addition to calling `clIdle()`, your program must call `clConnIdle()` once for each allocated connection. While `clIdle()` performs idle processing for the entire library, `clConnIdle()` performs idle processing for each connection.

The Connection Library idle routines are declared as:

```
OSErr clIdle(void);


OSErr clConnIdle(connRef cr);
```

The single parameter to `clConnIdle()` is `cr`, which is a pointer to the connection record upon which to perform idle processing.

If your application does not call `clIdle()` and `clConnIdle()`, connection callbacks will not occur, asynchronous send/receive processing will break down, and your application will eventually crash.

Closing and Disposing of Connections

Before your application terminates, it should close and dispose of all previously allocated connections. The routines for performing these operations are:

```
OSErr clConnClose(connRef cr);
```

The single parameter to `clConnClose()` is `cr`, which is a pointer to the connection record that should be closed.

```
OSErr clConnDispose(connRef cr);
```

As you might expect, the parameter to `clConnDispose()` is `cr`, which is a pointer to the connection record that should be disposed.

Calling `clConnClose()` breaks an open connection, notifying the other end of the connection of the impending break if possible. The connection record is still valid and may be used to open another connection by calling `clConnAllow()` again. Outland encourages programmers to use `clConnClose()` and `clConnAllow()` to reuse previously allocated connections rather than disposing of and allocating new connection records, since the latter method results in memory fragmentation.

The `clConnDispose()` routine closes the specified connection before deallocating the connection record. Each connection record must be disposed of with `clConnDispose()` before your application exits. If your program exits before disposing of allocated connection records, Outland warns that "the toolbox will not be cleaned up properly, and you will not be able to run your application again without rebooting."

Closing the Connection Library

In addition to disposing of all connections before terminating, your application should call `clExit()`. This routine is declared as:

```
OSErr clExit(void);
```

The `clExit()` routine prepares the Connection Library for application termination. If this function is not called, you will not be able to run the application again without rebooting.

Client/Server Communications

The Outland Connection Library is designed to support a client/server communications model, which means that you must write two different programs, rather than just the single program you would write for a peer-to-peer communications model.

The server program typically allocates a number of connections, listens for and deals with new connections and broken connections, receives and responds to incoming player "moves" or actions, and constantly informs all players of their game's status. The server can be very minimalist in its user interface, since only the developer and administrators at Outland will ever interact with it directly; however, since you will have to debug the server, you may wish to add support for displaying a log of its internal actions and status.

The client program will typically allocate a single connection record, attempt to connect to the server, allow the player to select a game to participate in, inform the server of the player's actions, and maintain the local player's view of the game's state as indicated by the server. Since the client application will be used by all players, you should concentrate your efforts on making its user interface as solid as possible.

Outland suggests that when you design your server, you include support for multiple simultaneous games. Similarly, your client application should be designed to allow the player to select from a list of games advertised by the server.

Finally, if you wish to avoid the whole issue of client/server communications, Outland can probably help by implementing the network protocols, server, and game lobby for you. Your responsibility would then be to add support for using High-Level Events to communicate between instances of the game. Outland used this approach with Spaceward Ho!, allowing its author to get the game up and running on the Outland network in only a few weeks. If you are interested in making an existing game compatible with the Outland network, this is the easiest approach to take, so you should contact Outland for more information.

Open Transport

The dilemma in writing the networking section of this book right now is that Apple plans to make everything you just learned obsolete. By the time this book goes to print, Apple will be shipping Open Transport; but at the time of this writing, Open Transport hasn't been released. Experience with it is limited to a few sneak preview alpha-test versions which aren't reliable enough to get any real work done.

Open Transport is Apple's new software interface for network programming on the Macintosh. It's important to understand that the networking protocols are unchanged—AppleTalk and TCP/IP are the same as they always were—but the way you write your program to use them is different. Correctly written programs will still be able to interoperate with each other on the network even if one is written using the traditional Device Manager parameter block calls that you just learned and the other is written using the new Open Transport calls.

Open Transport doesn't just change the names of the functions you call and the types of the parameters, it also changes the fundamental framework of how applications interact with the network software. To use it you may have to make significant changes to the way you structure your program.

Before you embark on this task, you should understand some of the background motivations and the difficulties and benefits that making the switch will bring you.

Why Apple Made Open Transport

Apple is introducing new networking software because of three factors: a need to update its networking software for PowerPC, a desire to unify support for different protocols, and a shift toward Open Standards.

When the original AppleTalk software was written in 1984, the Macintosh had 128K of RAM and a slow 8 MHz CPU. In order to give good performance, all the networking software had to be written in assembly code. Parameters were passed in registers instead of on the stack, and data was laid out in memory for optimal access by a 68K processor.

Apple's programmers succeeded in putting the whole thing together and for many years the Macintosh was the only personal computer that came with networking built-in. But with the

introduction of the Power Macintosh models in 1994, the old assembly code started to show its age.

C and Pascal code can simply be recompiled for the new PowerPC processor, but handwritten assembly code has to be rewritten by hand. Apart from the amount of work involved, some things just don't translate very well. How can you pass a parameter in register A0 on a processor that doesn't have a register A0? Clearly the specifications of the programming interfaces would have to be updated to make sense on the PowerPC.

Another thing that changed over those ten years was that AppleTalk was no longer the only protocol that people used on the Macintosh. MacTCP had been introduced, and just like the original AppleTalk team, the MacTCP team had invented their own set of programming calls without regard for any existing accepted standards. Programming for MacTCP is not any harder than programming for AppleTalk; it's just different. There's a different manual to read and different function names to call with different parameters.

Also waiting in the wings were MacOSI and MacX.25, which most people have never even heard of, much less used. OSI supporters claimed that the reason no one used them was that programmers didn't want to have to read yet another networking manual and learn another set of programming interfaces.

It was agreed that the next generation of networking software should be as uniform as possible. There should be one manual to read and one set of programming interfaces to learn, regardless of the protocol being used. Of course there will always be slight unavoidable differences between protocols, but they should be the rare exceptions, not the general rule.

The third factor that contributed to the design of Open Transport was Apple's nervousness about producing another piece of closed proprietary software. Apple felt that if it adopted an "Open Standard," more application developers would write for the Macintosh.

As a result of these influences, Apple adopted AT&T STREAMS and the X/Open Transport Interface.

The de facto networking standard in the Unix industry is BSD Sockets. Many people ask why Apple did not adopt the Sockets interface, since that has always been more popular, and current implementations of Sockets generally have better performance than their XTI/STREAMS counterparts. Apple has not answered this question, but one would imagine that since Microsoft has adopted Sockets, Apple felt compelled to adopt the other one. Apple engineers working on Open Transport have stated that they believe the performance problems of XTI are simply due to bad implementation and not due to fundamental flaws in the design.

What Is XTI?

Apple recommends that developers become familiar with the X/Open Consortium's book *CAE Specification, X/Open Transport Interface (XTI)* (ISBN: 1-872630-29-4); Apple also provides

its own Developer Note, which gives more specific information about Open Transport, Apple's implementation of XTI for the Macintosh. The Developer Note and other documentation is available on the Internet by anonymous FTP from `seeding.apple.com`.

The introduction on page 1 of the *CAE Specification* hints at the lengths that Apple had to go to in making XTI suitable for use on the Macintosh:

XTI is concerned primarily with the ISO Transport Service Definition.... XTI is UNIX version-independent.

First, although it was intended to be transport-independent, XTI doesn't guarantee not to lose some or all of the data you just sent. This is because the underlying transport protocol, OSI TP4, doesn't guarantee not to lose some or all of the data when you close a connection. That reliability guarantee was supposed to be provided by the OSI Session Layer, except that OSI was abandoned before anyone actually wrote a session layer for it.

Note

The OSI protocols *refer to Open System Interconnect, the official networking standard of the ISO (International Standards Organization).*

The second problem is that XTI was initially designed for UNIX, which has preemptive multitasking and memory protection. A UNIX program doesn't need to cooperate with other applications to share CPU time. If it wants to wait for a network connection, it can just call the listen routine (`t_listen` in XTI) and wait, forever if necessary, for a connection to come in. It is the UNIX kernel's job to notice that the program is blocked (not doing anything useful) and preemptively take control of the CPU from it to give to another application. Take that same program and run it on your Mac, and it's time to press the Reset button. So much for transparent portability.

The good news is that even when Apple introduces Open Transport, the old programming interfaces will still be supported for many years to come (so that all the existing applications will continue to work), so for the foreseeable future you are pretty much free to choose whether you want to use the traditional Device Manager networking calls or Open Transport.

The Features of Open Transport

The disadvantage of being the first to switch to new software is that you'll be alone. There's a wealth of sample code on the Internet showing how to use the current AppleTalk software, and a lot of experienced people on newsgroups who can answer your questions. If you use Open Transport, there won't be a lot of people who can help you, and you'll be the one finding all the early bugs for the rest of us. This is called being on the bleeding edge of technology.

On the other hand, the advantages are threefold: better performance, better features, and better longevity.

Better Performance

The traditional interfaces are in 68K machine code and always will be. This means that on Power Macs, you'll always be suffering the performance penalty of running network code under the 68K emulator.

However, performance may be something of a red herring. There are basically two kinds of network applications: defined requirement applications and best-effort applications.

A defined requirement application is the kind that the telephone companies like to deal with. They ask, "What is your application requirement?" You tell them how many bits per second you need and they say either yes or no. This is what happens when you pick up a telephone—the telephone needs exactly 64 Kbit/second, and the long distance lines either have enough free capacity for you or they don't, in which case you get the "Your call could not be completed..." message.

A best-effort application is something like file transfer or the World Wide Web. When you transfer a file, you don't decide in advance that you'd like it to take exactly 17.4 seconds. You just want it done as fast as possible. More to the point, when you upgrade your Mac from 10 MB/sec Ethernet to 100 MB/sec Ethernet, you like the file transfer to go ten times faster, but the traditional Device Manager networking calls will fail to achieve anything close to this kind of speed-up. When using 10 MB/sec Ethernet, the current Macintosh networking software is limited by the speed of the CPU, so putting in a faster network card will make little difference. It is at these speeds that the performance benefits of Open Transport will be noticed.

When users upgrade their Ethernet cards to 100 MB/sec, you don't want your game to go ten times faster. It should stay the same speed. A game typically has an almost fixed bandwidth requirement, and if you're sensible it should be less than 1/4 MB/sec or it won't work on LocalTalk (still the most popular network, especially in casual situations, such as a group of people getting together with PowerBooks to play a game). And if you can get it down below 14 Kbit/sec, then you have a chance of it working over modems too. At such low speeds, you're easily within the performance envelope of the Device Manager calls, so there is no reason not to use it.

Better Features

The old AppleTalk interface is frozen and Apple has no plans to improve it or even to fix bugs. All improvements in the future will be made only to the Open Transport code, so if there is some specific new feature you want, it's possible that it will be available only in the Open Transport calls.

One example of a new feature that is planned for the first release of Open Transport is *IP Multicast*, which is discussed in the final section of this chapter.

Better Longevity

At some point in the future, Apple will probably discontinue the Device Manager networking calls. How soon that is depends on how quickly everyone else converts their programs to Open Transport. But when they do, if your game is still using Device Manager networking calls, then it will not work on the latest machines with the latest System software (but that might be the year 2037, by which time your game might not work on new machines anyway because it doesn't support the multidimensional holographic virtual reality video board and the thought-controlled mouse).

However, the converse also applies. Most games have a shorter life span than other kinds of applications, so you might not care so much about longevity. In the short term, uptake of Open Transport is likely to be slow, given that Apple usually doesn't give away new system software for free, but sells it to recoup development costs. This marketing strategy will limit the number of customers who actually buy Open Transport for their Macs, which will limit the number of developers who will write for such a small audience, which will slow down migration from Device Manager to Open Transport and ensure that Apple has to continue supporting the Device Manager calls for many years into the future.

The Difficulties of Open Transport

The first thing you'll notice when you start looking at Open Transport is that it is actually three interfaces. One interface uses the standard XTI routine names such as `t_bind` and `t_accept` from UNIX and is included only for completeness. It is impractical to use the UNIX-style calls on the Macintosh for various reasons.

The second interface (the "Preferred C" interface) is a more Mac-like set of routine names such as `OTBind()` and `OTAccept()`. They return error codes as the result of the function call like other Macintosh system calls and support asynchronous operation more like other Macintosh routines.

The third interface is very like the second but implemented as a C++ class library (the "Preferred C++" interface). C++ programmers will love this, but unfortunately C++ run-time interfaces are not very well standardized yet and an unfortunate fact of life is that different compiler vendors handle C++ dispatching differently. So, you may run into problems programming to this interface unless you use the same compilers as the Open Transport team. The intent of the Open Transport team is to support SOM (IBM's System Object Model) when it becomes available, which should eliminate these problems and allow you to use any compiler you wish.

In addition to Open Transport defining Preferred interfaces, new calls have been added. XTI is stream-oriented, as is most of the current traffic on the Internet. Hence, XTI provides calls for connected "virtual circuit" streams such as are used for remote terminal login to UNIX computers. AppleTalk does have both a datagram protocol (DDP) and a stream protocol (ADSP), but unlike OSI or TCP/IP, it also has a heavily used transaction protocol, ATP. The vast majority of AppleTalk traffic is transactional—request/response exchanges using ATP,

such as reading files from a fileserver or printing to a LaserWriter—so Apple has added new calls such as `SndURequest()` and `RcvUReply()` to the Preferred interfaces in order to handle transaction-oriented communication.

Note
Purists will note that the Internet also has a transactional protocol, VMTP, but it is not widely supported.

Macintosh-like facilities have been grafted onto the existing XTI framework to make it not just usable, but efficient and powerful in the Macintosh environment.

In addition to the blocking/nonblocking option of standard XTI, there are now two more optional settings, AckSends/DontAckSends and Synchronous/Asynchronous.

Each of the three options can be set independently, making eight different combinations. Routines such as `SndUData()` (send a data unit) behave slightly differently in each of the eight modes.

Because UNIX has protected address spaces, when your program sends a message, the data first has to be copied from your application's address space into the kernel's protected address space before it can be sent. When you are running at 100 Mbit/sec or faster, the time to do just a memory-to-memory copy operation significantly slows down your performance. The AckSends/DontAckSends option allows you to tell Open Transport that instead of making a copy of the data and sending that, it should just use the data in place, the way the old Device Manager calls work.

The Synchronous/Asynchronous option allows you to have the call return to your program immediately and proceed asynchronously, just like setting the async flag on the parameter block calls. However, the familiar `ioCompletion()` routines are gone and instead there is a single event-handling routine. As long as your program does only one thing at a time, this is not too complicated. But if you make multiple network calls at the same time, then the event-handling routine will have to take special care to determine which call it is that's completing.

This also makes it harder for people writing libraries, because if your library makes some network call, then when it is complete, instead of calling a completion routine supplied by your library, Open Transport will notify the application's event-handling routine.

This proliferation of choices means that you have to be very careful to write extremely defensive code, making sure that you have covered all the possibilities.

For example, you might study the eight modes of operation of the `SndUData()` call and think you understand it fully, but when your program actually tries to send a small packet of data, nothing is received at the other end. That would be because you forgot to set the `XTI_SNDLOWAT` option. You might then change your program to set the `XTI_SNDLOWAT` option to make it work, but beware: Not all implementations have a `XTI_SNDLOWAT` option. It is optional.

The best example of this difficulty of coping with all the unexpected behaviors of XTI is the `kOTLookErr` error code.

The `OTLook()` Nightmare

The fundamental paradigm of who controls the flow of program execution is changed in Open Transport. In conventional Mac programming, the application tells the system what to do and, if it's at all possible, the system does it. If you want to read some data from a network connection (or from a file), then you do a `PBRead()` call, and when the call completes the System calls your `ioCompletion()` routine to tell you. In the new software, Open Transport tells the application what to do and when. When it tells you to read, you read. It won't tell you twice.

In most Macintosh System software, an error code means exactly that—an error. It means that some unavoidable circumstance prevents the system from carrying out your request, or sometimes it simply means that you made a mistake in your program.

In Open Transport, there is a special error code, `kOTLookErr`, that neither means you made any mistake, nor means that your system call cannot be performed. It just means that the networking software doesn't want to perform the operation right now. It wants to do something else instead. I call it the "won't" error code. When you call a routine and get the "won't," you are required to call the routine `OTLook()` to find out what Open Transport requires you to do.

Here's an illustration from the UNIX world. You want to write a program which offers some TCP service on the network, so you have to listen for incoming TCP connection requests and respond to them. An example of this kind of service is the UNIX finger program, which allows you to ask a UNIX computer for personal information about a user, such as the last time he logged on and the last time he read his email.

You might think you would code a finger server something like this (a very simple toy example, ignoring error handling and the niceties of cooperative multitasking):

```
static void FingerServer(EndpointRef ref)
  {
  while (1)
    {
    EndpointRef newRef;
    OSStatus errorcode;
    TCall call; // initialization of all the fields omitted for brevity
    TEndpointInfo info;

    // listen for an incoming connection
    errorcode = OTListen(ref, &call);
    if (errorcode != noErr) return;

    // create new local endpoint to communicate on
    newRef = OTOpenEndpoint(OTCreateConfiguration("tcp"), 0, &info, &errorcode);
    if (errorcode != noErr) return;
```

```
    // accept the connection
    errorcode = OTAccept(ref, newRef, &call);
    if (errorcode != noErr) return;

    // call my routine to respond to the service request
    send_finger_response(newRef);

    // close the new local endpoint
    errorcode = OTCloseProvider(newRef);
    if (errorcode != noErr) return;

    // return to beginning of while loop to await next incoming request
    }
  }
```


It looks like a nice simple piece of code. Even if you put all the error checking in, it wouldn't be that big, would it? The problem is that it might seem to work in light testing, but in real use it fails. Why?

In practice, the `OTAccept()` call frequently returns the error code `kOTLookErr`. Investigation shows that this tends to happen whenever two different people try to open connections at about the same time. You then have to call the routine `OTLook()` to find out what tribute XTI wants you to pay before it will honor your request.

I imagine the dialogue going like this:

```
Program:  "OTAccept please."
XTI: "Won't."
P: "Why?"
X: "New connection request just came in."
P: "Well, can't it wait until I've finished with this one?"
X: "No."
P: "I'll only be a millisecond. Promise."
X: "No."
P: "Why?"
X: "Who's the boss?"
P: "Okay. OTListen then."
X: "Thank you."
P: "Can I OTAccept now?"
X: "You can try."
P: "OTAccept please."
X: "Wrong one!"
P: "What?"
X: "You have to OTAccept the other one first, not this one!"
P: "All right. The other OTAccept please."
X: "Wrong again!"
P: "What?"
X: "You have to remember all the information I give you from every OTListen call, and
then give it back to me in the right order for each OTAccept call. I hope you have a
big array of structures."
```

```
P: "I do. Here you are. OTAccept with bells on."
X: "Won't."
P: "What! That was a perfect call."
X: "True. It was. While we were talking, a third person did a finger request."
P: "Okay. You can't fool me that easily: OTListen."
X: "Well done! What now?"
P: "OTAccept with bells on."
X: "Won't."
P: "What? A fourth person?"
X: "No, the first person got bored waiting and closed his connection."
P: "So what am I supposed to do now then?"
X: "You have to tell me to do an OTRcvDisconnect."
P: "With bells on, I suppose?"
X: "You're learning."
```

The lesson here is that there a are lot of pitfalls in using an XTI-based interface. You may be better off sticking with the tried-and-true Device Manager interface, unless there is some special feature of Open Transport that you really need.

Open Transport does have one special feature that game writers could really use, and that is IP Multicast.

IP Multicast

IP (Internet Protocol) Multicast allows you to send a single packet to a large group of computers, such as the DDP broadcast used in Netwar. Unlike broadcast, multicast is selective, going only to those computers that have expressed an interest in receiving it. (The third member of the network trinity is unicast, the most common form of network communication, in which packets are addressed to just a single destination machine.)

As you saw in Netwar, broadcasts are an easy and effective way of communicating information, but broadcasts have two major drawbacks.

First, broadcasts go to every computer on the network, whether or not they are playing your game. On some computers, receiving packets involves a lot of overhead, and a high rate of broadcast traffic can consume all of the computer's available CPU time just examining unwanted broadcast packets and discarding them. When the popular IBM PC game Doom first came out, the broadcast storms it produced were so severe that slower computers were completely crippled and many places had to ban the playing of networked Doom. (Although that may not have been the managers' only reason for banning it.)

Second, in order not to make this problem any worse, most network routers do not forward broadcast packets off the network on which they originated.

On a small Ethernet or LocalTalk network running AppleTalk, a controlled amount of broadcast is acceptable, but there are millions of computers on the worldwide Internet, and if you use broadcast, then you are limited to playing with only the people on your own network segment.

IP Multicast is the tool that lets the Internet provide communication between groups of computers spread all over the world without collapsing in a catastrophic broadcast storm.

Using IP Multicast

Internet addresses are usually written as four numbers, separated by dots, like this: `36.28.0.25`.

Internet addresses in the range of `224.x.x.x` to `239.x.x.x` are called multicast addresses. These addresses don't address an individual computer, but a group of computers that share some common interest (like playing your game).

When your game starts up, you join the multicast group for the multicast address that you are going to use. Then, any time anyone sends a UDP datagram packet to that address, you will receive a copy, just like broadcast packets.

How far the packets go is determined by the number you put in the "time-to-live" field of the packet when you send it:

- Multicast datagrams with initial TTL 0 are restricted to the same host.
- Multicast datagrams with initial TTL 1 are restricted to the same subnet.
- Multicast datagrams with initial TTL 32 are restricted to the same site.
- Multicast datagrams with initial TTL 64 are restricted to the same region.
- Multicast datagrams with initial TTL 128 are restricted to the same continent.
- Multicast datagrams with initial TTL 255 are unrestricted in scope.

This is great except for one small thing: IP Multicast is still largely experimental.

The host software for managing group membership has all been worked out and is now available on every major computer platform. The router software is mostly worked out, and the famed "mbone" (multicast backbone) that you may have read about in the press has been operational for a couple of years now, carrying regular video and audio transmissions. What has not been worked out is how to *use* it, and in particular what addresses to use.

At present, IP Multicast addresses are assigned by hand by IANA, the Internet Assigned Numbers Authority. It is obviously not practical for game players to call IANA to get an address assigned for them every time they want to play a game. (For more information on Assigned Numbers, see the Assigned Numbers RFC [Request For Comments], currently Internet RFC 1700, available from your favorite RFC FTP archive or via any World Wide Web search site.)

What is needed is some way for programs to get a temporary address assigned automatically for the duration of the game. There is a proposal to do this dynamically when the game starts. A block of 2^{24} (16,777,216) IP Multicast addresses will be reserved for use by multiuser collaborative applications (games). When your game starts up, it is free to pick any address at random within that range to use as its multicast group address. There is no guarantee that

some other program will not pick the same address, in which case your program will receive both sets of multicast packets, but the situation is the same when you use broadcast—there is no guarantee that some other application on the network won't be using broadcast at the same time. In either case, you should put some identifying pattern in your packets so that you can detect which are your own and which belong to another application (or to another instance of your own game running on the same network). When your program detects packets belonging to another application, it can simply ignore them, or if your program is really sophisticated, it could automatically switch to another randomly chosen address.

The final caveat to all of this is that although most computers are multicast-capable these days, not all networks are. Unless your network routers are multicast-capable and your network is connected to the worldwide mbone, your multicast packets will be limited in scope, probably to the physical network you are connected to, just like broadcasts.

Nevertheless, IP Multicast deployment and development is proceeding rapidly and offers exciting new possibilities. Read the `comp.sys.mac` Internet newsgroups and browse the World Wide Web to keep abreast of the latest developments.

By now you have a good idea of what's involved with adding networking capabilities to a multiplayer game. We've looked at everything necessary to use AppleTalk's NBP, ATP, and DDP services, including asynchronous use of these services. Also, we've looked at direct or modem-based serial connections, again with emphasis on the use of asynchronous calls for maximum performance. We've also looked at the Outland Connection Library and its support for Internet game-playing via TCP/IP streams. And finally, we've taken a brief introductory look at Open Transport and the pain and promise it holds for the future.

If you're still interested in learning more about networking, you should probably look at the source code for Novel Netwar, included on the CD-ROM that accompanies this book. You may also wish to look at the documentation available from Apple listed at the end of this chapter.

Now, good luck, and start programming!

Other Sources of Information

For more information on the various forms of networking available on the Macintosh, consult the following sources:

Inside the Macintosh Communications Toolbox, published by Addison Wesley (ISBN: 0-201-57775-5). This book describes the Applications Programming Interface to Apple's defunct Communications Toolbox. The Communications Toolbox has been abandoned in favor of Open Transport.

Inside Macintosh: Networking, published by Addison Wesley (ISBN: 0-201-62269-6); also available via anonymous FTP from `ftp.info.apple.com` as `Apple.Support.Area/Developer_Services/Technical_Documentation/Inside_Macintosh/Inside_Macintosh/Networking/Networking.sit.hqx` (the index is in the same directory but is called `Networking.idx.sit.hqx`). This book is an extremely thorough guide to AppleTalk.

Inside Macintosh: Devices, published by Addison Wesley (ISBN: 0-201-62271-8); also available via anonymous FTP from `ftp.info.apple.com` as `Apple.Support.Area/Developer_Services/Technical_Documentation/Inside_Macintosh/Inside_Macintosh/Devices/Devices.sit.hqx` (the index is in the same directory but is called `Devices.idx.sit.hqx`. This book includes a thorough guide to the Macintosh Serial Driver.

The Outland Development Guide, available from Outland, 405 El Camino Real, Suite 224, Menlo Park, CA, 94025, 1-800-PLAY-OUT, `outland@aol.com`. Outland is an Internet-based online gaming company that offers excellent support for developers of multiplayer games. It offers royalties of 4 percent - 14 percent on games it licenses from developers; Outland is definitely worth contacting.

The MacTCP Programmer's Guide, available via anonymous FTP from `seeding.apple.com`. This document describes the programming interface to MacTCP, the implementation of TCP/UDP/DNS/IP for the Macintosh. If you are interested in playing games over the Internet, then you should get this document.

Profile: Bill Dugan

Bill is the "Macintosh High Priest" at MacPlay, which is the Macintosh group of InterPlay. He has overseen the development of MacPlay's entertainment titles—over 20 so far.

Q: You work at MacPlay and you're the Mac guru there. What exactly is your position?

A: The job title is Macintosh High Priest, and that's because we favor humorous job titles around here. Generally, I oversee the development of all the Mac games. I work with the producers who work with the programmers and I make sure that all of our games are going to be good enough for the demanding Mac market. On the development side, that's the most important thing: making the calls on what we need to do with a Mac game, deciding what technology we want to do, etc. Also, human interface problems—that's easy to make decisions on.

Q: What sort of games that MacPlay has put out have you had a hand in?

A: All of them. The MacPlay games are Out of This World (which I did half the programming on), Battleships, Battleships Enhanced CD-ROM, Bridge Deluxe with Omar Sharif, Mario Teaches Typing, Star Trek 25th Anniversary, Star Trek 25th Anniversary Enhanced CD-ROM, Castles: Siege and Conquest, Alone in the Dark, Flashback, Wolfenstein 3D, Mario's Game Gallery, SimCity Enhanced CD-ROM, Mario Teaches Typing Enhanced CD-ROM, Flashback CD, Power Pete, and there's all kinds of other games coming up. There's also some other kinds of

continued

titles—MacPlay publishes Monopoly and Legend of Karandia, Caesar's Palace, Risk, and Scrabble, which are all titles that were originally programmed by Virgin Games. That was a situation where MacPlay took the games and enhanced them a little bit and put them out under the MacPlay label. We're also the distributors of Explorer 1, which is Peter Gabriel's thing, Astrochase 3D, and that's it.

Q: And you mentioned that somebody there is working on Descent.

A: That's right. Descent is one of our many upcoming products.

Q: For a Mac game, what's the size of the development team that you use?

A: Typically, our teams consist of one programmer—the most consistent attribute is that typically there's one programmer, one guy who does the music, one guy who does the sound effects. Those are both very important components, and typically, those are done by one person. And as far as art goes, there is usually more than one artist. It's difficult to track artists because artists can come and go. One artist might do the backgrounds and one might do the sprites and another artist might do miscellaneous tinkering of logos, but on average, three artists might work on each game. And often the music and sound effects person is the same, but sometimes not.

Q: Why are they porting these games to the Mac platform, because typically this is a smaller platform?

A: From the point of view of the big publishers, the Mac is indeed regarded as the secondary market. The way that the big publishers look at the Mac is that they do the DOS version first and then the Mac as a way to get additional revenue off the development investment that they've already made. For instance, if you take one of InterPlay's games that we did ten years ago—a game called Mind Shadow for Activision, an adventure game—we did it on the Apple II and the Commodore 64 and we did it on the IBM and the Macintosh, which was only black-and-white at the time. And each of these platforms was just regarded as another incremental piece of revenue that we could get. We didn't give any thought to user interface—taking advantage of the platform was something that was desirable, but it certainly wasn't harped on in the way that we really have to do now in the Mac world. Mac people are going to go nuts if you directly port a DOS game, and you know they're going to hate you.

I think the reason that there's so much interest in the Macintosh is because of the Power Macintosh, especially considering all of the things that Macintosh owners have that DOS people have as an option. Macs have local bus video so the fast video can really speed up the games, and Mac owners have 640 x 480 monitors. That was a pain in the butt on the PC until a few years ago. Mac people have great sound. Their computers have a lot of things that are attractive to programmers, so from a developer's sense, everyone's excited about the PowerPC, and Macs are good machines. From the business side, Macs are

viewed as an incremental sales generator by most companies. There are some companies that really care about the Mac and try to make good games because there's a lot of money to be made in the Mac market. It's a much smaller market than the DOS market, but there's still quite a bit of money to be made.

Q: How about the expanding home market, where Apple says it is number one in that area? Would the growth of the home market, with the Mac being the lead platform, be a factor in these business calculations, or is it a niche machine?

A: I'm afraid that the niche machine mentality and "Let's make a version for the Mac" is probably the main factor that a lot of companies are using. The leading home market for computers is obviously still DOS. It's still 90 percent of the market. That 10 percent of the market is a huge market. People used to think that Mac people didn't want to play games, but they have changed that. They're really supporting the game developers now.

Q: What are they doing to support the game developers in terms of technology?

A: I think that QuickDraw 3D is going to be very important for the future, not running in software alone because it's too slow, but if there's a hardware accelerator card, the games are going to be bitchin'. We're planning for Descent QuickDraw 3D. It's going to be too slow if the user doesn't have a video card that actually draws texture map polygons for you. QuickDraw 3D and software is much slower than we can do; Descent's engine is much faster than QuickDraw 3D because the Descent engine was made to run Descent. If there's a video card that will draw polygons really fast, QuickDraw 3D will be wonderful.

I think that's probably Apple's most important technology coming up in the near future. QuickTime has been important mostly just to save us the work of having to write a movie player. And QuickTime also offers compatibility with all the Mac models, which saves quite a bit of back-end time during the Q/A cycle.

Q: Does it help with animation, or is that more a matter of custom software or just sprite stuff?

A: Yeah, we run our own animation drivers for everything. QuickTime's too slow. Again, it's a situation where QuickTime is a very generalized architecture which is incredibly useful and it's also useful for playing back video, but as far as drawing things on the screen, we're very good at doing that and we always roll our own.

Q: What are some of the things you do to boost performance?

A: You mentioned what Apple has done for game developers. QuickDraw 3D is important. QuickTime is also important, but for business purposes, Apple's doing a great thing with game kitchens. They put on game kitchens several times a year. We've gone to a couple of them. The whole idea is that Apple engineers sit around and help us optimize our code. They invite all the game

continued

developers to come. We're working on an upcoming Star Trek project. We were able to take the PC code and get it up and visible on the Mac in about a day and a half, and Apple's engineers were helping us optimize stuff and work with QuickDraw 3D. It was great. There's an upcoming game kitchen in Chicago and the Descent people from Paradox will be working with people there. I think that's a wonderful step for Apple to do. It's greatly helped us optimize our games for the Macintosh platform.

Q: What are some of the optimizations you do—in case of the rendering engine for Descent. Use this as an example. What do you do to roll your own for various game components?

A: Unfortunately, I have to weasel out on that question because it completely depends on the game. Descent is a 3-D engine that's used to draw shaded, texture-mapped, lit polygons, whereas another game like Power Pete is a much simpler two-dimensional game that just has to draw a lot of sprites and tiles very, very fast.

Power Pete is a good example of some of the optimizations that we were able to get. It's kind of within two stages. At first, Power Pete played in a window that is about half of the screen, and if you haven't seen Power Pete, the demo is going on the CD-ROM for this book. Power Pete is a game in which Power Pete runs around, and it's kind of like a topped-down Nintendo game—in the manner of the zombies ate my neighbors—where there's a lot of land that scrolls around very quickly on the screen. So we're moving a lot of data. Initially, Power Pete was only able to run in about half of the screen size—more of a vertical thing toward the left half of the screen. Then the programmer, Brian Greenstone, went up to one of the Apple game kitchens and the engineers showed him some tricks that he used with the PowerPC processor, and the trick was that on 601 processors it can be really efficient to use the floating point registers rather than using 32 bits at a time. You can use 64 bits just by using the FPU registers. He wrote a blitter that used the floating point registers so he was able to blit twice the amount of data at a time, and Power Pete became a full-screen game. But then, the funny thing is that as soon as the 603 and the 604 came out, Power Pete ran really slow, because the 603 and 604 have much bigger problems with misaligned data accesses than the 604 does. After we got a 604 as a seed unit in here, we were able to diagnose the problem, and then he did a different version of the blitter that runs on 604 and 603 machines which does a couple of tricks so that he's always accessing the data on even boundaries. He's not doing misaligned data access, so Power Pete's fast again on the 603 and 604.

Q: How much of a slowdown on the misaligned?

A: After he rewrote the blitter, the game speed more than doubled. The problems on the 603 and 604 with misaligned data access are going to cause a lot more problems than anybody ever thought.

Q: You seem to be doing ports. What sort of play testing do you do to know if the interface is good or rotten, or maybe the game rules should be tweaked?

A: There is some controversy about that and it has to do with focus groups. A lot of companies put a lot of weight on having focus groups where they take a bunch of kids and they sit them down in front of the game and let them play. We do focus groups occasionally; there's a place for it, but kids are certainly not game designers. Generally, we rely on our own instincts for what's good in a game and what isn't. I mean, if there's a game that's fun enough for us to play, then it's going to be fun. If there's a game that we hate and we just finish the game and get it out there, then obviously nobody's going to like it. We're a bunch of game nerds anyway, so we know what makes a good game.

It used to be that port always meant bad port; that's really not true at all. Castles: Siege and Conquest was a port from IBM, but in reality most of the ports had to be redone. Castles is a wonderful Macintosh game and it wouldn't have been any better if we had done it from scratch on the Macintosh. Wolfenstein 3D is also a port.

We have a Macintosh Q/A department, and they sit around and play the games to death and hunt bugs. Everyone is on a network with a variety of hardware. Typically, we give them the product when we think it is done. They send it back with problems and we fix them. To a large extent, Q/A deals with design issues. We take feedback and roll it back into game design. We don't really use external beta testers.

Q: What sort of development tools do you use?

A: Just about everything is Metrowerks these days. We switched to it when Symantec didn't have a Power Mac compiler. MPW we leave around for doing certain work, but generally only use Metrowerks. On certain internal projects that have a time frame of a year, we're doing simultaneous versions of Mac and PC. Then we're using WatCom.

Q: You use a core code-type development? And the interface like DOS or Mac is the custom part of the game?

A: On our system there's a core system and even a core windowing system. There's also some other exceptions, because Frankenstein is a situation where they're doing everything in Director and Lingo. We also use out-of-house contractors.

Q: How often do you use assembly? Is that a last resort for performance?

A: Typically, you want the innermost loops of very time-dependent code to be in assembly. You would be out of your mind to write and entire game in assembly. It just doesn't make sense.

continued

Q: You do profiling to determine the key parts?

A: Yes. Standard programmer's tool.

Q: You mentioned that some developers roll their own version of `CopyBits()`.

A: Every programmer thinks he can beat `CopyBits()`. Every programmer writes his own. Typically, `CopyBits()` is pretty good, but it depends.

Q: Even though PowerPC is attractive, there's still a lot of 68K computers out there.

A: Currently, everything we do is a fat binary. It will be some time before 68K machines go away. Descent will be a fat binary. I don't like having separate binaries; I prefer everything in one application.

Q: Do you write a 68K with 68K assembly?

A: Hopefully, the same code will work correctly when compiled on 68K. As far as writing, we just leave the assembly version and then put it in PowerPC.

Q: Do you see the market changing any? Harder or easier to develop games for Mac?

A: Game development is becoming more expensive. To beat the competition, you have to pour more resources into the game. It has to look better and be a really great game. A really great designer can make a really great game on a shoestring. If you're going to stand out, then you need to have a knock-out game with a lot of art, sexy graphics, great sound, animation, and blow people away with experience.

Another issue is shelf space. Mac shelf space is a drastic minority in a typical retail store. Gets a shelf or two. That's difficult because we have to compete against each other and IBM. Lot of pressure to make the game as flashy as possible and not to release it unless it will be a hit.

Q: Do you think that will change or will people keep upping the ante?

A: We're still at a stage where a couple of guys in a garage can blow people away. They did it with Doom. As time goes on, it will be more difficult. Look at movies—little movies that cost very little often have their own niche. It's not like we're going to get crushed. We also have shareware distribution going for us. Doom had direct distribution.

Q: Talk about some of the Star Trek games. Did these have animation or QuickTime movies?

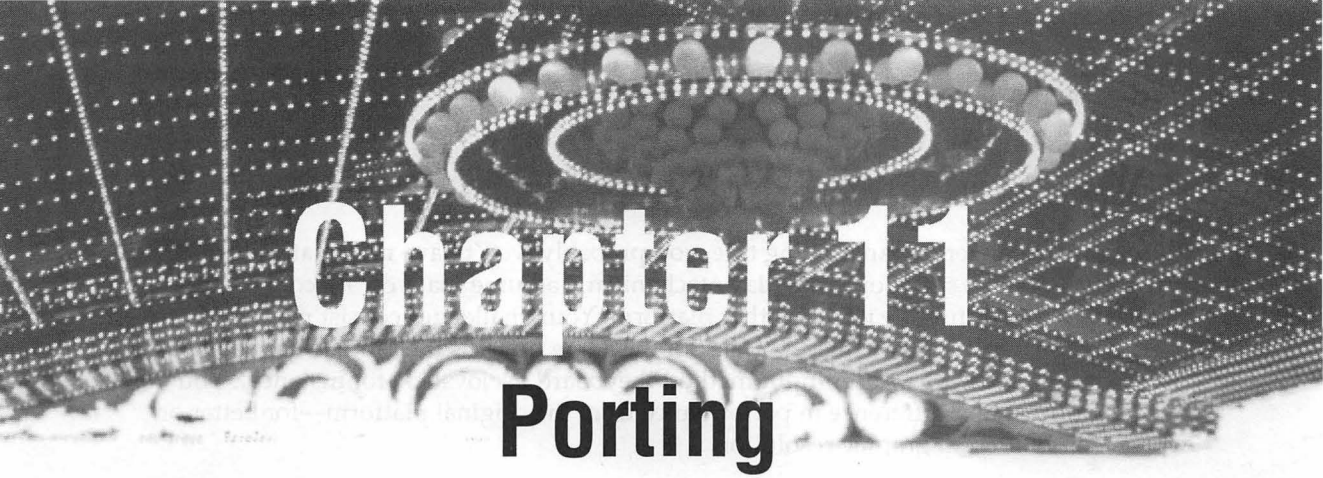
A: For the CD-ROM Star Trek 25th Anniversary, the whole game was animated. Got the original actors and recorded voices to record over the cartoons. An animated game with voiceover, rather than video. Some people are doing a great job with lip synching—we didn't worry about it with Star Trek.

Q: Any thoughts about QuickTime VR?

A: QuickTime VR is a good technology for certain types of products. Peter Gabriel Explorer 1 had a primitive version of QuickTime VR, and if QuickTime VR had been available, it would have been used. As far as fast-action, 3-D games, I don't think QuickTime VR has much application. QuickDraw 3D excites me a lot more as far as action games go.

Q: Any final thoughts?

A: Mac gaming will continue to go strong for many years in the future. The Mac is the best platform there is, but we don't have the most market share, and for that reason everyone is making these games on the PC. But we know we have the best platform and hopefully game developers will continue to make great games for the Mac. We're no different than the PC people. We want a similar experience. We're just more sensitive to bad interfaces. The Mac gaming market will continue to grow and looks great for the next five years. I can't wait for accelerated polygon video cards.



Chapter 11
Porting

The first section of this chapter will show you what work generally has to be done in a game conversion and how to do it. The following section offers solutions to selected problems you may be facing in your particular conversion.

Approaching the Conversion

More games have been converted to the Macintosh in the last couple of years than ever before. With the recent success of the Power Macintosh and the looming presence of Macintosh clones, an even greater variety of entertainment titles will be converted to the Macintosh over the next several years. More programmers than ever will be converting and "porting" games to the Macintosh and Power Macintosh.

Game conversions or ports are judged by users in several areas, including:

1. *Game play*—Is it fun?
2. *Graphics*—How does it look?
3. *Sound*—Does it sound great?
4. *Interface*—Is it Mac-like?
5. *Technical prowess*—Is it fast? Cool? Innovative?

Each of these areas is important for delivering a commercial-quality game, and each must be evaluated when approaching an existing title. You should address each of these issues during the planning of the conversion. This allows you to define the extent of the work to be performed during the conversion. Once defined, the work can be scheduled.

A clear definition of the work to be performed is crucial to a successful game conversion and perhaps one of the more overlooked aspects of software development. Make sure you know what is expected, and make sure it is clearly defined in any contract you sign.

Game Play

Since you are converting an existing title, you probably won't have much latitude in changing the basic game, story, or game play mechanism. I assume that you are converting an immensely successful title from another platform. Your challenges consist mainly of converting the game in such a way that the "feel" of the game remains the same in spite of the difference in hardware or user input method (keyboard vs. joystick, for instance). You may need to address any difference in performance from the original platform—for better or worse—or a different graphic resolution.

Any changes you make that affect the game play affect the basic nature of the game. Make certain that any "improvements" you make to the game really are improvements. I suggest that you solicit opinions from game players you know and then listen to the opinions of the majority.

Graphics

Different hardware platforms have a variety of different graphic resolutions that are available. The most common resolution for DOS games, for instance, has traditionally been a mode called "13H," which results in a screen display of 320 pixels wide by 200 pixels tall. Directly converting a game using that resolution results in a very small window on the Macintosh that more closely resembles a postage stamp. The other interesting characteristic of "13H" is that the pixels are not square—they are vertically oblong, and when viewed on the Macintosh they look a bit "squashed."

With the advent of Windows, however, more and more products are being released with a square aspect ratio. Those games can be acceptably ported, or when appropriate, pixel doubled to the Macintosh screen. That is, a 320 x 240 image can be doubled to 640 x 480, filling a standard 13", 14", or 15" Macintosh monitor. Consider supporting the smaller 12" color monitor sold with early LC models, especially for educational or children's games. Even the built-in monitors of the various Performas, Color Classics, and PowerBooks should be considered in terms of pixel resolution.

The 12" monitor that was popular when the LC was first released has a resolution of only 512 x 384 pixels, the same as the resolution of the Color Classic's built-in monitor. PowerBooks, depending upon the model, are almost always 640 x 480 pixels, except for a few models that change resolution when switching to 24-bit (Thousands) mode. In that mode, several displays allow a resolution of 640 x 400. By supporting a resolution of 512 x 384 pixels in 256 colors, your program will support every color monitor Apple has ever made.

It's often necessary to resize graphics that are used on other platforms to correct the aspect ratio, or even to increase the resolution of the graphics. For an original Macintosh game, the current standard of high-quality games is a resolution of 640 x 480 pixels (the menu bar has to be hidden to obtain those dimensions on most monitors). For conversions of high-performance or arcade-style titles, a pixel-doubled 320 x 240 graphic resolution can be successful as well. Eric covers some of the more sophisticated problems and solutions in his half of this chapter.

Sound

How a product sounds is often just as important as how it looks and plays. For fast-action racing or sports games, as well as arcade-style games, sounds are practically all-important. Background music and sound effects are commonplace on all platforms and crucial to a commercial-quality game.

Traditionally, a continuous, looping MIDI soundtrack has been used on most game systems and PC compatibles. Until recently, very few options for playing MIDI scores existed for most Macintosh users. Although Mac sound cards that support MIDI have been available for several years, their lack of popularity in the user community has forced game programmers to find other solutions. With the advent of QuickTime 2.0, however, a MIDI (Musical Instrument Digital Interface) interpreter—implemented completely in software—has provided a mechanism to use music and sound without the high memory overhead of digitized sound.

Compare, for instance, a digitized sound at 11 KHz that requires about 11 kilobytes per second of memory, versus 15 or 20 kilobytes for several minutes of MIDI music. This is a somewhat simplistic comparison, because it does not take into account the overhead associated with QuickTime or the instruments that QuickTime loads into both the System and application heap zones. Nevertheless, in some instances, the amount of memory saved can be substantial.

For sound effects, the Macintosh has always supported digitized sound samples. Even playback-time software decompression is supported by the built-in Sound Manager, requiring very little effort on the programmer's part. The Macintosh is also capable of playing a number of sounds simultaneously, mixing mono and stereo sounds automatically, regardless of whether the user's Macintosh has hardware stereo capabilities.

Interface

Every programmer converting a game to the Macintosh must eventually make several decisions that are crucial to the converted product's success in the Macintosh user community. These decisions lie in several key areas that make Macintosh programs unique. The most important decision, affecting both the technical and financial aspects of the conversion, is this:

Will the converted program be a "real" Macintosh application?

Practically any program ever written in C, Pascal, or any other popular programming language can be easily converted to the Macintosh. Thousands of programs have been ported to the Mac over the past ten years. A good percentage of these programs, however, were converted by programmers unfamiliar with the Macintosh or unwilling to learn how to write a "real" Macintosh application.

A "real" Macintosh application isn't just a DOS program with a menu bar and Macintosh windows. The differences go far beyond the screen layout; they affect a program's entire structure. Let's examine the basic requirements of a real Macintosh program.

The Macintosh Difference

Programs that run on the Macintosh are profoundly different from typical Sega or Nintendo games. Obviously an eight-button control pad is much different from a keyboard and mouse combination, but the differences go even deeper. A Macintosh program—even a game—should be prepared to format a blank disk if one is inserted (even in the middle of a hairy dogfight at 30,000 feet!). It should allow the user to switch back to Excel to continue sorting a 40,000 line spreadsheet if the boss walks by.

The single biggest difference, though, has nothing to do with the System software or the hardware. It's the user—and his or her expectations.

As a Macintosh user since the mid-1980s (when the 512K Mac was first released), I have a number of expectations of any program installed on my Macintosh. My expectations are very high, and those programs that do not measure up are dragged to a small icon in the lower right of the desktop with the word "Trash" printed clearly underneath it.

For instance, any program that does not respect the time-honored tradition of using Command-Q to mean Quit is practically begging for the trip to the trash can. A program that crashes on startup because QuickTime is not installed is immediately shown the way to the dumpster. A game that insists I adapt to a set of keystrokes and gives me no way to customize it isn't Mac-like, and I probably won't play it.

Each and every part of a game carries with it a number of expectations on my part, and for Macintosh users like me. I expect to be able to play network games over AppleTalk on my PowerBook while sitting in an airplane connected to another PowerBook by a serial cable. I expect the program to inform me when it requires SmartDraw Version 1.03 and doesn't find it installed in the System folder. I expect the program to let me know when a file it's looking for cannot be found. In other words, I expect a real Macintosh application to make almost no assumptions about the software and hardware that it runs on and a lot of assumptions about me. After all, as the consumer, I'm the one who will buy more games from you—or not.

Learning about the Macintosh interface and the user's expectations is the first step in converting any program to the Macintosh. This is even more important when the program is a game. After all, you are trying to entertain, not frustrate, the user. The user's expectations are all-important if you expect him or her to play your game over and over—or even once!

A good resource for learning about the interface requirements and user expectations is *Human Interface Guidelines* by Apple Computer. This book explains how a Macintosh program should behave and how you as a programmer can make sure that your game does not end up making the short trip to the trash can, and maybe back to the store.

If you are not a Macintosh user, go find one. Or two. Then ask a lot of questions. Find out what an average Mac user's expectations really are. You might be surprised.

And play some games on the Mac yourself!

The Mac Top Ten Expectations

The following are my top ten expectations about a program that has been converted to the Mac. I hope they help you understand the basic demanding nature of a Macintosh user.

10. I want to use a keyboard and a mouse, or both, depending upon my mood. I might even have a joystick, trackball, or optical head-tracking device. Support them all.
9. Some keyboards don't have function keys. And very few mice used on the Mac have two buttons. Some keyboards don't even have a numeric pad. Keep your eye on the details!
8. The Escape key and Command-Period both mean No or Cancel. The Return and Enter keys both mean Yes or Okay. Conform to Macintosh conventions.
7. I've probably seen your game running on a friend's PC. If the Mac version turns out to be ten times slower, I won't buy your next game.
6. I like alerts and dialogs, but I don't love them. I like icons, but I'm not an art critic or mind reader. Make sure your meaning is obvious!
5. The Mac version should always be better—and easier to install and use—than the DOS version. This includes the abilities to alert me when something goes wrong and to "play nice" with other programs.
4. If I can see it, I want to click on it. Let me.
3. If it isn't broke, don't fix it! If you create a custom scroll bar or button, it better look plenty awesome. The Mac's scroll bars and buttons are just fine, and chances are you haven't done something else I would have wanted (like making your program run really, really fast), so I will think that you have wasted your time. Stick to the standards.
2. If it is broke, fix it! QuickTime and `CopyBits()` are pretty slow. Do your best to get every bit of speed out of the program that you can. Do it as "legally" as possible, because if your program crashes, I won't like that, either. (Am I picky, or what?)

And my number one expectation for Mac games is...

1. If it looks like DOS, smells like DOS, and tastes like DOS, it is DOS. Your game should self-configure itself, explore its environment, and then let me override it—easily! After all, this ain't Windows!

As I said, I am very finicky about the programs and games I buy for my Mac. So are a lot of other Macintosh users. Keep this in mind above all else: Every consumer out there bought a Macintosh, not a DOS machine. And that should mean something to you, because it sure does to us.

Technical Prowess

Just because your game is ten times as fast as anyone else's or is technically marvelous in some fashion does not guarantee its success. Currently, one of the most popular games on both the Macintosh and Windows is a game called Myst, published by Broderbund Software, Inc. Myst is programmed almost entirely in HyperCard, very different from hand-optimized assembly. Although it is an exception to the general rule, Myst does illustrate the fact that technical prowess is not necessarily critical to developing an extremely popular game.

On the other hand, I would not suggest that you attempt to create a flight simulator using HyperCard. Although it could probably be done, flight simulators are best when they are blazingly fast. The speed required to calculate the 3-D points, combined with the full-screen imaging that is a requisite of the genre, demands really low-level assembly optimization.

In other types of games, the use of C or another high level programming language may be sufficient. Many successful games are not based on high frame rates or arcade action at all. Shanghai, SimCity, and Civilization are examples of popular games that succeed without needing highly advanced graphic engines. The designs of those games do not require technically advanced graphic capability. Car racing games, however, would require fast, responsive graphics because of their fast, arcade-like designs.

Match the technical prowess of your coding effort to the game. Hand-optimizing a blit for a chess game is time wasted. Your effort is better spent in providing more options for the player or enhancing the intelligence, since those are the elements of the game that create the challenge. Whenever possible, spend your time focusing on the portions of the game that really make it fun, exciting, or challenging for the player.

Conversions vs. Ports

Although the words "port" and "conversion" are used almost interchangeably within the entertainment software industry, they can mean different things. That difference is easy to explain.

A game that makes you feel like you are playing it on an IBM PC has definitely been "ported." A game that has been "converted" makes use of the Macintosh interface, the high resolution capabilities, the ease of use that is standard to any Macintosh program, or some combination of these enhancements. In other words, you know you are using a Macintosh by more than the Apple logo on the front of your computer.

For instance, games that make extensive use of the keyboard and fail to provide menu commands—and Command-key equivalents—are obviously missing the boat. Users are always able to refer to a menu when they can't remember the keyboard equivalent, even if the game's manual has been left in the floorboard of the car under a week-old hamburger wrapper.

Another example of porting (when conversion is clearly called for) is a mouse driver that due to the frame rate is slower than the Mac's built-in mouse. The time it would take to convert the code to support the built-in cursor, or a faster custom one, is time well spent. Nudge, nudge. Wink, wink. Know what I mean?

Perhaps the greatest blunder of all is making a false assumption. Take function keys as an example. There are at least three models of Apple keyboards that don't have function keys at all. Another dangerous assumption is that every monitor is 640 x 480. A majority of Macintosh models sold to date have displays that are not 640 x 480. Detecting the size of all attached graphics devices is simple and straightforward, requiring only a few lines of code. Follow Apple's guidelines and detect what size monitor your program is running on and arrange your game's window appropriately within it. With the latest multiscan monitors, it is even possible that the monitor will change sizes while your program is running.

A conversion takes these factors into account. Commands that are frequently used in the game are implemented in menu items, have sensible Command-key equivalents, and keyboard equivalents as well. Command-A, "A," or "a" all mean "accelerate," for instance, and Accelerate is a menu item in the Actions menu.

Of course, a cluttered menu bar or menus with 30 items would be overdoing it a bit. Maybe an icon bar would help, or at least a menu item called Help that displays a guide to the keys used by the game. Maybe try a dialog that allows users to enter the keys they want to use. Now that's the Macintosh way!

Porting a Title

There are times that porting is economically appropriate, especially in the case of entertainment software. Games have a relatively short shelf life at the retail outlet, and often the effort to completely convert a title to the Macintosh is outweighed by the anticipated sales at the retail level. But there are ways to "minimize the damage."

First of all, concentrate on the aspects of the game that are critical to the game. If you're working on an arcade-style game that relies on speed and a high frame rate, concentrate on that. Make it feel nice. Make it feel right. Make it "run like the wind." At least the people who do like it will forgive you for not providing all the Mac bells and whistles they otherwise expect. They won't care. They will be having fun playing your game.

Second, make really, really sure your game is solid. Games that crash a lot get little sympathy. Generally, crash bugs consist of a number of common errors that fit roughly in the following categories:

1. Toolbox errors
2. Memory errors
3. Plain old dumb stuff

Toolbox errors are probably the most common for a beginning Macintosh programmer. The Macintosh System software can be a wonderfully horrific thing. Always check the return value and handle any error conditions. If nothing else, display an alert that notifies you of the error number while you are developing and testing your game. This technique saves countless days of hair-pulling, teeth-gnashing misery. For calls to the Memory Manager, always check `MemError()` just afterwards. For calls to the Resource Manager, check `ResError()`.

The 11 - Porting folder on the enclosed CD-ROM includes a file named Errors.c that includes some handy functions for the checking and reporting of errors. Use these functions, or similar ones, to ensure that you are not passing invalid results to other parts of your program. Build your own to detect error conditions from your own functions.

Memory management is generally where programmers begin their distrust of the Macintosh and where the Macintosh honeymoon usually ends. There are several categories of memory errors that occur on a fairly regular basis. The first is not checking the return value of a memory allocation request such as `NewHandle()`. Always check return values and errors.

The other mistake I see fairly often is the failure to understand the concept of a handle or a pointer. Once you can clearly understand the concept, you are much better off. A handle is a pointer to a pointer, but it is even more. A handle is actually a pointer to (contains the address of) a pointer variable, whose value can change at specific and well-defined times. Make sure you know when you are using the pointer variable and when you are using a copy of the variable. The pointer variable can change at well-defined times, as follows:

1. Calling other code segments (68K)
2. Calling other libraries or Cross-TOC calls (PPC)
3. MacOS routines that can move memory (use Think Reference or Toolbox Assistant for quick reference)

Note

You'll need either Think Reference or Toolbox Assistant anyway. I have no idea how anyone programmed the Mac before they were available. Toolbox Assistant, from Apple Computer, is newer and more complete, although Think Reference has advantages for beginners, such as examples and helpful information.

The enclosed CD-ROM contains a file named MemAlloc.c that contains a few memory allocation functions that can come in very handy when allocating and deallocating memory on the Macintosh. Although they are available, `malloc()` and the standard C memory allocation functions merely end up calling the Macintosh Memory Manager anyway, adding overhead to your game.

In the category of "Plain Old Dumb Stuff," there is no end to the list of what can happen when you sit at a keyboard and let your mind wander while writing code. For example, never dispose of a handle or pointer and then try to use it later in the same routine. The file called MemAlloc.c contains functions that prevent that particular error. You might forget to initialize a variable before using it. Or test your game only for a week on a single Macintosh model and then call it "Final."

There are a number of ways to minimize the Dumb Stuff. First, turn on every warning your compiler has and then read each and every warning—often. Look at every warning and make certain you understand the full implications before ignoring it. This saves days of searching for that elusive bug later! If you are using C, turn on Require Prototypes. This may prevent your `InitMenus()` function from becoming confused and calling the Menu Manager's `InitMenus()` function, which will almost certainly crash your Macintosh or produce other interesting effects, none of them good.

Another instance of Dumb Stuff is calling a function that is not a function, or confusing a global variable with a local variable. A number of problems can be solved in a rather simple way. From experience, I know that the method I propose takes discipline, patience, and some degree of detail orientation to implement, but once learned, it is well worth the time, and can actually go a long way toward making the conversion process easier.

Most PC source code I deal with, especially C source code, is rather difficult to read because it is generally impossible to determine the difference between a macro, global variable, function name, or constant. If we adopt a naming convention, however, it is possible to easily figure out what code is doing (or attempting to do).

Apple has pretty much already defined a standard (albeit halfheartedly), which all the authors of this book pretty much conform to. Take a copy of the code you have to convert, and, using the search-and-replace function of a text editor, rename all the oddball variables to something that makes sense. All global variables in my programs start with a lowercase `g`, followed by mixed case—for instance, `gLivesLeft`. If I see the name, I know what it is. All my functions are merely capitalized and contained mixed case, such as `MyVerySmartFunction()`. All my constants start with a lowercase `k`, as in `kMyPiConstant` or `kMyDialogIdNumber`. Macros inevitably start with a lowercase `m`. Local variables always start with lowercase and are mixed case, as in `myLocalVar`. Type definitions begin with a lowercase `t`, as in `tMyWindowStruct`.

It seems like a lot of work. And without a doubt, it is. I challenge you to convert a program with function names like `my_function_a` and macros named `get_mem` and global variables named `my_window`. Then convert a program that uses a standard naming convention such as I have suggested. After you have converted, debugged, and shipped both programs, come tell me which method you prefer. I have a feeling I know which one you would choose.

Legibility of code is important. Commercial conversions are rarely completed using a single compiler, and some source code modifications are almost always required when changing compilers. The introduction of the Power Macintosh has almost guaranteed that you will use more than one compiler before shipping a game. Certainly you will use one for the 68K-based models and another for the PowerPC-based models. You might even use one for development and debugging, and another for the final testing and build. And since Macintosh compilers cannot by nature be standard ANSI compilers, there are always differences in how compilers arrange global data in memory, how functions are called, and a host of other differences that result in having to look at individual macros, globals, and function declarations. The faster you can accomplish this—and the more reliably you can do so—the faster you can deliver a game to market.

Included among the Dumb Stuff is the Really Really Smart Stuff. Macros in C, for instance, are really cool. Macros are the tops in Smart Stuff. You can optimize with macros. You can save yourself a lot of time using macros. And you can do some Really Dumb Stuff with macros. Quickly.

An example of Dumb Stuff is using macros to change the names of functions to Macintosh equivalents, or as a result of conditionally compiling for the Macintosh. While this is perfectly acceptable in most cases, remember that you are adding a level of indirection within your code. What looks like a call to `MyMenuHandler()` may actually be a call to `MyKeyHandler()`, which does something much different. Looking at your source code later may confuse you and will certainly confuse someone else. You are much better off to spend a few minutes using a search and replace in your text editor than to allow your preprocessor to do so. Consider the following:

```
#define InitMenus() InitWindows()
```

The line of code above will replace all occurrences of `InitMenus()` within your program with `InitWindows()` at the time it is compiled. Unfortunately, your code still reads "InitMenus," and unless your memory is very good, you might not remember over a long period that it means "InitWindows" to the compiler. This is a silly example but illustrates the point. Spend the extra time to search and replace if necessary.

Each level of indirection adds a layer of complexity when you or someone else looks at your source code at a later time. Make sure that each level of indirection has a purpose and that the benefits outweigh the disadvantages.

A good example of smart macro use is defining constants that are otherwise meaningless. `kBufferLength` can be clearly understood at a glance, while `1024` has no inherent meaning. A general rule of thumb is that if a macro either makes your source code more understandable or accomplishes a valid purpose—such as optimization—then use it. If, however, you are using a macro to save time typing, you might reconsider. You may spend more time later trying to find a problem related to your macro than you would have spent typing in the first place.

Another disadvantage of macros is that they are expanded by your compiler's preprocessor, which is not technically part of your compiler. It is therefore not subject to the same rules as the ANSI-compliant compiler you are using. Each preprocessor can implement a different set of functionality than any other preprocessor; this can result in very different meanings. Proceed carefully, especially if you intend to use more than one compiler during the conversion.

Portability and the C Language by Rex Jaeschke is an excellent source of information about portability and discusses the implementation of preprocessors. If you intend to convert programs using the C language, I recommend that you obtain a copy of this book and read it carefully. Mr. Jaeschke discusses not only preprocessors, but a number of issues relating to portability, including the standard C libraries, lexical elements and tokens, and recommendations for writing portable—and legible—source code.

By mentioning the "standard" C library, I cannot resist a suggestion. All file, memory, and most numeric conversions are built into the Macintosh System software. The standard C library in most cases calls the System software, or in other cases reimplements the same functionality. In a few cases, such as `memcpy()`, calling library functions can result in much slower results than if you call `BlockMove()` or `BlockMoveData()` directly. Memory allocation is best done through the Macintosh functions `NewHandle()` and `NewPtr()`, or through the memory calls I have included on the CD-ROM that allow you to track the amount of memory allocated and freed by your program.

Converting a Title

When converting a title as opposed to porting it, you have to work even harder to make your game a real Macintosh application. We might even call this "Macintizing" it.

In a Macintosh application, the standard menus are displayed at the top of the screen (the Apple, File, and Edit menus at a minimum), and a variety of keyboards, monitors, and input devices (mouse, joystick, game pad) are supported. A good deal of the Macintosh OS should also be supported, when appropriate.

An example of this process is what occurs when a needed file cannot be found. On PCs and other operating systems, files are usually located by their full path names, for instance, C:\GAMEDIR\DATA\MYDATA.DAT. On the Macintosh, the user's freedom to easily move files can be supported without too much trouble on your part. While you can still use a full path name, if the desired file cannot be found, it is fairly simple to ask the user to find it for you. First, display an alert or dialog stating that the file could not be found (this takes one line of code and a trip to ResEdit). Ask if the user wants to find the missing file or quit the game. Then either display the `StandardGetFile()` dialog or quit the game. This is far preferable than a cryptic message that "Error #103A14 has occurred." An example is included in FindFile.c and FindFile.rsrc on the enclosed CD-ROM.

In addition to finding files, you may also want to enhance a game by making use of the Macintosh interface. Almost every game worth its salt is capable of saving and restoring a user's position within the game and the general conditions of the game environment. Often this capability is built into the game through a keyboard command or through some sequence of commands. In order to simplify the interface for your Macintosh users, provide Save and Open menu items in the File menu, and tie these menu options into the code that expects the special cryptic keyboard commands. If desired, you could also support the cryptic keyboard commands in addition to the standard Command-S and Command-O keyboard menu equivalents.

Unless your game is "full screen," which actually ends up meaning 640 x 480 in most instances, support a standard Macintosh window in which you will present the game's graphics. Keep in mind that 640 x 480 is much smaller than full screen on some monitors. Generally this can be handled by creating a window that is as large as the "main device," painting the background with black, and then drawing into the center of that window.

Consider hiding the menu bar when appropriate—the techniques for this are shown in chapter 3, "Advanced Graphics." Of course, if you do so you should also make it fairly obvious to the user how to make a menu selection when the menu bar is hidden. For instance, if the mouse is moved within the menu bar's rectangle, make the menu bar visible. When the user moves the mouse outside the menu bar's rectangle, hide the menu bar once again. Scott Knaster and Keith Rollin's book, *Macintosh Programming Secrets*, contains, among other things, source code that illustrates how to hide and show the menu bar. Since that book was published, however, Apple has redefined some low memory globals, including `GetMBarHeight()`. Use `LMGetMBarHeight()` instead, which is prototyped in Apple's header file named LowMem.h. Although there is no practical difference in the functionality, at least you will be using the latest header files and methods to accomplish your task.

While some of these program modifications are easy to make, some are not. In fact, some may alter the structure of the original program dramatically. The value that you add to the original game, however, is appreciated by the Macintosh community and then has a good chance of adding to the sales of your game.

Converting a Game Idea

If you are a hobbyist programmer and are considering "porting" an existing game to the Macintosh in your spare time, there are some considerations that apply to you. The unauthorized use of copyrighted or trademarked characters, intellectual properties, or interface designs may result in legal actions against you. Obviously it is not legal to directly copy a game, even if you write the code from scratch. It *is* legal, however, to use an existing game as inspiration.

In the past several years, a number of court decisions and new laws have fundamentally redefined a number of issues relating to intellectual property. These issues have become very complex and cannot be addressed in the space that we have here. I would recommend that

you research the intellectual property laws and perhaps even consult an attorney if necessary. While this method is not exactly legal, you could create an exact duplicate of a game for the Macintosh and share it only with your family. The chances of you getting sued for that are obviously minimal. If you were to post the game on an electronic bulletin board or online service, however, you should probably expect a call from the original game's publisher or the holder of the copyright or trademark you have infringed. The general rule is the more money that's involved, the bigger the problem, although that rule is not always the case.

I would challenge you to instead use a game that you like as inspiration and create a brand new game. Use the basic idea or basic game play mechanism, but change the characters, reverse the goal, or create something unique from the inspiration. A good example of this is the game Shanghai from Activision, which is based on the centuries-old Mah-jongg dominoes game from China. The rules were changed a bit and the goal was reversed. A solitaire mode was added, graphics and sound created, and the game became an immediate and huge success.

Technical Considerations

The technical considerations when porting a title affect each and every component of the game, including the graphics, sound, game play, and interface of the game. Depending on which platform the game was created for, you may be able to use the existing graphics. Or the existing sounds, although this is less likely. Perhaps only a simple conversion of a file's format is required. Or maybe you need to rewrite the entire mechanism that loads and draws graphics and then reformat all of the graphics.

Other technical considerations include the programming tools to be used and the programming language used to convert the game. Almost invariably, the C programming language, combined with assembly when needed for speed and optimization, is the language of choice for most commercial game development. The reasons for using C become even stronger when you are converting your game to both 68K- and PowerPC-based Macintoshes. Most existing games are written in either a combination of C and assembly or, in rare cases, assembly only. When you're converting from one programming language to another, two of the most important factors are the number of comments in the original code and the availability of the original programmer to answer questions that inevitably crop up.

Tools of the Trade

There are at least three development tools of prime importance to your conversion. The first and foremost is, of course, your compiler. Whether you end up with MPW (slow, but reliable), Symantec C/C++ or Think C, or Metrowerks (new kid on the block), spend a lot of time reviewing the various options and settings for the compiler—especially in the case of MPW, which has the most compiler options and is the hardest to learn.

Some of the newer compilers for the PowerPC, for instance, currently have idiosyncrasies when optimizations are enabled. While developing your title, you should turn optimizations off in order separate the compiler's errors from your own. Optimized code is also more difficult to debug when using a source-level debugger. Registers and variables often don't contain the values you expect on a line-by-line basis. Some lines of code are "optimized out"—they simply are not needed in the final program and thus cannot be "traced" during debugging. During the playtesting cycle, you'll want to thoroughly test the optimized code, however. Never assume that a compiler's optimization is bug-free. Remember that your users will not know the difference between your compiler's bug and your own, original bug. Nor will they care.

The second tool you should spend time learning is ResEdit. ResEdit allows you to graphically edit your menus, windows, dialogs, and other resources needed by your application and the Macintosh operating system. Resources include small pieces of data, compiled code (68K only), templates for windows and dialogs, control templates, and fonts. ResEdit is available from the Apple Programmer's and Developer's Association. Always, however, keep a backup of any file that you are editing. ResEdit is historically not the most stable of all programs. Giving it a larger application heap generally helps this situation dramatically (Command-I after selecting ResEdit in the Finder, then increase the Preferred memory setting).

The third required tool, and generally least liked, is your debugger. In most instances, the debuggers shipped with the Big Three compilers are all fairly stable (I assume you are using C, but this applies to most commercial-grade compilers). Eventually, you have to learn at least a little about Macsbug, Jasik's The Debugger, or the Macintosh Debugger for PowerPC. High-level debuggers are adequate for most debugging tasks, but they do not allow you to poke around your application's heap or examine memory on a block-level basis, which is inevitable unless you write really, really good code. Small utilities such as ZoneRanger—included with Metrowerks' debugger—are also invaluable.

If you are using MPW and do not have access to either Symantec's or Metrowerks' compilers, you have a lot more to learn before you're able to begin debugging. The debuggers that work with MPW (including SADE, The Debugger, and others) take time to learn and become proficient with. For that reason, I suggest either Symantec's or Metrowerks' compilers if you can afford them, especially if you are new to programming. Even if you build the final "rev" on MPW, the cost of those compilers are well worth the trouble you save in the debug cycles of any moderately sized program.

If you are currently a PC programmer and are considering programming on the Macintosh, you will probably be more comfortable working in a compiler environment similar to the one you're already using. MPW is similar to most command-line compilers on the PC, such as Watcom, Borland, or Microsoft. Both Symantec's and Metrowerks' compilers are based on a project metaphor, or IDE (Integrated Development Environment). If you are familiar and comfortable with those types of environments, you may be more comfortable with Symantec's or Metrowerks' compilers.

The Apple Programmer's and Developer's Association (APDA) is a key source of information about programming tools for the Macintosh. Although it is not the only source of information, it is nonetheless a vital source when determining what tools are available. APDA may be reached at (800) 282-2732 or (716) 871-6555, in Canada at (800) 637-0029, or on AppleLink at APDA.

Knowledge Is Where You Find It

No one person or source will have all the information you will require in order to write or convert games for the Macintosh. Every source of information will be critical in one way or another to your efforts to become a proficient Macintosh programmer. There are a number of books that are devoted to programming for the Macintosh and focus on specific areas of programming. There are books devoted to learning to program or books (such as this one) that assume you already know how to program but focus on the differences when programming the Macintosh. My advice is to look at them all and read the ones that apply to you.

If you are going to be using MPW in your development efforts, you will probably want one or two books that familiarize you with the MPW development system. Symantec's and Metrowerks' are simpler development environments and can be learned with the documentation that is included with the compilers, assuming you already know the basics of how to program. If not, consider buying a programming tutorial for the Macintosh. Several of these books are available, and a few even include smaller versions of Symantec's or Metrowerks' compilers. In this book, chapter 0, "A Game Programming Primer," will help you get in tune with the minimum skills for game programming—or you might want to get a copy of *Programming Starter Kit for Macintosh* by Jim Trudeau (Hayden; ISBN: 1-56830-174-X).

The *Inside Macintosh* series published by Addison Wesley (but written at Apple Computer) is practically required in order to program for the Macintosh. *Inside Macintosh* documents the system software and operating system, which you absolutely need to know. The entire series is now available on CD-ROM and is available at local bookstores, computer stores, or directly from APDA.

Think Reference or Toolbox Assistant, electronic databases of the information contained in most of the critical volumes of *Inside Macintosh*, are also very helpful. With thousands of function calls in the Macintosh System software, they provide a convenient method of determining the header files that should be included, the prototypes of the functions, helpful information, as well as examples and suggestions for accomplishing various tasks.

The Internet and online services, such as eWorld, America Online, and CompuServe, may also be helpful if you are new to programming or to the Macintosh. These services provide a means for asking for advice from more experienced programmers or sources of example source code that demonstrate various means of accomplishing programming tasks.

Note

If you're looking for programming resources on the Internet, a good starting place is Yahoo (`http://www.yahoo.com/Computers/PCs/Macintosh/Programming/`). Also be sure to check out the Web site devoted to this book, which you'll find at `http://www.mcp.com/hayden/mac_game-gurus`.

The Developer University program, sponsored by Apple Computer, is another excellent source of information. DU offers a number of courses that teach everything from basic programming techniques to Macintosh-specific courses covering networking, MPW, System software, and a variety of other topics. A number of these courses are even offered in "correspondence course" style. Otherwise, classes are currently taught in various locations across the United States, with a majority of the classes offered in Cupertino, California, at Apple's headquarters. The advantage of traveling to Cupertino is the opportunity to visit Apple's Company Store, where you can purchase anything from T-shirts to coffee mugs, all happily emblazoned with the Apple logo. These items are generally referred to as Nerdware, but are essential for any Macintosh programmer. The Developer University can be contacted at (408) 974-4897 or through AppleLink, eWorld, and America Online.

Programming the Macintosh when the System software was relatively new and the size of the ROM was only 128K was much easier, believe me. These days the *Inside Macintosh* series alone stands about two feet high and weighs 30 or 40 pounds. Don't be discouraged, however. Most of the information is related to a lot of System software you really will not need when converting game software. After all, unless your game supports electronic mail, "drag and drop" data conversion, "publish and subscribe" (called OLE in the PC world), or other esoteric features, you will really be faced with learning only a moderate amount.

I have compiled a "short list" of really critical topics with which you should be familiar and the corresponding volumes of *Inside Macintosh*. Start with these and progress as needed through the other volumes. Also, keep an eye on what features of the Macintosh OS are implemented in other, top-selling entertainment software. That will be a good clue about what you should know and implement in your next game, unless you plan to lead the pack.

Subject	*Inside Macintosh* Volume
Memory Management	*IM: Memory*
Basics	*IM: Overview*
	IM: Macintosh Toolbox
	IM: More Macintosh Toolbox
File I/O	*IM: Files*
Mac Interface	*IM: Human Interface Guidelines*
Sound	*IM: Sound*

Notice that I have not included all the books about networking, interapplication communication, QuickDraw GX, or a number of other advanced topics. Generally, unless you are converting some really advanced game play system, they will not be needed. A "scriptable" game, for instance, would be kind of neat, but would your market difference be significant enough to warrant the additional work? (Look, mom, I'm playing it in my sleep!)

Converting the Code

The first step in converting a program to the Macintosh depends upon the programming language originally used, the target language, and a good many other issues. Instead of concentrating on converting one type of language to another, I will instead offer a general method to use when converting a large program to the Macintosh, regardless of the programming language or compiler.

First, you should make a copy of the original source code provided to you and place it in a separate, safe place. You will need to refer back to this code, and if you are like me, you will need it fairly often.

Second, after making a copy of the source code, see if you can get the code to compile. Don't worry if the compiled code will do anything or not. Just get as much of the code compiling as you can. My suggestion is to comment out any section of code that will not compile and cannot be fixed within a very few minutes.

If we compare a conversion to painting a picture, this begins the "sketch" phase. Your main goals during this phase are to:

1. See how much trouble you are really in.
2. Acquaint yourself with the overall structure of the program.
3. Identify any missing pieces, such as assembly or the environment's runtime libraries.
4. Determine which sections of code must be rewritten from scratch.

By focusing too closely at this stage on individual routines, you could be wasting time on code that is not even incorporated in your final build. Basically, once all the code compiles and links (or at least provides a list of link errors), you should be able to intelligently answer the four questions posed above. You should clearly identify which sections of the program are unsuitable for the Macintosh, which sections or modules need a little help or minor changes, and which functions are perfectly fine as they are. The amount of code you had to comment out is also an indication of how much work lies ahead.

After this stage is completed, you can prepare the final schedule and technical implementation plan for the conversion. Sometimes this can be as simple as stating that everything will be converted to work pretty much as-is, excluding the obvious blit-to-screen vs. page-flip. Or it can very complex if major portions of the game code have to be reworked. The basic question is this: Do the underlying algorithms need to change, or just the methods in which the algorithms were implemented?

An example of changing an algorithm's implementation, but not its design, is file I/O. In almost every game conversion, the standard C `fget()` and `fread()` are simply not sufficient and need to be replaced with calls to such Macintosh System software functions as `FSRead()` and `FSOpen()`. These are implementation changes, but the overall algorithm (open the file, seek to a specific location, read a specific number of bytes, close the file) remains the same.

In other cases, you may be converting a program that constantly allocates memory in an interrupt function using a call such as `malloc()`. This is impossible to accommodate on the Macintosh without a change to the underlying algorithm, since you cannot allocate memory during interrupts using the Macintosh Memory Manager. In other words, you might have to implement your own custom memory management functions. If you preallocate a large block of memory at startup using the standard `NewPtr()` call, it would then be possible to create a method of allocating memory at interrupt time. You could do so by managing the one large chunk of allocated memory and subdividing it as needed. This, however, is a change in the basic algorithm—you will need a custom memory manager. And memory managers do not write themselves, nor are they usually bug-free as initially written.

Thus, you identify the exact technical implementation for the conversion of the entire game. If you are not sure how much time coding a module will require, plan a couple of methods, including a fallback method. Add up all the time you have scheduled for each module of the program, be pessimistic, and then add about 10–20 percent depending on how well you have estimated work in the past. If the final schedule is much beyond what you have been allowed in order to complete the work, I suggest you honestly present the new information and negotiate a revised schedule.

If the company or individual who hired you to complete the conversion is unwilling to renegotiate your schedule or demands that the conversion take only the scheduled time, obviously a compromise must be reached. In some extreme cases—but they do happen—it is best to walk away from the project at that point. Late delivery in some cases can be worse than never delivering. Also, if in delivering your project late you are in danger of going out of business or missing the rent payment, you won't be around for the next project. Identify and address scheduling issues immediately. The sooner these issues can be resolved, the better for you and your employer.

In some cases, however, this is the point at which reality conflicts with wishful thinking. Maybe your game can end up being a little slower, a little less Mac-like, or a little less wonderful and still ship as scheduled. Especially if you are working on contract, these will be important issues. Or maybe you can work a little harder and complete the project as originally scheduled.

Believe it or not, the most important thing about a developer, whether an individual or a corporation, in the eyes of an entertainment software publisher is this: Does the developer complete projects, and are they finished on time?

The recent rash in "vaporware" releases, often a year or more before the game is eventually released, is proof that this occurs even with large, well-funded companies. Sometimes it

occurs specifically because of the size of the developer or the size of the development team. Large teams can accomplish more work if they are well coordinated, the technical designs and goals are clear, and the lines of responsibility are well defined. Large teams can also create very large messes when one or more of these conditions are not met. Planning is a vital portion of any programming task and applies to conversions as well as original designs.

After your technical implementation plan is complete and you are comfortable with your revised schedule, you are ready to begin the conversion. In other words, it's time to go to work. If your time estimates for each section of the conversion were well planned, your conversion will be a straightforward effort of implementation. Keep your eye on the schedule. If you get ahead, great. Be happy and go on to the next section or module. If you're behind, consider falling back to another method. Define the conditions under which you will fall back to another method, and when those conditions are met, do so. Constantly keep yourself informed about your actual progress versus your projected—and scheduled—progress.

Technical implementation plans can incorporate a wide variety of information, depending on whether the plan is prepared for your own use, for your employer, or even for your employees. Be specific as you need to be, depending on your audience. Err on the side of more information than is required. Provide sections of pseudo code if necessary. List any experience you and anyone working on the game has had with the different technical aspects of the game. Make a note of any code you already have—and can legally use—to assist in the game implementation, listing any necessary changes to that code base. If you are unfamiliar with a subject, such as sound, you will need to get as much information about it as possible. Perhaps even write a sample program to familiarize yourself with the subject. Obtain any sample code you can lay your hands on and look at it. Try to determine how difficult it is really going to be to move that code into the game you are working on.

The area of technical planning, in my experience, is the one area in which technical personnel are most often lacking or most often overlooked. The ability to reasonably plan a technical implementation and stick to the plan is vital to your ability to actually ship a game.

Conversely, if you are not involved in the technical planning process, it will be one of the areas about which you will probably complain the most. The realization of how difficult this process really is gives you a greater appreciation for those actually involved in the planning and scheduling of technical projects. Ask to become involved in this process and then hang around for an entire project. Write down what assumptions you made at the beginning of the project. Learn which assumptions turned out to be true and which did not. This process may give you more empathy for those responsible for technical planning and scheduling.

I have been involved in both technical planning and implementation, and both are frustrating in different ways. The rule of thumb is that there is never enough time to do all you want to do, even if you work hard, work smart, and use every trick in the book. When you have to rely on others to implement your ideas or technical designs, add time for coordination and miscommunication that invariably happens.

During the implementation phase, your experience, intuition, and knowledge will be continually valuable. Some portions of the implementation can be written, debugged, and shipped. Some portions will be written, rewritten, debugged, and rewritten again before being shipped. Knowing the difference while writing the code will assist in keeping you on schedule. When in doubt, follow this process: Get it up and running, determine if it works (or will work after optimizing), then go back and finalize the code after all modules are up and running. Not before.

Some programmers have a tendency to attempt to write the final code on the first pass. Resist this temptation unless the function is really, really simple. When you are working on an untested algorithm, the goal is to evaluate the feasibility of the algorithm, not to optimize it. By proceeding in that way, you save time if the algorithm must later be changed and then reoptimized.

Once the entire program is up and running, perhaps in an unoptimized, barely playable state, a lot of new information is available, including the current date (versus the date you scheduled you would reach this point). If you are running late at this point, another set of decisions can be made about the exact Mac models to be supported (it takes a lot of work to support them all), including memory configurations, System software versions, and processor speeds. These decisions are made considering the amount of time left, the desired functionality of the converted game, and the amount of work to accomplish the original, planned functionality.

Original Platform

The game's original platform affects the conversion in a few critical areas. Graphics formats, for instance, are rarely directly portable. Sound, user input methods, and file I/O are also areas that change dramatically between platforms. The difference in compilers and basic data types can also be an issue.

When writing or converting programs, I begin by creating a standard "types" file that redefines most of the basic data types. For example:

```
typedef short tInt16;
typedef unsigned long tUInt32;
```

In this way the code base can quickly be moved to another compiler simply by making a few changes to this basic types file. When you're converting another program, it is often helpful to begin by replacing all instances of `int` with `tInt16` or `tInt32`, depending on the compiler used by the original programmer. Most Macintosh compilers allow `int` to be defined as either 16- or 32-bit values, although MPW predefines an `int` to be 32 bits. Using a standard types file limits the confusion that is introduced by using more than one compiler.

After converting the base types used by the compilers, you should investigate any structure alignment issues. This mostly applies to structures that are used to access data that is stored

in files or on disk. Structures that are created in memory and automatically initialized or initialized through code are often different sizes on different platforms without adverse effect.

Compilers tend to "pad" a structure so that 8-bit or 16-bit values are aligned on 16-bit or 32-bit boundaries. 68K compilers, for instance, prefer a 16-bit boundary, while PowerPC compilers sometimes prefer a 32-bit or other boundary for efficient memory access. This difference in memory alignment between platforms is compounded by whatever byte boundary is implemented by the original compiler. It is especially troublesome when the original platform is RISC-based and the default alignment is 32 bits. In those cases, you may need to "hand align" the structures that are defined and used by the program by inserting new fields so that the compiler performs no padding on its own. This conversion of structure alignment and padding may be complicated by the options or lack of options in your compiler. If you are using several compilers during development, hand alignment is the best solution.

The use of function pointers on the Macintosh is also quite different than on most hardware platforms, especially in the case of code that is compiled for 68K-based Macintoshes. Function pointers referenced from code in a different code segment may have a different value than function pointers referenced from within the same code segment. If you compare the two, even if they point to the same function, the value of the two pointers may be different, because some function calls are implemented indirectly through a jump table or other means.

Look for instances in which a function pointer is referenced in the original code, and be careful about when and how you use a function pointer. A more reliable method is to store a function pointer into a global variable and always use that function pointer variable. The following code is an example:

```
tFuncPtrType gMyFunctionPtr;

void function2PointAt (void)
{
   // Some code here, no doubt.
}

void main (void)
{
   gMyFunctionPtr = function2PointAt;
   // Use gMyFunctionPtr instead of function2PointAt after this point…
}
```

Graphics formats also vary widely between different hardware platforms. The 3DO Multiplayer, for instance, stores graphics in memory in a compressed state, unless the programmer has developed a proprietary graphics format. On other platforms it is common to store graphics in a "raw" or uncompressed state. This is especially true in 3-D applications that make use of a destination-based texture-mapping algorithm. Depending on the requirements of your game, you may or may not wish to store graphics in a compressed state.

A particularly poor format in which to store graphics is one in which data is needlessly repeated. PICT files, for instance, store an entire color palette along with the graphic data. While this is necessary when one picture is to be stored for later retrieval, it is extremely wasteful for a game. Generally, the color palette and all the graphics associated with that palette are saved only once. Of course, this requires that you develop a method of loading, storing, drawing, and even decompressing your graphic images.

While this may seem to be a lot of work, it can in fact be accomplished quite readily. In essence, a PICT is merely a 256-element array of colors, defined by separate red, green, and blue values and a two-dimensional array of references into that palette of colors. Rarely are games programmed for graphics modes other than 8-bit color due to speed and compatibility concerns. If your game uses more than 256 colors, consider saving as many graphics as possible in 8-bit format. This will save on RAM required to run your game, as well as the time to draw each frame of your graphic.

Sound formats, like graphic formats, vary widely from platform to platform. There are two basic formats for storing sounds: digitized and MIDI. Of those two, digitized sound is definitely the easiest to work with on the Macintosh. A variety of sound capabilities are built into Sound Manager 3.0, which runs on all Macintosh computers running System 7.1 or later. Actually, Sound Manager 3.0 runs on Macintosh computers under System 7.0 and later if QuickTime is also loaded.

If you intend on playing long digitized sounds, you probably want to "stream" them from the hard disk or CD-ROM. The preferable format when streaming is the AIFF format (Audio Interchange File Format). An AIFF file can be played directly from disk with only a few lines of code, assuming that you use Sound Manager 3.0. Make certain that you check the availability of Sound Manager 3.0 before making specific calls to it, however. You can check for it as follows, after including Apple's Sound.h header file:

```
Boolean soundManager3Available (void)
{
   NumVersion   sndVer;

   sndVer = SndSoundManagerVersion();
   if (sndVer.majorRev < 3)
      return FALSE;
   else
      return TRUE;
}
```

A number of utilities exist—both in the public domain and commercial applications—that allow you to convert digitized sounds to the AIFF format. SoundEdit 16 from Macromedia is a solid choice and handles a variety of sound file formats in addition to supporting stereo sounds, editing capabilities, and a number of sound manipulation functions.

Although the AIFF play-from-disk functions are convenient, they are not always the fastest, especially in terms of response time. You may consider using the double-buffering support

built into Sound Manager 3.0 and your own file format if response time is critical to your game. Generally, since all the sounds in your game will be stored in a single format, you can also eliminate the header information or other data generally associated with AIFF or 'snd ' format files.

Remember that you cannot read a significant amount of other data from the disk while streaming sound using the Macintosh's built-in routines without adversely affecting the audio. If you need this capability, a customized solution is required (and you will probably have to write it yourself).

Short sounds that are loaded into RAM can be preloaded and played in addition to, and simultaneously with, any streamed sounds. The preferable format, and the one easiest to deal with, is the standard 'snd ' resource format. As with the AIFF format, a number of utilities exist that convert sounds to this format, including SoundEdit 16. Once stored into a resource file, the sounds may be loaded with a simple `GetResource()` function call.

MIDI is an abbreviation for Musical Instrument Digital Interface, and is a method of specifying a number of musical instruments and then playing notes on those instruments at specific times and for specific lengths. The popularity of sound cards in the PC world has made MIDI a standard for creating music and sound effects on DOS machines. These sound cards contain a number of standard instruments that are played using a synthesizer chip built into the card. A software MIDI interpreter generally uses these standard instruments and merely specifies the notes to be played.

Each and every MIDI interpreter, however, on each and every platform known to man, prefers its own MIDI file format. The Macintosh is no different.

Since the installed base of Macintosh sound card owners is so small, a software solution is called for. There are only two options that currently exist. The first is SoundMusicSys, a MIDI driver available from Steve Hales that requires a licensing fee for commercial applications. The sound and performance are both first-rate, and a great number of very popular Macintosh games use that driver. Fortunately for you, the authors of SoundMusicSys have written Hollywood API and included it for you on the CD-ROM (see chapter 6), so your programs can have many of these same capabilities for just the price of this book.

The second solution, available only recently, is built into QuickTime 2.0, available from Apple Computer, Inc. Although the core technology for the MIDI driver in QuickTime 2.0 is based on the SoundMusicSys driver, there is no licensing fee. In fact, it's free. Just remember to call the `Gestalt()` function and make sure not only that QuickTime is installed, but that the current running version is equal to or greater than 2.0.

Both QuickTime and SoundMusicSys have limitations, and each has its own strengths. QuickTime 2.0, for instance, has a limited number of instruments, and custom instruments cannot be created for it (at least, not yet and not easily). Knowing your exact requirements will help you in deciding which is most appropriate.

If you wish to include QuickTime 2.0 with your finished game, however, contact Apple's Licensing department and sign a licensing agreement. Recently, Apple even dropped the licensing fee for QuickTime, so it can be shipped with your product free of charge. Apple's Software Licensing can be contacted at (512) 919-2645 or on AppleLink at `SW.LICENSE`.

In order to convert a standard MIDI file into a QuickTime movie, you will need the QuickTime 2.0 Developer's Kit, available from Apple Computer, Inc. A number of shareware and public domain utilities have also been released which convert standard MIDI files to movie files. Once converted, MIDI movie files can then be opened, loaded into memory, and played, or simply played from disk. The Developer's Kit contains a number of examples for playing QuickTime movies, including music-only movies.

Original Programming Language

The use of the C programming language is predominant for games on personal computers. Some assembly level optimization is almost always needed for final speed improvements and optimization, except perhaps on PowerPC-based Macintosh models. The use of assembly as a development language is generally restricted to console-based game platforms, such as the Sega or Nintendo machines. Conversions of games written completely in assembly are more difficult and require a lot of programming knowledge and talent.

The conversion of a C language program, however, is easier and can be accomplished by an average programmer. The wide availability of C compilers, combined with the language's efficient usage of the hardware's capabilities (such as a large register set), make C an ideal choice for game development. Programs that have been originally written in C are about the easiest to convert of any programming language. Once the hardware interface issues have been separated from the game code and replaced with Macintosh equivalents or near equivalents, the rest of the source code conversion focuses on compiler and runtime differences, plus any program errors introduced in the process.

Other languages are seldom used for game development, but include Basic, Pascal, HyperTalk, and LISP, to name a few. Games that are developed in these less popular languages can usually be converted to C or crosscompiled much easier than assembly code, although the process is generally tedious. The availability of compilers or interpreters for these languages for the Macintosh may also speed the conversion process.

Your Development Platform

Selecting a development software platform is easy, provided you have an almost unlimited budget. For the rest of us, it is a bit more difficult. No one development environment for the Macintosh is ideal or solves every problem and challenge. MPW has problems in that it is slooooow and the debugging environments are either slow or difficult to learn. It is, however, a very solid environment and appeals to programmers who are comfortable with make files, command line interfaces, and batch files.

Metrowerks' and Symantec's compilers both approach the development environment in substantially the same way. They are both easy to use, support an intuitive debugger, and are self-contained. Symantec was slow in its release of a PowerPC C compiler, though, and still has not delivered a solid PowerPC C compiler as of this writing. If you are developing a native PowerPC application, this may concern you.

Of all the PowerPC compilers, MPW is the most solid to date. The availability of Motorola's new PowerPC C and C++ compilers also tip the scales in MPW's favor.

Given a reasonable budget, the purchase of either Metrowerks' or Symantec's compilers, combined with MPW for the Beta and Final PowerPC builds, is the optimum solution as of today. Source debugging can then be accomplished easily and the final build remains solid and as optimized as possible.

Your actual development hardware should sport a large hard drive; enough RAM for your game, the compiler, and the debugger to all run simultaneously (plus all your favorite INITs, of course); and a standard color monitor. The faster your processor is the better, especially if you use MPW on a daily basis. A minimum configuration for commercial development would be a 68040-based Macintosh, 250 MB hard drive (if you are converting a disk-based game), 16 MB of RAM, and a 14" color monitor.

If your development environment is extremely high-end, make sure you run your game on a mid-level system fairly often during the development cycle. It's good to keep in mind how the average Macintosh owner feels when playing your game. This also applies to large monitors, huge amounts of RAM, and quadruple-speed CD-ROM drives. Remember the average and lower-end users. They are money in your pocket, unless your game will not run on their Macintosh, or runs so poorly that it is returned to the store for a copy of my latest game.

Multiple monitors, uninterruptable power supplies, hardware debuggers, and logic analyzers are really nice—and really unnecessary, for the most part. Supporting multiple monitors affects a very small percentage of the Macintosh market, although I have seen a flight simulator with a 3-D display spanning three separate monitors. That was kind of neat.

In general, use your best judgment. If a feature or device is necessary and within your development budget, buy it. A feature that is only desirable should be avoided. You may need that part of your budget later, when things are not going well and the project is running late. And the rent is due. Again.

The Need for Speed

Almost any game that animates a significant portion of the screen is going to be too slow. Plan on that from the beginning. Do everything you can to speed up your program, and do it from the start of your conversion. As I said, though, don't spend too much time at the beginning on any single function. Concentrate instead on the method you are using, not the precise implementation.

If you are converting a game from DOS, for instance, chances are the conversion to Mac will make it two to three times as slow. Take the example of a DOS game that has a resolution of 320 x 200 pixels and uses a "page flip" to display a new screen practically instantaneously. But 320 x 200 on a Macintosh monitor is tiny. Real tiny. At a minimum, the image must be doubled while transferring it to an on-screen window.

You are copying 256,000 bytes every frame if you completely redraw a 640 x 400 rectangle at each update. The PC has to copy 0. You are already way behind the game.

In some instances, all new graphics are prepared for Macintosh versions of games. Larger, more detailed graphics. Graphics that take more time to load and draw in the first place. Strike two.

Third, the Finder steals a little bit of time (sometimes a lot) every time you call `WaitNextEvent()`. Then there is AppleTalk, which takes its slice of the CPU pie. Strike three.

Sounds hopeless, doesn't it? Chances are your version of the game is slower when compared directly to a DOS version, CPU to CPU. For the most part, you do have to buffer graphics off-screen and blit them on-screen to avoid animation flicker. And you do have to endure the countless background tasks, INITs, and other CPU cycle-thieves. The object is to work smart, not hard.

Update only the section of the screen you absolutely need to. Make more, but smaller, blits to screen. Erase or clear only sections of the screen that have changed. Run your profiler (Metrowerks provides a good one) and speed up the functions that are called most often or do the most work. Write your graphic imaging code in assembly.

Do not synchronize with a vertical blank. This synchronization will not only slow down your game, but it is very difficult to do (there can be more than one monitor connected to a Macintosh, each running at different refresh rates). Even Apple has suggested that doing so may cause problems now and in the future.

This issue is addressed in the next section on following Apple's rules, but more than likely, you'll want to write your own screen blitting code and provide it as a user-selectable option to `CopyBits()`. This is especially true in cases in which a blit is scaled, or doubled, on its way to the screen. An August 1992 article in Apple's *develop* magazine (issue 11) gives some guidelines when writing to screen memory. Follow the guidelines and always provide `CopyBits()` as an option. Just in case.

Besides optimizing your code and algorithms, there are a number of things you can do that make your game appear to run faster. Check for update events before calling time-consuming code. A large, white area of a window left on-screen gives the impression that it's taking forever to get things done.

Another method to make your game feel faster is to examine the interface and combine any steps that can be combined, and replace dialogs with menu selections. Basically, remove any repetitious actions the user has to endure. Modal dialogs should be used sparingly, like salt. Avoid displaying sequential dialogs whenever possible.

Do not draw text directly to screen. Doing so will cause flicker, which makes the program appear to be running slower than it is. Graphics and text should be buffered off-screen and then quickly blitted to a window on-screen to avoid flicker. You can even use a bitmap instead of a pixel map when drawing text off-screen in order to save memory. Let `CopyBits()` colorize the text for you during your blit to the window. Doing so is as easy as setting the foreground color (use `SetForeColor()`) to something other than black just prior to calling `CopyBits()`. Technical Note QD-4 also discusses this colorizing in greater detail.

Spin the cursor. A small trick, but everyone does it and for some reason Mac users are used to waiting and watching the cursor. They are so used to it, in fact, that they may think your game has crashed unless you do spin the cursor. Fifth Law of Thermodynamics: Time passes more quickly when the cursor is spinning. A file on the enclosed CD-ROM, called CursorStuff.c, illustrates how to make a few simple calls in order to spin the cursor. Of course, it will be most effective if `spinCursor()` is called repeatedly from within a loop.

Following Apple's Rules—and Breaking Them

The System software for the Macintosh has become more and more complex over the past several years. The interaction of the various Toolbox managers, installed INITs, System extensions, and control panels requires delicate management of System resources and careful programming. The System software expects you to write programs in a particular manner and will not operate correctly unless you do so. When you're converting programs to the Macintosh, this often poses a problem when the structure of the original program is significantly different.

DOS programs often use a "polling" method for determining when a key is pressed and rarely implement a screen update procedure. On the Macintosh, this method does not work as well due to the underlying Event Manager that automatically queues all keystrokes, update events, and mouse clicks. Usually these events are handled by a centralized function that dispatches the events to other functions that handle the different event types. Replacing multiple instances of keyboard polling with a call to a centralized function that then dispatches other types of events may cause problems with assumptions of the original program. If the original algorithm assumes that the keyboard polling is instantaneous, the call to `WaitNextEvent()` may slow the program on the Macintosh, perhaps unbearably so. Or perhaps handling an update event at that point may be structurally inconvenient.

The solution in that case would be to place the call to `WaitNextEvent()` within the original program's main loop and then buffer any keystrokes. The buffered keystrokes can then be read by a slightly modified polling function. This will not work, however, if the function that polls the keyboard does not return until it receives a keystroke. The solution in this case is to call `GetKeys()`—a Macintosh equivalent of keyboard polling—and then flush the event queue of any `keyDown` or `autoKey` events prior to exiting the function. You would not want to handle a `keyDown` event twice (once because of `GetKeys()` and once for `WaitNextEvent()`).

As you can see, the original program's structure combined with the expectations of the Macintosh System software both need to be considered. In some cases, it may require that you remove and replace entire sections of the original source code in order to correctly implement the Macintosh interface. In other cases, you can bend some rules or structure the calls to the System software slightly differently than would otherwise be "normal" in order to convert the program.

There are instances in which Apple's most sacred rules must be broken. When this becomes necessary, you should tread carefully in order to maintain the highest degree of compatibility possible under the circumstances. Writing directly to screen memory, for instance, is at minimum an ambitious undertaking when you consider all of the ramifications. If you are converting a flight simulator, racing game, or other game that animates a significant portion of the screen continuously, it's probably worth the effort. Apple does make certain recommendations on how to accomplish that to ensure the greatest compatibility with all Macintosh configurations (*develop* issue 12, distributed by APDA, contains information on how to maintain compatibility when writing directly to screen memory).

Another instance is the use of full path names, which Apple has always discouraged. This discouragement was primarily based on the fact that users are allowed to rename hard drives and disks. If your game is distributed on CD-ROM, there is practically no other choice than to use a full path name. Doing so in that case is appropriate, since applications are generally written to run from a hard drive and access data from a CD-ROM disc, making a relative path name unusable.

A number of the Toolbox Managers, such as the TextEdit services and the List Manager, were created for limited, general usage for a variety of different applications. You would not create a word processor based on TextEdit, nor would you create a spreadsheet program using the List Manager. For larger applications, custom solutions become necessary. In this same way, QuickDraw is inappropriate for use in a graphics-intensive application such as a game. Although it is possible to store and draw color pictures using QuickDraw, it is not optimized to be used as a graphics engine for high-speed animation. A custom solution is required in order to effectively animate images on the Macintosh, although QuickDraw may be used in certain cases to display the final product on-screen.

Thus, control of the off-screen imaging buffer becomes necessary. You need to know the exact format of the off-screen buffer in order to create an algorithm to draw graphics into it. Merely allocating a two-dimensional array of bytes of arbitrary size, however, will limit your ability to transfer the finished image to the screen. By allocating a QuickDraw-friendly buffer, you can use `CopyBits()`, `DrawString()`, or other Toolbox functions to transfer your image, add text to the image, or accomplish other graphic operations. Apple has long provided information about how to allocate your own off-screen graphics device and pixel map and specify the color palette to be used and the color depth of the pixel map. By using these routines in conjunction with your custom drawing functions, you can be assured of compatibility with future hardware and software advances.

A good source of information about off-screen drawing operations can be found in Apple's *Macintosh Technical Notes*. Technical Note #QD-13 (QuickDraw Note #13) contains the latest recommendations from Apple about allocating off-screen graphics buffers, GWorlds, and pixel maps. Included in the technical note is the necessary source code (both in Pascal and C). The *Macintosh Technical Notes* are available from various sources, including APDA, online services, and AppleLink.

Converting the Artwork

There is an absolute requirement if you are converting graphics on practically any machine, much less if you are converting graphics for the Macintosh. That requirement is a program called Debabelizer, published by a company called Equilibrium. It is available from practically any mail order source across the nation. Get it.

The first issue is the location of "white" and "black" within a Macintosh palette. Unless you plan on completely ignoring the Color, Window, Menu, and Palette Managers (and writing your own; but then we talked about that, didn't we?), you will have to address this issue. Color 0 in a 256-color palette on a Macintosh is always white. Color 255 is always black. Period.

Generally, that is the first problem that crops up when moving graphics from DOS or Windows (which uses the exact opposite scheme). Debabelizer is invaluable in performing this conversion. If a white and black already exist in the original palette, you are home free. Simply create a script within Debabelizer that swaps your white and black around until they are in the right places, click on the Remap Both Palette and Pixels option, and voilà! Later in this chapter, Eric will show you how your game program can convert palettes on the fly.

If you are converting graphics from one bit depth to another, say from 32-bit color to 256 colors, you need to group your graphic conversions into groups of graphic images that will be displayed—or could be displayed—on the screen simultaneously. This is because for each graphic group you will need to change the palette in order to obtain the best results. For each of these graphic groups you will need to find a 256-color palette that provides the best results for all of the images within the group.

As an example, divide up your graphics into "functional" groups within the game. This division can be based on natural breaks within a game—between different backgrounds, for instance. Once you have divided up the graphics, create a 256-color palette for each group using a tool such as Debabelizer. Once a "super palette" has been created that has taken into account all of the graphics, remap each of the graphics into the new palette.

In certain circumstances, it's necessary to duplicate graphics that will appear in more than one background. If your game is not distributed on CD-ROM, you may have to create an algorithm that remaps your graphics at the time they are loaded or at other appropriate times. If your game is CD-ROM-based, ample storage space may be available to store multiple copies of graphics that appear in more than one palette.

Converting the Data

Converting program data can range anywhere from no work at all to practically impossible. If the original program's data has been stored in discrete chunks, is well labeled, or the access method to the data is obvious, conversion of program data is generally straightforward. If the original programmer has thousands of files, all identified with a cryptic numbering system and no notes, and each file is in a different format, I would say that you have a problem.

The first concern when converting data is byte ordering of data stored on disk. Intel chips store bytes starting with the least significant and ending with the most significant. This byte ordering scheme is known as "Little Endian," for some reason that no one I have found can explain satisfactorily (the phrase was introduced in Jonathan Swift's *Gulliver's Travels*, roughly 200 years before Intel got in the microprocessor business).The Motorola 68K series of chips, however, accesses bytes in exactly the opposite order, and this is known as "Big Endian."

If the data you have been given was prepared for an Intel chip such as the 80486, you need to flip all 16- and 32-bit values—see "Selected Problems and Solutions" later in this chapter for the code you'll need. Note that this applies only to data written to disk and later retrieved in the opposite format.

Other considerations must be made for data stored on CD-ROM or the use of a large number of files. A directory (folder) that contains a thousand files, for example, is accessed much faster under DOS than under the Macintosh OS. A better solution, even on the PC, is to write a utility that combines all the data files into one. Store the location and length of each "subfile" into a second "table of contents" file, which you can then load into memory as needed. File access times decrease dramatically using this or similar schemes.

In some extreme cases, it may become necessary to create entirely new data files from scratch during the conversion process. Usually some type of utility is required to enter this data. If you should find yourself in this situation, there are a number of options available to you.

First, you could create a custom template in ResEdit, although it will not address every case adequately. Some complex data structures are impossible to represent with ResEdit's template functionality. The documentation for ResEdit, available through local bookstores as well as from the APDA, contains the instructions for creating a template. You can then directly enter data in a structured manner and save the data within a resource file, if that is appropriate. Resource files can contain only several thousand data chunks, however, so this option may not be appropriate.

Second, you could create an editor in HyperCard. The ability to quickly create text fields, buttons, and export data make HyperCard an easy choice for simple data entry. If you are familiar with HyperCard, you might consider using it to create your development utilities.

Third, you can custom program a utility in a high-level language such as C. This option, while by far the hardest, must be carefully analyzed. Whatever time you spend in creating utilities is time you will not be spending in converting your game. Balance the needed functionality with the goal of completing your game and keep your eye on the schedule. Avoid the temptation to do too good of a job in creating a utility, but don't skimp on features if it helps get your game finished on time. Any major editors or utilities needed for a conversion need to be factored into the conversion schedule and budget.

Converting the Music and Sound

Conversion of music and sounds for the Macintosh can be readily accomplished if the original music and sound is available in a couple of common formats, such as General MIDI files or AIFF files. These formats are supported by a number of Macintosh utilities or may be directly supported by the Sound Manager.

Most games being developed for the Macintosh at this time require the support of Sound Manager 3.0. That version of Sound Manager provides high-level support for a variety of sound functionality, including the ability to "stream" music files from disk. Streaming means that only small sections of the sound file are loaded into memory at one time, then played, and finally overwritten with other sections of the file. This limits the amount of memory that is required in order to play the sound, thus minimizing the impact on your game. AIFF files can be directly streamed from disk or CD-ROM.



Other formats can be used, such as the 'snd ' format discussed in *Inside Macintosh*. That format is ideal for digitized sounds that are completely loaded into memory and played. For sound effects that are played often during game play, this format is ideal. It does require that you allocate enough memory to your game so that all the sounds may be loaded simultaneously. Another option is to load the sounds required by certain sections of your game as each section is loaded, or provide other standard times to load and unload sets of sounds.

There are several commercial-quality sound programs that provide the capability to convert sounds between a variety of formats, edit existing sounds, and compress or resample sounds. SoundEdit 16 by Macromedia is a good choice, especially if you're dealing with stereo sounds and need the capability to edit or reformat stereo sounds. A number of public domain sound utilities also exist on the various online services which provide other capabilities, such as converting Windows' .WAV format to Macintosh-compatible sound formats. These utilities are often available for the cost to download them or for small shareware fees. The large services, such as America Online and CompuServe, have a variety of sound utilities available in both their programming and music sections.

SoundEdit 16 also supports MACE compression, which is built into the Sound Manager. This compression allows you to choose between compression ratios of 3:1 or 6:1, depending upon the trade-off of size and quality that you choose. MACE-compressed sounds can be

decompressed and played directly by the Sound Manager. SoundEdit 16 also allows you to save sampled sounds directly into the resource fork of a file. This capability, combined with the ease in loading and managing resources from within your program, make loading and playing sounds a snap.

In some cases you may wish to create your own sound-streaming functions. Sound Manager 3.0 also supports double-buffered sound play functionality that can be added to your game with a little additional work. The `SndPlaySndFromDisk()` function uses this capability when streaming an AIFF file from disk. You probably will not need to directly use the double buffering functions.

Sounds and sound effects that are in other formats, such as "patches" for specific synthesizer chips used in different hardware platforms, usually must be recreated or digitized from an original source. A number of alternatives exist in all price ranges for digitizing hardware and microphones, depending on the desired frequency and number of channels. Sound effect libraries are another source for sound effects, although in some cases the sounds are copyrighted and may not be used without first paying a royalty to the copyright owners.

QuickTime movies can be played with only a few function calls, and chapter 6, "Audio" shows how to do this in your own programs. This is by far the easiest and least expensive option to implement music on the Macintosh. QuickTime even allows "streaming" of MIDI movies, or loading an entire movie into RAM if you need immediate response when starting the movie. One issue to be addressed is QuickTime's limit on the number of instruments that can be loaded simultaneously.

If each of your QuickTime movies uses a lot of instruments, you may not be able to load all of them simultaneously. My recent experience shows that the current version of QuickTime (2.0) will load only about 65 instruments. If each of your MIDI tracks contain 16 instruments, you will be able to open and play only about four MIDI movies at one time. The idiosyncrasies of QuickTime are normal for a first version of software, and should be addressed by Apple over time.

A PC MIDI card, on the other hand, is concerned only about the instruments currently loaded, which is generally under a programmer's control. The instruments loaded by QuickTime, however, are loaded automatically.

While QuickTime does a fairly decent job of software MIDI playback, it's important to note that it does not truly support direct MIDI file format playback, nor does every instrument sound as good as a hardware synthesizer. QuickTime actually converts a MIDI file's data into a proprietary format when imported into a QuickTime movie using MoviePlayer, or some other utility. The output of the "Note Allocator" component within QuickTime can be either a Sound Manager component (software synthesizer), a General MIDI instrument, such as a keyboard, or a Nubus synthesizer.

For the best results you can support the available sound cards from vendors such as Roland, although the installed base of sound cards for the Macintosh is so low that you may also wish to implement music using QuickTime.

Macintosh-Specific Considerations

When converting titles to the Macintosh, there are a few areas in which the System software surpasses other platforms or varies widely from common standards. Apple's QuickTime, for instance, has surpassed the video capabilities of DOS-based machines for the past several years, although that is changing rapidly. The capabilities that are available and the tools to integrate video into your product are advanced and easy to use. If you are converting a title that makes use of video, QuickTime may solve your needs. However, you may have to create your own movie playback technology if you have specific requirements that QuickTime does not support.

The Macintosh Memory Manager, on the other hand, is unlike most of the common memory allocation systems available. It is based on an additional level of indirection that allows the System software to reorganize memory in order to avoid fragmentation within your application heap. You can take advantage of this fact, or, if the game you are converting already solves that problem, avoid the issue altogether.

Other areas that are unique to the Macintosh include the file system and standard file package, which allows the user a standard method of selecting and saving files. Windows provides similar functionality. Your support of the native file system will not only speed up your program somewhat, but result in a more Macintosh-like interface.

There are other considerations when porting a nonwindowed game to the Macintosh. Often these games will copy from the screen buffer to the off-screen buffer to restore a previous image. On the Macintosh this isn't possible, since your window could be spanning four monitors, each at a different bit depth. You will need to account for this difference in behavior by double buffering or by using other techniques.

QuickTime and Video

QuickTime is a remarkable technology that Apple has developed, allowing the use of digital video, sound, and musical scores easily in any application. A number of extremely advanced editing tools exist, such as Adobe Premiere, VideoFusion, and others. The ability to play video sequences on the computer enhances a number of applications, including games.

There are a number of uses of digital video that QuickTime does not yet address. If you want to composite a video image in a 3-D environment and use the video as a texture map or skew or rotate the image, QuickTime does not include that functionality. The ability to "layer" a movie on-screen with other graphics is also not currently available. For these types of applications, a custom solution is required. For the most part, however, QuickTime can be used in a wide range of games to play both music and video.

Conversion to QuickTime is fairly easy provided you have the original animation or complete set of frames for your movie. A tool such as Debabelizer or Adobe Premiere can be used to create a QuickTime movie file from sources such as PICT, TIFF, or other graphic

formats. Sounds must be converted to a few base types such as AIFF or 'snd ' files before including them in QuickTime format. The availability of the data in these formats or other compatible formats is necessary if the conversion calls for digital video.

In many cases, the original video may be provided on a video cassette. For professional-quality results, it's best if the video is provided in Betacam SP (a standard professional-quality format used in television) or similar formats. SuperVHS and Hi8 are the next best formats after Betacam. Normal VHS quality is usually insufficient for production of commercial games.

If your source format is Betacam SP, you should include enough in your conversion budget to rent a Betacam SP player. Local professional video stores that supply the video professionals in your area usually carry these players and are willing to rent them to you, although at a fairly high daily price. Additionally, they may offer the digitizing services as well if you would rather not do it yourself.

You will also need a high-quality digitizing card capable of at least 30 frame per second capture. A wide range of digitizing cards is currently available for the Macintosh and many include software, although if you have Adobe Premiere you will be able to capture video sequences directly to QuickTime format. A good choice is Radius' VideoVision Studio card, which provides a number of input and output formats.

SuperVHS player/recorders are available in affordable configurations at most consumer electronic stores. You may also be able to find a professional video store nearby that rents SuperVHS if you do not wish to purchase a SVHS VCR.

Regardless of the video format, a large hard drive is required to store the captured images. If additional editing is required, even more hard drive space is needed. Often a one or two gigabyte drive is large enough to store video images. AV hard drives, which continuously calibrate themselves, have dropped in price dramatically within the past year or so, and may also help prevent video stutters that may occur on hard drives that calibrate themselves periodically. Check your local computer supplier or mail order company to determine the price for AV hard drives if you plan to do a lot of video digitizing. For small jobs, and assuming you already have a fairly fast hard drive, you may not require this functionality.

Video you have digitized is rarely directly usable without some work. Often it's necessary to crop the movie or mix it with other sound effects, music, text, or graphics. Reducing the colors and video compression may also be necessary. If you are unfamiliar with the tools and methods of doing these tasks, some practice and experimentation ahead of time allow you to better estimate the time to process video. In some instances, the process is extremely time-consuming, so approach the subject with caution. If each frame of the video needs to be retouched by hand, for instance, it may become a very expensive process.

If you are compositing a video source with a static background that must later match a section of the background within your game, you will face challenges in matching the colors exactly, especially if a compression method is used. Although compression allows a higher

frame rate for QuickTime movies, it also has an adverse effect on the appearance. A "boxed" movie, separated from the rest of the screen's graphics by some type of border, is much easier to implement.

The largest concerns in implementing video, therefore, are the formats of the original graphics and sound, the availability of the required equipment, and the final method of displaying the video. Plan carefully for each stage of the process. Fortunately, there are a number of references available that discuss the preparation of QuickTime video, the tools that are available, and techniques to ready your video source material for inclusion in a multimedia application or game.

Memory Management

Managing available memory can become a daunting task, especially if you are involved in converting a game from a platform whose operating system does not take up a significant amount of available RAM. You want to support the widest range of Macintosh configurations, and yet still have a great game. How do you convert a program that uses one more megabyte than is available on a 4 MB Macintosh and the System is using 2 MB of that? Well, you can and you can't.

Hopefully, the version of the game you are converting does not compress any graphics in memory and there is no reason not to. Problem solved. Or maybe there are things loaded into memory that do not need to be in memory. Maybe your problems can be solved in obvious ways. Maybe they cannot.

If you are converting a game with a compact, efficient memory management scheme that is pointer-based, you may wish to use it. In allocating each handle on a Macintosh, you are incurring overhead to the tune of about 32 bytes, give or take. For thousands of discrete data chunks that are loaded in a specific order (and needed at specific times), this becomes an enormous waste of resources. Perhaps you can allocate one pointer through the Memory Manager and fill it up with small pieces of data.

A priority system for memory allocation is another method of determining what needs to be in memory and what does not. Assign a priority to your graphics, sounds, and musical scores, and release the ones with the least priority when memory becomes tight. You could create your own memory allocation function that "preflights" the allocation call by calling `MaxBlock()` and comparing the returned value to the amount of memory required. Or it can be accomplished through a `GrowZone()` function, documented in *Inside Macintosh: Memory*.

Another method of finding more RAM is to kill the Finder and allocate System memory via `NewPtrSys()` or `NewHandleSys()`. If you do so, free it as soon as possible. This is a slightly sticky area, so be careful. Always ask the user first before taking such a drastic step. The *Game Developer's Handbook*, prepared by Apple Computer, contains a discussion of this issue as well as sample code for forcing other running applications to quit.

Although handles are recommended when allocating memory because they can "float" in memory and thus avoid fragmentation problems, using them is a two-edged sword. At certain times it is necessary to lock handles so they do not move out from under you. Prior to doing so, it is recommended by Apple that you move them high in your application by calling `MoveHHi()` or by calling `HLockHi()`.

This one function call, however, can rearrange your entire application heap, or very nearly so. The result can be a noticeable delay that slows the execution speed of your game. It might be advantageous, therefore, to evaluate which of your application data should be allocated as pointers and which as handles. In many cases, data such as graphic images, which are accessed during the time-critical sections of your game, are best implemented as pointers. Data and sounds that are accessed infrequently are probably better implemented as handles.

One of the best sources of information about low-level memory management is Scott Knaster's book, *How to Write Macintosh Software*, now in its third edition. Scott also explains a lot about debugging the Macintosh and different techniques that may be applied when those difficult-to-find crash bugs occur.

File Management

The file system of the Macintosh is a curious blend of old and new. Apple's efforts to keep the filing system backward-compatible to the old days prior to the HFS (Hierarchical Filing System) have a noticeable effect on the complexity of low-level calls to the File Manager. This is simplified somewhat with the "File Spec" calls that were added in System 7.0. While we will not discuss the entire subject here, there is a great confusion to most Macintosh programmers as to the exact meaning of a "directory ID" versus a "directory reference number," as well as the myriad of ways in which to specify a file to access.

If you do not need to support System software older than 7.0, you will have a much easier time. By using the new `FSSpec()` calls, you are saved a lot of grief and can easily access files in a number of ways, including by relative path name, by full path name, and by volume/directory ID combined with the file's name. If you are accessing files on a CD-ROM, the process is even simpler because a full path name is almost always used.

Unless there exists a compelling reason not to do so, you should convert any calls to the standard C file access functions, because they make use of the Macintosh File Manager anyway. Also, mixing the Standard File package and the standard C functions is not that easy to accomplish. With the release of *Inside Macintosh: Files*, Apple has provided a wealth of sample code that discusses different methods of file access which simplify the subject enormously. Once you understand the basic calls, the conversion process becomes fairly straightforward. Calls to `fread()` are replaced by calls to `FSRead()`. `fopen()` is replaced by `FSpOpenDF()`, or open data fork.

If you cannot find a file you are looking for, the Macintosh provides a method to allow the user to find it for you. The Finder has long made it easy for users to move or rename their files, and they often do. An average DOS user, on the other hand, moves or renames files much less often due to the complexity involved. The Standard File package compensates for this capability.

The function calls `StandardGetFile()` and `StandardPutFile()` allow the user to browse all mounted hard disks, CD-ROM discs, and floppy disks, as well as networks and remote volumes. This adds the capability for the user to locate a file, and costs you only one or two lines of source code. The returned `FSSpec` record may then passed directly to other file system calls to access the specified file.

There are a number of lower-level file access functions that you may use from time to time. If an asynchronous read is needed, for instance, you need to use the lower-level `PBRead()` function. These lower-level functions require a bit more work on your part in setting up a "parameter block" structure that is then passed to the function. There is no substantial speed or memory penalty for using the higher-level functions, so use them if possible.

Graphic Glitches

A common method of restoring graphics while running under DOS is to copy graphics back from screen memory to an off-screen buffer. Both the Macintosh operating system and Windows make this technique unusable for most applications. If the game you're converting uses this method, you will need to make some modifications. Even if you do not allow your game's window to be moved by the user, you cannot guarantee that your entire window is constantly displayed.

A number of INITs, control panels, and other user goodies allow Macintosh users to enhance their systems, but some of them operate without your program's knowledge. Screen savers, screen capture utilities, on-screen clocks, and other INITs all interfere with the display of your graphics. What you just drew to the screen—or copied there—may no longer be displayed. Some monitors, such as Apple's 17" multiscan monitor, even allow the screen resolution to be changed while your game is running. Imagine suddenly having only a fourth of the screen real estate you had at your last blit to the screen!

Furthermore, if your game supports anything other than a single bit depth of color, or grayscale, your graphics may be in an entirely different format once copied to your on-screen window. And there is no reason not to support grayscale, especially the same bit depths for which your color support is designed. Practically no source code changes are needed at all, even if you have created a custom blitting routine. There are a whole lot of users out there toting their PowerBooks through airports just so they are not so bored during their crosscountry flight. Wouldn't it be nice if they could play your game?

CD-ROM Considerations

A hard disk can transfer several megabytes of information per second, depending on which CPU and Macintosh model you have. A CD-ROM disc, in comparison, is merely a very large, very slow storage medium. A typical CD-ROM drive can transfer only about 300 kilobytes of data per second. That's quite a difference. Additionally, a CD-ROM drive is comparatively slow in moving its read/write head to the position on the disc where it will begin to read data. For this reason, the positioning and layout of your data on the CD-ROM becomes very important.

There are a variety of things you can do to enable your game to run faster from a CD-ROM. By compressing data you can shorten the amount of time it takes to read data from disk. Arrange data in the order in which you will read it from the CD-ROM to minimize the "seek" times. Combine multiple files into one single file to avoid the seeks into the CD-ROM's directory or desktop file. All of these methods, when used together, provide some time savings when reading data, music, or graphics from a CD-ROM.

There are a number of references that provide information about how to create a title that is to be distributed on a CD-ROM. Included among those is a handbook that was written at Apple Computer entitled *Apple CD-ROM Handbook*. It includes instructions, given step by step, that explain how a CD-ROM disc is created and discuss the required software and hardware.

The handbook makes a number of practical recommendations, covering such topics as running a Macintosh application from a CD-ROM and accessing data on a CD-ROM disc. For instance, if your application cannot be run directly from the CD-ROM disc, it is better to include the application as a self-extracting archive. This ensures that the user will install the application on a hard drive or other writable media. The use of full path names when accessing files on a CD-ROM disc is also suggested by Apple.

Disc Layout

The two main factors in seeking and reading data from a CD-ROM disc are the location of the data relative to the read/write head's current location and the length of the data to be read. For this reason, it will increase your game's performance dramatically if your data is stored on a CD-ROM disc in the order that it will be read and if your data is compressed. Additionally, storing subsequent chunks or sections of data in the same file also increases the performance of your game.

To arrange your CD-ROM disc correctly, you must determine the order in which you are accessing your data files. There are a few schemes that make this a relatively easy matter. At the beginning of your game, open a "log" file using `fopen()` or `FSOpen()`, whichever method you prefer. Then, as each file or section of data is read from disk (you could be using a hard disk at this point), log the entry of which file and section was read. Write this information to the log file you created. At the end of your game code, close the log file.

Now play through your game as you would expect a normal player to do. Finally, after accessing most of the data contained in your game (hopefully you have installed cheats to help you quickly play through your own game), quit and review the log file. If you chose to merely write out a text file, you will have to move all the files to the "staging" hard disk by hand just prior to creating your CD-ROM disc. If, however, you create a structured log file, you can then create a utility that copies the files in the order of access listed in your log file (perhaps rewriting the same file in several places, if you have the room).

The best method is to create a few very large files that contain all of your game's data in the exact order in which it is read when a user plays. Of course, the random nature of games sometimes makes this impossible. You'll have to find some compromise between creating a single file and storing each piece of discrete data in its own data file. This will depend upon the structure of your game and the need for accessing large numbers of random data chunks in an undefined order. Simply keep in mind that the closer to the ideal you come, the faster your game's data will be loaded from the CD-ROM.

Often, a partial solution is reached. Data that can be randomly read during game play can be installed on the user's hard drive. This is possible if the data is not overly large and the user chooses to install it. Requiring such an installation should be reserved for "life or death" situations only or for very small amounts of data. Should your game discover that the expected data is not available on the user's hard drive, simply read it from the CD-ROM disc instead. The user has chosen hard drive space over speed. So be it.

Compressing Data

Data may be compressed in a number of ways, depending on the nature of the data itself. Graphics, for instance, may be compressed by a variety of schemes depending on the type of graphic. Video compresses best when a "lossy" type of compression is used, that is, the compressed data "loses" some of the original data in order to make it smaller. Typical cartoon-style graphics can be easily compressed without data loss using a run-length encoding algorithm or an algorithm based on the work of Lempel and Ziv (such as LZW or LZSS). Run-length encoding typically can be decompressed much faster and is therefore heavily used when speed is a concern, although the compression rates are often far below the results of LZ compression.

Compressing sound is usually accomplished using a lossy compression. The Macintosh System software supports sound compression in the 'snd ' and AIFC formats, including 2:1, 3:1, or even 6:1 compression. The better the compression rate, however, the worse the sound fidelity when it is played. Imagine a sliding scale labeled "Sounds Great" at one end and "Really Tiny" at the other.

A variety of compression algorithms are available on the popular online services in source code form. Check carefully to ensure that the author has not retained the rights to the source code and that some statement allowing you to use the source code in your game is included. Otherwise, a variety of licensing fees and credits are required to include the code in your own products.

Commercial compression libraries are also available from a variety of vendors. The quality of these libraries varies from those available in the public domain to professional, clean source code that you can modify and use in your game within a few minutes. Generally, a licensing fee or one-time charge is required to include the library in commercial or even "shareware" products.

Shipping the Title

Assuming that you have kept your team working hard, working smart, and have gotten the game converted, all that remains is fixing the programming errors, touching up any artwork problems, making any last adjustments, and shipping your product. Often this is the stage of a project when you are emotionally lowest, the most tired, and least able to cope with the last bit of effort required. It is also the last ten percent of effort that usually makes the biggest difference in having a high-quality game or a mediocre one. Getting rid of the last few bugs, fixing any interface problems, and putting in the final touches make a big difference in how the game feels to the player.

Fixing the Bugs

Debugging a large program is akin to witchcraft, ESP, and detective work all rolled into one. Intuition and experience help immensely in finding program bugs. There is no recipe or simple outline to follow when debugging, although there are a few useful tools to make debugging a bit easier.

Source-level debuggers are almost invaluable in their ability to allow you to quickly walk through functions and watch as variables are assigned new values, the ability to set conditional breakpoints, and the ability to find logic errors. Out of memory, heap fragmentation, wild pointers, and other types of bugs are best found using a low-level (assembly-style) debugger such as Macsbug. A working knowledge of the Macintosh application environment and assembly language will assist you in using such low-level debuggers. Knowledge of where your globals are really stored in memory (yes, the exact address!) or where your jump table is located is essential. Read the sections of *Inside Macintosh* that describe the A5 world (*Inside Macintosh: Memory*) and the way memory is organized. This knowledge allows you to make better initial guesses about what is going wrong when it does go wrong.

When a bug does occur, the first thing to discover is when it really occurs. Often this is difficult because a random crash may not happen every single time a certain sequence is performed. In those cases, it may be necessary to place "pressure" on the application. For instance, what happens when the memory partition allocated to the application is lowered slightly? Then dramatically? Turn on the Discipline feature in Macsbug or TMON. Are the correct values being sent to Toolbox routines?

When does a value go bad or a variable contain an invalid pointer? One method of narrowing in on a bug is the binary search. Divide the program in half (or some sequence of it).

Which half does it occur in? Of those halves, divide again. Keep dividing until you find where the value is stored incorrectly. Commenting out source code or sections of it can also provide a tool in determining when a value is inappropriately changed.

A good deal of bugs in Macintosh programs are related to undetected Toolbox errors, insufficient memory errors and memory leaks, improper use of dereferenced handles, and simple mistakes. One of my favorite bugs—and I write it into practically every Macintosh program I write—is the following:

```
void myFunction (void)
{
   GrafPtr savePort;

   GetPort(&savePort);
   ...function body...
   SetPort(&savePort);    // Ooops -- should be SetPort(savePort);
}
```

The problem is that this particular bug will not blow up until later, such as when you try to draw into the current port—which makes it extremely hard to find. By turning on the warnings on your compiler, however, this bug can be caught by the compiler. As I mentioned earlier, turn on all your warnings. And then read them.

Creating a debug window is also a good idea. Although it takes a bit of work to create the support functions that draw and scroll text, sometimes it's worth the effort. A debug window can contain any message you wish and can help you track down your bug quickly. By placing conditionally compiled calls in your code, you can trace the functions that are executed, the value of parameters and variables, and the return values of calls to the operating system.

You'll need at least several functions including, at a minimum, `debugOpenWindow()` and `debugSPrintf()`. `debugOpenWindow()` merely opens a new window and initializes any needed variables. The `debugSPrintf()` function can be modeled after the `sprintf()` function in the C library, to which a variable number of arguments can be passed. It can in turn pass these on to `sprintf()` and then draw the resulting string into the debugging window. By keeping track of how many strings you have printed in the debugging window and scrolling the contents upward when necessary, you can quickly implement an invaluable tool for finding the bugs (or at least finding which routine you've crashed in).

An alternative to a debugging window is a debugging file. `fopen()`, `fprintf()`, and other standard C Library functions can be used to open a file and write out debugging information which can later be analyzed. You may have to call `fflush()` to flush a debugging file often, especially if a crash bug exists. This method slows down a game more severely than a debugging window, but provides data which can be meticulously reviewed at your leisure. The implementation of a combination debug window/file can be effective in both ways and 's fairly straightforward to implement.

Polishing the Application

Perhaps this section should have been titled "Spit and Polish," representing the amount of work it really takes to deliver a highly "polished" game. Polishing your game should occur in three distinct areas: game feel, interface, and final cleanup.

The first area to concentrate on is how the conversion "feels." This intangible, highly subjective area is best addressed by obtaining as much information as you can from a number of playtesters. A number of issues must be addressed: How does the mouse or keyboard respond, what do users expect the program to do that it does not, and so on. Final speed optimizations also usually occur during this phase.

Ask a few people you trust to play your game and give you their feedback. After they play your game, ask detailed questions about the "feel" of the mouse and the responsiveness of your program. Is it fast enough? Do the menu items make sense? Or the keyboard equivalents? Is the game sensitive to individual users, and can it be customized?

The converse argument is do not get overly excited when people do not agree. If one person tells you the mouse makes your game move too fast and another person tells you it is too slow, make your mouse control adjustable. Unless, of course, the problem can be solved by the sensitivity controls in the Mouse control panel. Remember that it is better not to reinvent solutions that already exist.

The second area to polish is the interface the game presents to the user, and you should not address just the Macintosh-specific portions of the interface. Are there improvements you can make to the original interface that make sense in light of the Macintosh conversion? Obtain feedback on these issues from your playtesters as well.

Next, polish the interface that you have added during the conversion. Make certain that all your alerts and dialogs are consistent in their placement of buttons, icons, and graphics. Next, ensure that the usage of text and default selections is correct (based on whatever rule set you adopt—safest, most obvious, or least obtrusive). Always have someone else walk through all the text included in your game to correct misspellings and grammatical errors. Preferably, have more than one person proofread your text.

Ensure that the text clearly conveys the appropriate meaning. I have actually seen the following in a Macintosh dialog: "Is it OK not to save the file named 'YourDataInJeopardy'?" with OK and Cancel buttons. As it turns out, Cancel would save the data, which seems completely ludicrous to me. But then, I do not use that program any more. Whatever it was.

Take a last look at your interface. Do things work in a Mac-like way, with buttons that highlight when the mouse is clicked within the button, unhighlight when the mouse is moved outside the button, and only register a "hit" when the mouse up event occurs within the button area? If not, spend some more time polishing your game. Believe me, it is not yet ready to ship.

The final part of polishing your game is the most consistently overlooked and probably as important as the other two. I call it the "final buffing."

As you get closer to delivering your game, look carefully at your product. Are there pixels flashing on the screen where there shouldn't be? Fix 'em! Is there a lot of flashing going on, especially in places where text is displayed? Then draw the text off-screen and blit a whole rectangle to screen at once. Does the cursor disappear when it should be visible? Do buttons flash continually while the mouse is down? Is the menu visible when it shouldn't be? Does the menu bar flash too often because you are enabling and disabling menu items? Does the program crash if you install it on any generic Macintosh just out of the box?

One thing to remember—especially if you have never shipped a game before—is that there are game reviewers on the other end of your project schedule. And generally they are not paid to praise a game that they do not absolutely love playing. If you expect good reviews, work hard to ensure that you've poked, tweaked, fiddled, and polished as much as possible. Game reviewers are almost certainly going to address the one area you do not address in development. And one negative article can be read by as many as half a million of your potential customers.

A quotation to keep in mind is that "You can't please all the people all the time." This is absolutely true. Do not plan on everyone responding enthusiastically about your game, even if you spend two years making sure everything in your game is just perfect. I have literally read reviews of flight simulator games in which the reviewer complained that although the game was great, he did not really like flight simulator games—and he spent several paragraphs expounding on the subject.

The bottom line is to work to make your game the best it can be, hope for the best, be prepared for the worst, negotiate for more royalties for your next project, and then start writing your next game.

Let's move on to Eric's section of the chapter....

Selected Problems and Solutions

The goal of this section is to explore the programming issues that present the greatest challenges in bringing games to the Mac, and to provide at least partial solutions. With a focus on issues concerning technological barriers, we will see how even the strengths of the Macintosh have made it a difficult platform to write games for, and how recent advances in hardware and software have begun to provide solutions.

Although many of the concepts discussed apply to game conversion from a variety of platforms, the emphasis will be primarily on conversion from IBM-compatible computers, and the acronym "PC" will refer to those machines.

CPU Issues

While this section is certainly the most bland and unexciting part of the chapter, it is also the most fundamental. None of the graphics, sound, or control solutions will be of any help until the game at least compiles. Once the game is at least trying to run, then we can move on to the fun parts.

Any computer program, once removed from its native home platform, is bound to experience changes resulting from its move into a new operating system. This is especially true of games, and even more so when the platforms are as fundamentally different as Intel and Motorola. The issues discussed here are at the core of each program and bring out some of the fundamental differences between the two processors.

Assembly Language

If the game you are porting to the Mac has no assembly language in it at all, then you should skip ahead to the next section. Everyone still here? Thought so.

Let's face it, assembly language is used more in games than in any other type of computer program. Why? Choose your two favorite reasons from the following list.

- Speed is almost always critical. Most games rely on fast response and smooth animation. With most computers, this requires a good assembler and an intimate familiarity with the machine.
- Games typically perform many tasks at once. With graphics, sound, control inputs, and game logic running simultaneously, some of these processes are bound to be handled during interrupt. Most interrupt routines call for some degree of assembly, either for efficiency or for interrupt/noninterrupt register coherence.
- It is unlikely that most games will require a long-term upgrade path, as a word processor might. Games are typically written, debugged, and released (hopefully in that order) as works of art and entertainment. As with any other entertainment medium, such as music, books, paintings, or motion pictures, a work is typically presented as is, without the promise of future improvement or modification. Because of this, game programmers are often less careful about unnecessary assembly than they should be. As soon as the current project is over, they will move on to the next one.
- Many games programmers are propeller-heads who just enjoy writing assembly language programs.

Now, just because the original version of the game has assembly code strewn all over its insides doesn't mean your code has to. In fact, it is extremely likely that your program's speed bottlenecks will not be the same as those of the PC version. For this reason, you should consider porting the assembly language code straight into C and...

Keep a Working Nonassembly Version

If the game's original PC programmers had done this, you would not even have to read this section. The concept is simple. If your game's high-level language is C, keep a C version of your code in good working order as you translate the game's assembly language routines, no matter how simple any particular routine may be. Every time you modify your assembly code, modify the C version and test it to make sure it still works.

While this may seem tedious, it is guaranteed to bring you the following sanity-saving advantages:

- Ease of debugging. By shifting back to the "all-C" version of a routine, you can tell if your assembly code is causing problems.
- Better understanding of the code. Bringing code from assembly language to C can often give a programmer better insight into the task the assembly routine hopes to accomplish. With this knowledge, you may find that there is a more efficient way to meet the same goal.
- Optimization options. For the newer PowerPC chips, the compilers can, by balancing processor pipelines, actually write better assembly code than you can.
- Instant 68K/PowerPC compatibility. Instead of needing to write two assembly versions, you can write a single C version which works on both machines and then optimize only where necessary.
- Backward marketability. When the PC programmers start on the protected mode or Windows version of their program, you can sell the C code back to them. They'll probably need it.

Memory Segmentation

While it is not the intent of this chapter to discuss antiquated memory structures, there are some things you should know about the way memory is organized in PC programs. For PC programs which operate in "protected mode," memory segmentation is less of an issue, so the following mainly applies to programs written in "real mode."

If you have always been a 68K or RISC programmer, consider yourself fortunate. These processors typically deal with RAM as one contiguous block, using a 32-bit pointer to identify any single byte within a four gigabyte possible address space. Within this simple, elegant model, a pointer is a pointer is a pointer.

With PC programs, memory is not one contiguous block, but a jillion individual 64K chunks. This means that most PC games do not have the option of allocating a data area larger than 64K for any reason. If they need to store more information than can be contained in one block, they allocate multiple buffers and deal with the partitioning on their own.

Due to this segmentation of memory, pointers on the PC come in different flavors, depending on the maximum range needed. The keywords used in C programs for the different types of pointers are `near`, `far`, and `huge`. If your game was originally written in real mode on a PC, you are likely to find these keywords all over the place, and your compiler has no doubt already complained.

Solution: Ignore It Completely

One thing you can be thankful for is that you are moving the program from a segmented system to a contiguous system and not the other way around. The all-in-one solution to this problem is bound to be the simplest task in your entire project. Repeat after me:

```
#define near
#define far
#define huge
```

The above lines should be added to one of your program's main header files, so that they will be available to all source files in the project. As long as you do not use these key words as variable names somewhere, the standard C preprocessor will eliminate them from the code without disturbing anything else. From there, you can go about your business, forgetting about memory segmentation altogether.

You may, however, wish to use this as a chance for optimization. Armed with the knowledge that the original program was unable to allocate more than 64K at a time, look for places where extra work had to be done in order to break up a large buffer. If you are careful, you may be able to strip out a few lines of code, resulting in a speed increase.

Byte Order

A consistent pattern which has emerged in the Mac/PC, Motorola/Intel relationships can be characterized by absolute disagreement over each of the most fundamental issues in computer science.

"Endian-ness" is a characteristic which divides processors into two categories, depending on the method used to store 2 byte and 4 byte numbers in RAM. A processor which is "big-endian" stores the 32-bit number 0x01020304 in memory as 01, 02, 03, 04, so that the digits are stored in consecutive addresses in the order shown. Seems logical, no? It is merely a convention. The other type, the "little-endian" processor, stores its data in memory backwards. The number 0x01020304, when viewed as consecutive bytes in memory, comes out as 04, 03, 02, 01. As the processor of each type reads and writes memory, it deals with the data in its own, native manner, never giving a second thought to what the order of the data might look like to a human (see figure 11.1).

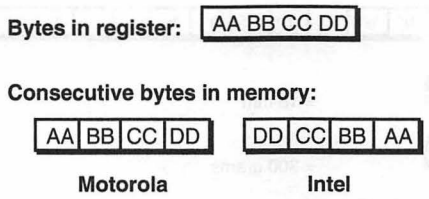


Figure 11.1 *Byte order: Intel vs. Motorola.*

As you might expect, Intel and Motorola decided long ago to be on opposite sides of this very tall fence.

Whoa! What do you mean the bytes are in the opposite order? Why does anything ever work at all? How can the same "Hello World" program compile and run on two computers with such a fundamental difference? Wouldn't the program crash the first time it tried to use a variable?

Not really. The truth is, unless you are porting across platforms, you would never even notice. When you're writing C code on a PC platform, the local and global variables are treated normally. The processor writes all variables into RAM the same way it read them in, so it never sees them as backwards.

So, why is there a problem at all? If each computer deals with the data in its own way, where does the mix-up occur? The mix-up occurs when reading and writing files. When a Mac tries to read a file that was written by a PC, some of the data (but not all) will appear in a goofy order.

Consider the following example. Our game's protagonist, Albert the Slug, is represented by a C-style structure in memory.

```
typedef struct our_hero {
   char name[16];
   short  height;
   short  weight;
   long elasticity;
   char can_fly;
   char can_waterski;
} our_hero;
```

In the PC version, Albert is stored as shown in figure 11.2.

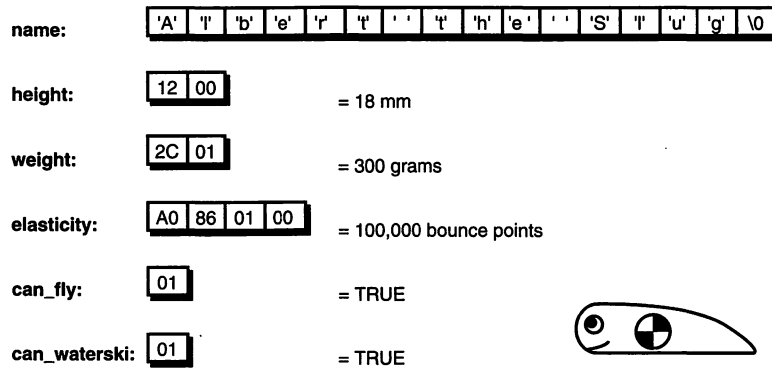


Figure 11.2 *Albert the Slug, Intel format.*

Now, the PC programmers, deciding that they like these particular qualities in a heroic gastropod, save this structure into the game's data file.

Aha! Warning. Warning. The data was saved, byte for byte, in exactly the same order in which it appeared in memory. Now, given the task of porting this game to the Mac, you take their C code, compile it, and run it. The structure, loaded verbatim into Mac memory, appears as shown in figure 11.3.

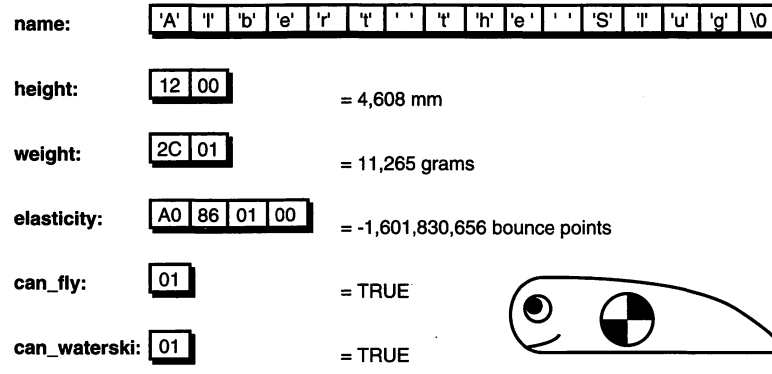


Figure 11.3 *Albert the Slug, Motorola format.*

This is a very different slug. Although some of the variables are exactly as you would expect, others are hideously twisted.

Of course, you could easily write a new `SlugLoader()` routine to swap and fix the affected values, but what happens when someone changes this data structure, changing the names, or worse, the sizes of the fields? What happens when you have hundreds of other animals to consider as well, each with its own data structure, inside of which some of the data is fine and some is wrong? It could happen, and you do not want to modify hundreds of different loading routines in order to keep them all straight.

Solution: Preprocess the Data Files

One way to avoid this problem is to make sure all of the data is stored on disk with the Mac byte ordering, as it would appear in the Mac's memory. This way, only a single, straightforward loader is ever needed.

Be careful, though. While this may seem like the easiest way around this fundamental problem, the solution itself can present the following new headaches:

- Loss of crossplatform data. The files are no longer the same across machines, so you must forever keep a distinction between the Mac and PC data.
- Complication of game tools. If the tools that write the game data are required to know Mac and PC formats, they may be more difficult to maintain or work with.
- Version control. When a small change or bug fix is added to the PC version, the Mac will not be able to benefit from it without some effort.

Of course, if you are already resigned, for whatever reason, to use different versions of the program data on different machines, this may be the solution for you.



Better Solution: Use Macros to Convert Automatically

In order to share as much data as possible with the PC version of your game, you will need to do some extra work. This is another case in which the C preprocessor is your knight in shining armor.

The bad news about this solution is that it will clutter up your nice, elegant code with macro definitions and usage. The good news is that the PC version of your game will be able to reincorporate your changes, if so desired, so that the code will be truly crossplatform. Consider the following definitions:

```
#define SWAP_SHORT(x) (                 \
            (x << 8) |                  \
            (((ushort)x) >> 8)          \
            )

#define SWAP_LONG(x)   (                \
            (x << 24) |                 \
            (((ulong)x) >> 24) |        \
            ((x & 0x0000ff00) << 8) |   \
            ((x & 0x00ff0000) >> 8)     \
            )
```

These two definitions will, of course, perform the byte-swapping operation that you sometimes desire. Rather than placing SWAP directives all over your code, however, you want to do something a bit more clean, such as the following:

```
#if USING_MOTOROLA
#define MOTOROLA_SHORT(x    (x)
```

```
#define MOTOROLA_LONG(x)    (x)
#define INTEL_SHORT(x)    SWAP_SHORT(x)
#define INTEL_LONG(x)     SWAP_LONG(x)
#else
#if USING_INTEL
#define MOTOROLA_SHORT(x)   SWAP_SHORT(x)
#define MOTOROLA_LONG(x)    SWAP_LONG(x)
#define INTEL_SHORT(x)    (x)
#define INTEL_LONG(x)     (x)
#endif
#endif
```

Now, any time you use one of these directives within your code, you are making the declaration, "That number is in Intel format," and each computer may deal with it as necessary.

The final step, if you have an optimizing compiler, is the following:

```
#define MOTOROLA_NUM(x)  (                                    \
   (sizeof(x) == sizeof(short)) ? MOTOROLA_SHORT(x) :         \
   (sizeof(x) == sizeof(long)) ? MOTOROLA_LONG(x) : (x)       \
   )
#define INTEL_NUM(x)   (                                      \
   (sizeof(x) == sizeof(short)) ? INTEL_SHORT(x) :            \
   (sizeof(x) == sizeof(long)) ? INTEL_LONG(x) : (x)          \
   )
```

If you are going to use this, make sure that your compiler will optimize out the conditional statements. The last thing you want is to generate code which checks two constants to see which is greater. This is, however, a nice way to be sure that you do not have to change your code if the size of the data elements changes.

If you really wanted to, you could now label every single variable in your entire program as either Motorola-format or Intel-format. If you did this, your program would be no less efficient, thanks to the magic of the C preprocessor. It would, however, be an illegible mess, and your friends might even begin to avoid you.

Which variables get labeled? The rule is fairly simple. Any time you access a 2 byte or 4 byte value that was loaded directly from disk, it must be labeled.

Interrupt Cycles

The final major processor issue which I have come across in porting games from Intel machines is that of interrupt timing. It is very common for a PC program to require a 240 Hz interrupt cycle. The tasks accomplished by these interrupts are often very minor, ranging from timing functions to musical note sequencing.

Of course, the Macintosh has interrupts too. The tricky part is that they are set up differently and have a different set of efficiency rules.

Solution 1: Approximating with the VBlank Interrupt

If the PC version of your program calls for a 240 Hz interrupt, you may be able to improve efficiency by faking it. Look to see what the interrupt routine actually does. If it simply adds one to a variable named something like `tickTimer`, then you probably want to use a more coarse, but much more efficient, timing method known as VBlank. The name comes from vertical blanking, though nowadays, it's just jargon for a timed interrupt and not connected to monitor refreshing. It's one of those lovely half-acronyms inherited from coin-op video games.

Even the original Mac 128 had the ability to do a quick VBlank interrupt at a nice, regular 60 cycles per second (actually 60.15 cps, but generally referred to as 60 cps). The name is derived from the fact that this interrupt used to be tied to the Mac monitor's scan rate, which was how the original Macs were able to have such crisp cursor motion. This interrupt is also tied to the Ticks system global (returned by the system call `TickCount()`), which increments once every 60th of a second (60.15th of a second, actually, but nobody can pronounce that).

These days, it is common for a Mac to have several monitors, each with a different scan rate, so the original purpose of the VBlank interrupt has lost its meaning. Although the modern Vertical Retrace Manager does allow you to tie tasks to the actual monitor scan rates, these are not generally as useful as the good old 60.15 cycle-per-second VBlank and Tick.



If the PC game relies on having a value incremented every 240th of a second, consider using the VBlank interrupt and incrementing it by four every 60th of a second instead. The efficiency of this method is better than most others. The Mac always invokes its 60 Hz interrupt, whether or not your task is included, so the setup time for the interrupt has already been paid. All you need to do is call `VInstall()` to add your task to the list, and off it goes.

Remember, of course, to have your program delete its VBlank tasks on exit, or they will stay around, randomly and gleefully stomping on memory.

Solution 2: Using the Time Manager

If you really do need a more frequent interrupt, the Time Manager is a nice way to get it. While not as efficient as the VBlank interrupt, it is much more versatile. You can specify in microseconds how often you wish your routine to be called (although the actual accuracy will be a bit lower than that, due to the time it takes to actually execute instructions). You can also change the interrupt rate at will, start and stop the cycles, or whatever else you need to do. Remember, though, that the more time you spend entering and leaving interrupt routines, the less time you have to run the rest of your program.

Graphics Issues

Crisp, high-resolution displays are a primary selling point in today's computers. One rarely hears a computer owner complain that the on-screen image is too clear and sharp. Macintosh users have enjoyed high-resolution color desktops since the first appearance of the

Macintosh II. In fact, most Macs do not even have the option of using a lower display resolution. Unfortunately for game developers, this presents a few major problems.

For the past few years, the canvas on which almost all non-Macintosh computer games operated has been a screen 320 pixels wide, 200 pixels high, and 8 bits deep. Even on computers capable of displaying higher resolutions, the graphics controllers typically shift down to 320 x 200 to play games.

Why this particular resolution? First of all, 320 multiplied by 200 comes out to a nice even 64,000 bytes, which fits nicely into a single 64K segment of PC memory (see the "CPU Issues" section earlier for further discussion of memory segmentation). Second, this is a small enough amount of data, even for low-end computers, to fill with new images each frame without slowing down the game.

The problems this presents for the Macintosh platforms are significant and are responsible for a good deal of the headache which Macintosh game programmers must face. The most significant is that none of the standard Macintosh configurations are capable of shifting into a 320 x 200 display mode. So what's the problem? When the PC game decides to animate the whole screen at once, it needs to write only a quick 64,000 pixels. The same full-screen operation on the Mac requires writing a hefty 307,200 pixels, almost a third of a megabyte.

The second major problem is that, even after doing all sorts of extra work to expand the pixels and after heaving all of that data to the screen, the end result is an image which looks squashed—because of a difference in aspect ratio between 320 x 200 PC screens and 640 x 480 Mac screens.

To top it all off, there is a fundamental difference in the color schemes of the two machines. The Mac considers color 0 to be white and 255 to be black, while the PC has the two switched. Chalk it up to a philosophical difference; the Mac says, "Zero means an empty page," and PC says, "Zero means all electron beams are off."

High-Resolution PC Games

Very recently, some game companies have begun releasing high-resolution (640 x 480) games for PC machines, mostly on CD-ROM. If you are porting a game which already runs at high resolution, take a few moments right now to do a quick dance of joy, as a significant amount of work has already been done for you. Issues having to do with memory usage, drawing speeds, and aspect ratio were dealt with by the original game authors. In addition, the higher original resolution will make your game look significantly more Mac-like than the vast, low-resolution PC game majority. Finished dancing? You may now skip directly to the only major graphics issues which you now face: getting an image to the screen, using color palettes, and possibly direct screen writing.

Getting an Image to the Screen

Before we worry about aspect ratios, pixel doubling, or direct screen writing, it's important and extremely satisfying just to see your game running at all. Getting the first images on the screen is not very difficult, and it can help point out problems in other areas of the program.

After crawling around in PC code for a while, you will probably find that the program expects to write directly into a 320 x 200 section of memory. The quickest way to get things going is to simply point the screen address variable in the drawing routines into your own 320 x 200 memory buffer. This can be accomplished through the following steps:

1. Make sure that the program already compiles and at least tries to run.
2. Have your program allocate a buffer of size (320 * 200). Make sure it is locked and unpurgeable, so that the memory manager won't invalidate your pointer.
3. Look through the PC game code for places where absolute addresses are used. These should be easy to find, as they're probably already causing the program to crash. If you're lucky, there will be a global definition of something like `SCREEN_BASE` which you can change at will. Otherwise, you may find hundreds of references to an address like `0xB00000` which you need to individually replace.
4. Have your program build a pixmap structure, and point its base address to your off-screen map. Once each time through the game's cycle, make a call to `CopyBits()` to transfer the entire image to an on-screen window (see figure 11.4).
5. Test run it, and see what happens.

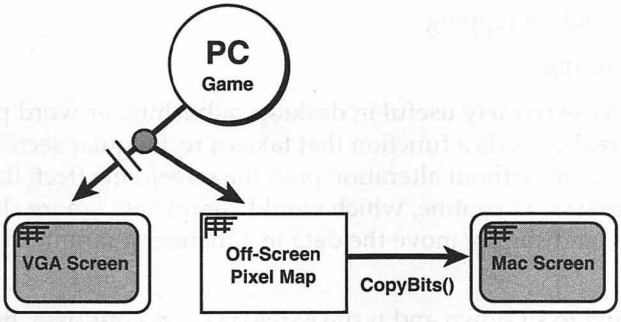


Figure 11.4 *Handing the program an off-screen screen.*

If you see no image at all during the test run, you may want to do some troubleshooting, specifically to make sure that your new display code is being called. More likely, though, you will see an image that has one of the following things wrong with it:

- The game's colors are completely wrong, but you can make out the shapes.
- The colors are almost correct, but poorly matched.
- The image copy is way too slow.

We'll get to color correction soon enough. Right now, the important thing is to make sure that the image on the screen is, byte for byte, the same image as the off-screen pixel map. To determine this, we must find out how much extra work `CopyBits()` is doing for us, and make it stop.

Outsmarting `CopyBits()`

For those not so familiar with a Macintosh Toolbox routine called `CopyBits()`, now would be a good time to look it up. `CopyBits()` is used more often in Macintosh programs than Yellow Dye #5 is used in junk food. From the moment you turn on your Mac, virtually every image that appears on the screen that is not a line, box, or oval passed through `CopyBits()` at some point.

`CopyBits()` is an amazing function. It is an extremely fast, clean, and versatile way to move pixel images to the screen. `CopyBits()` is, however, a bit too versatile for game programming. The following is a list of features that `CopyBits()` gives you automatically, but which your program is unlikely to need and cannot afford to waste time on.

- Arbitrary image scaling
- Image clipping and masking
- Color depth translation
- Color table matching and remapping
- Dithering and anti-aliasing

While these functions prove extremely useful in desktop publishing or word processing, the games programmer only really needs a function that takes a rectangular section of pixels from one map and moves them without alteration onto the screen. In effect, the games programmer needs a "`BozoBits()`" routine, which would completely ignore clipping, image size, and color differences, and simply move the data in a manner guaranteed to be compatible with all screen types.

While it may seem tempting to sit down and write `BozoBits()` on your own, hold off for now. Even if you do eventually use direct screen writing in your program, you'll still need to fall back on `CopyBits()` for compatibility on some machines, so you might as well get it right. Also, it is much easier to debug your game if you know for a fact that screen writing is not causing problems.

So, let's tell `CopyBits()` not to worry about size, clipping, or color translation. As it turns out, these are not options to `CopyBits()`. It will invoke whatever special features it feels are necessary for each particular copy task. The answer, then, is to fool `CopyBits()` into thinking that the extra features are not needed.

Image size is the easiest to take care of. Simply make absolutely sure that the source rectangle is exactly the same size as the destination rectangle. Close does not count. If the two are even a single pixel different, `CopyBits()` will notice and will implement general image scaling, in all of its sluggish glory.

The clipping issue is also a simple one. Make sure that in the destination window there are no obstructions to the rectangular target area. Things to look for are overlapping windows, menu bars, and yes, even those cute little rounded corners at the edge of the Mac screen. You may even want to set the window's `VisRgn` and `ClipRgn` regions to the window's entire rectangle, just to be sure. Also, make sure the window frame you use is rectangular, because some standard types are not. For example, the window used for the standard Calculator desk accessory has rounded corners, which would slow down `CopyBits()` significantly. Finally, be sure to specify NULL in the `MaskRgn` argument to `CopyBits()`.

If `CopyBits()` is copying between two pixmaps of different pixel depths, it will automatically do color translation. In this case, you probably want it to. This will allow your game to play as well as possible on pixel depths for which it was never intended. Be aware that it will be slow. If the game is to play at any reasonable speed, the screen depth must match that of the off-screen pixel map.

The color remapping and translation issue, however, does not stop there. This is the most difficult and insidious problem `CopyBits()` presents to the common game programmer. If `CopyBits()` determines that the color tables of two pixel maps are the same, it will perform a simple, quick data copy. If, however, it suspects that they are different, it will go plodding through the source image, inspecting each pixel and finding the closest match in the target pixel map.

Even when two color maps were the same, I've had instances in which `CopyBits()` decided that they were different. Furthermore, if more than one entry in the target pixmap's table points to the same color, as often happens when using color cycled animations, the results are unpredictable. `CopyBits()` will choose arbitrarily which value to give the target pixel.

Solution: Setting the `ctSeed` Value

The best solution to this problem I have come across is not only cheating, but also silly, underhanded, unsupported, and not guaranteed to work on future Macintoshes. Needless to say, I have used it in all of my games since 1989.

The first check that `CopyBits()` does concerning color tables, which it uses as its "trivial reject" detection, is to compare the `ctSeed` field of each pixel map's color table. Each time a new color table is created, it gets its own `ctSeed`. The seed is changed every time the color table is changed in any way. `CopyBits()` looks at the two `ctSeeds`, and if they are equal, looks no further. It assumes that the color maps are identical in every way, so no translation is necessary.

The following function, called as often as necessary, will keep `CopyBits()` from translating colors unnecessarily.

```
void setSeeds(void)
{
   long seed;

   /* This is kind of an evil thing to do, but it has great advantages. */
   seed = (*((*(macWind->portPixMap))->pmTable))->ctSeed;
   if (macPixMap) (*((*(macPixMap))->pmTable))->ctSeed = seed;
}
```

This is such a fast operation, in fact, that it can be called just before each call to `CopyBits()`, in case something (screen saver, mail program, etc.) has decided to alter the screen's `ctSeed` value.

Fixing the Color Map

Once the program is copying the game's raw pixels to the screen without translation, chances are the colors are now all wrong. There are several ways to remap the colors of the screen, and more than one of the solutions is correct.

Some programmers make use of the animating features of the Palette Manager. This can become slow and confusing, however, and there is a simple approach which has worked on all Macintosh models so far.

Among the most basic routines in the Color Manager is one called `SetEntries()`. When used correctly, it enables strict control over what color is displayed by each individual pixel value. This is the kind of control that VGA-based PC games expect, and it is completely necessary in order to avoid color translation.

Additionally, `SetEntries()` is fast enough to be called often. Even if your PC game makes extremely heavy use of color cycled animations, the cost on the Mac will be insignificant and the resultant image will be the same.

This solution has the added advantage that, if the game is running on a grayscale display instead of color, the matching, cycling, and drawing simply work without any intervention on your part.

Resolution Doubling

So, now you've got the game playing within a tiny 320 x 200 window on the screen. The good news is that from this point, the game could potentially be completed, debugged, and sold as a desk accessory. Unfortunately, any text in the game is probably too small to read, and the game's miniature environment is less than cinematic (see figure 11.5).

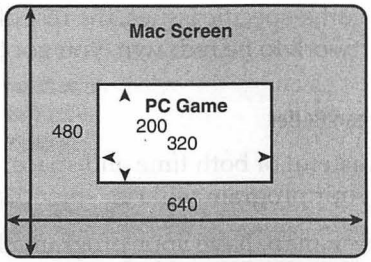


Figure 11.5 *A 320 x 200 window on a Mac screen.*

No problem. Simply go to the Monitors control panel and shift your Mac's screen into 320 x 200 mode. What? No low-resolution option? It's time to do some work.

For those who know `CopyBits()` well, it seems that the obvious solution is to simply use a bigger target rectangle in your `CopyBits()` call and let the Mac Toolbox expand the image for you. Yes, this will work. The problem, however, is that the generalized scaling used by `CopyBits()` is very slow. It will scale the image to any size or proportion you like, if you are willing to wait. As `CopyBits()` moves the image to the screen, you will see a "wipe" or "ripple" effect as the image is expanded and copied. Take note of this effect, in case it may come in handy in your next submarine game. For now, let's try to minimize it.

The ripple effect is happening because the pixels are actually being expanded by `CopyBits()` right as the image hits the screen. In order to avoid this effect, while you're still using `CopyBits()` to get the image onto the screen, the image in the off-screen pixel map must be the same size as the image on the screen. What's the best way to accomplish this when you're starting with such a tiny original image?

Improbable Solution: Redrawing the Artwork

With most games, the artwork is among the most expensive aspects of the development process. Normally, it is not feasible to have the art redone for a Macintosh version, so we need to work from the 320 x 200 off-screen buffer and expand the pixels as needed.

In addition to the extra work of drawing the art, the program would need to deal with memory and storage requirements for images which have at least quadrupled in size. The game begins to become less of a port and more of an original product, complete with a larger budget.

Some games, however, can get a tremendous benefit from a small amount of work by redrawing some of the art. For example, if the game is a flight simulator, the cockpit can be redone in high resolution to provide a nice, clean view, while the outside images are displayed as in the original. Adding high resolution to on-screen widgets, words, and other interface elements can improve the overall look of the game with some small extra effort.

As this begins to delve into special, game-specific issues, the rest of this section will assume that when you asked for all of the artwork to be redrawn, you got laughed at.

Simple Solution: Expanding into an Off-Screen Buffer

The following solution may seem wasteful of both time and space, but as you will see, it is the most robust way to ensure that your program will run smoothly and ripple-free.

In addition to your 320 x 200 off-screen map, have your program allocate a new 640 x 480 pixel buffer (see figure 11.6). This is 80 lines more than you really need right now, but it's best to leave yourself room for aspect correction later.

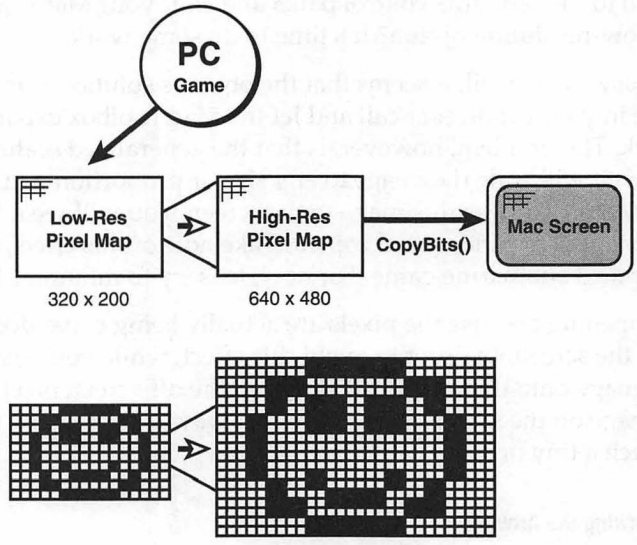


Figure 11.6 *Expansion into an off-screen buffer.*

Each time through the game cycle, have a custom pixel-expansion routine run through the off-screen pixel map, expanding each pixel into a two-by-two group in the destination map. For now, do this for the entire image. Later in this chapter, we'll talk about how to reduce the workload by converting less than the entire image.

The pixel-expander you write should be fairly efficient, since you know that it only needs to expand one-by-one into two-by-two. You can probably also find a way to use lookup tables or increase the number of pixels you expand at a time. This routine is an excellent candidate for assembly optimization when you get around to it, but be sure to keep your C version close at hand, as you'll need to add more capability to it later. The following example of 68K assembly code shows how the expansion might be done in registers, without a look-up table:

```
;; expand four pixels into 8x2 pixels
;; note: PIXEL is defined as d2
move.l (INC)+, d1      ;; get the source 4x1 pixels as one longword
```

```
move.l  d1,d0           ;; make a copy
swap    d1              ;; d1 will be the other two pixels
lsr.l   #8, d0          ;; these two lines "space out" two adjacent pixels
lsr.w   #8, d0          ;;   into separate words.
move.l  d0, PIXEL       ;; copy out "spaced" pixels
lsl.l   #8, PIXEL       ;; shift the copy over
or.l    d0, PIXEL       ;; …and recombine
move.l  PIXEL, 640(OUTC) ;; write the next row
move.l  PIXEL, (OUTC)+ ;; write this row and increment the write ptr.
lsr.l   #8, d1          ;; space out the other two pixels…
lsr.w   #8, d1          ;;
move.l  d1, PIXEL       ;; make another shifted copy…
lsl.l   #8, PIXEL       ;;
or.l    d1, PIXEL       ;; and recombine.
move.l  PIXEL, 640(OUTC) ;; write the next row
move.l  PIXEL, (OUTC)+ ;; write this row and increment the write ptr.
```

When the pixel-expansion routine is finished with the entire image, call `CopyBits()` to get it to the screen. Ta-da! No ripple. (If there still is a ripple, see "Outsmarting `CopyBits()`" earlier in this chapter.)

Medium Expansion for 512-Pixel Screens

When Apple first introduced the Mac LC, it also introduced a low-cost, lightweight 12" monitor. If you say the words "12" monitor" within earshot of anyone who has ported a PC game to the Mac, you are likely to have something thrown at you.

These less expensive monitors have a display resolution of 512 x 384 pixels. While this is still far too large to let the game programmers use the original 320 x 200 resolution, it is small enough that, if the pixels are simply doubled, at least 24 percent of the image is off the screen (see figure 11.7).

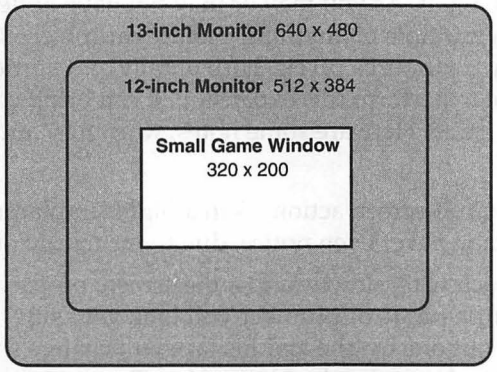


Figure 11.7 *Relative window sizes.*

On the positive side, the introduction of the LC series and the 12" monitors meant that color Macs were finally inexpensive enough to be adopted by families and students, two of the game industry's primary audiences. Unfortunately, they sold so well that a significant portion of the people buying games were playing them on 12" monitors, so the games programmers had to take them very seriously.

There are three major solutions to this major problem:

1. Design the game at 512 x 384 from the beginning.
2. Play the game at 640 x 480 and allow for some scrolling or panning mechanism.
3. Use the dreaded one-and-a-half pixel expansion, bringing the action up to a width of 480 pixels.

The first option is an option only if you are already redrawing all of the artwork. Even when this is the case, it can be extremely disappointing for the millions of 13" monitor users to see the game playing in only a small portion of their screen. Drawing two sets of artwork, one for each of the screen sizes, is usually not feasible.

Option number two, playing the game at a larger resolution and allowing for panning or scrolling, can be suitable under certain circumstances. The mechanism for performing the scrolling is necessarily very game-specific. For example, the Mac version of a board game in which the whole board is visible on a 13" monitor may behave in a manner such that when the cursor touches an edge of the screen, the board quickly zips across to show the hidden portions. If, however, the game has scrolling built into it, be very careful not to confuse the player. For example, if the player is guiding a plumber through a scrolling landscape, adding a second layer of scrolling is likely to be awkward and give the distinct impression that the game was not intended for this player's computer (of course this is true, but it's rude to make it obvious).

If neither of the first two options is useful, welcome to the ugliest scaling factor in the history of computer graphics: one and a half. As you may or may not have noticed, 1.5 is not a power of two, nor is it an even multiple of anything useful. This pixel expansion involves adding an extra pixel for every two source pixels, horizontally and vertically. While it is not very difficult to write a routine to perform this expansion, it can bring countless necessary tweaks and problems into the game. Here are some notes, warnings, and things to keep in mind:

- If your game involves fast, full-screen action, as in a flight simulator or a video player, take heart. Your players may never even notice, due to the nature of the motions.
- If you have characters which walk slowly across the screen, prepare to laugh pretty hard. Chances are, your little pirate or plumber will look very silly. As he strolls around, his eyes will bug out one by one and his face will change shape, as though he's contemplating spitting out a large chunk of lemon (see figure 11.8).

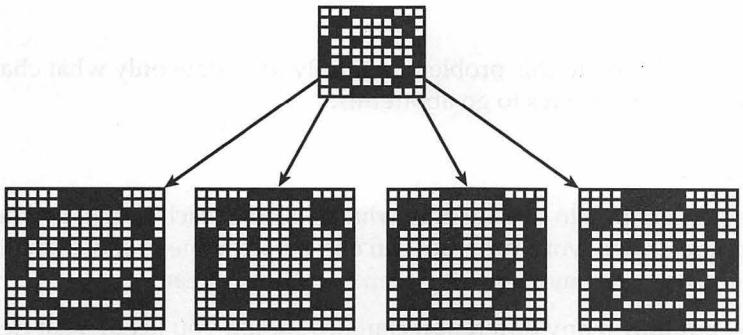


Figure 11.8 *Awkwardly humorous pixel expansion.*

- Text is a problem. If some of the text in your game can be replaced by real Mac fonts overlaid on the game, this will be helpful. Any text which goes through the one-and-a-half expansion will come out looking like it was drawn by a six-year-old. If this is the look you're going for, great. Otherwise, you may want to consider using a fatter letter style or making sure that all text is drawn on even lines, or odd lines, just so that it all gets expanded similarly.
- If there is a static interface element in your game, such as a radar screen or a "lives remaining" counter, consider displaying this at either the original resolution or double size. This can take some of the awkwardness out of the look of the game.
- If you are using line skipping (see the next section on full-screen drawing) to increase the game's speed, be very careful. At the 1.5 scaling factor, half of the pixels occupy only a single line. Therefore, a full 25 percent of your game's pixels will never make it to the screen. If you use these together, some or your text is likely to be unreadable, so make sure you game is well tested and always give the user the option not to use line skipping.

Once again, do not rely on `CopyBits()` to expand your pixels for you. Although it will work, it will be far too slow to be useful.

Full-Screen Drawing

Something you may or may not have noticed by now, depending on the speed of your development computer, is that updating a full-size, 307,200 byte display takes time. On a Quadra or Power Mac, it can be done fairly quickly, but a Performa, any of the LC series, or even the good old Mac IIci won't be able to keep up.

If you're pixel doubling the full picture into an off-screen 640 x 400 image and then displaying the image via `CopyBits()`, then your Mac is shipping at least 576,000 bytes across its bus for each frame, not counting the work it takes to actually expand the image. Compared to the quick 64,000 bytes the PC version of the same game gets to toss around, you can see that it will take a little cleverness on your part to match the game's intended speeds.

Solution 1: Don't Do It

For most games, the best solution to this problem is simply to update only what changes each frame. There are two simple ways to go about this.

Keeping a Change Rectangle

Some PC games are smart enough to update only what is needed each frame. By keeping track of which regions are drawn, your program can end up with one or more small rectangles, which can be drawn much more quickly than the whole screen.

The concept is simple, but how many different rectangles should you keep? That depends on the nature of the game. The simple approach is to keep one rectangle that encompasses the entire area to be updated. This works well if the screen is mostly static with a few moving characters, but presents a problem if you have several small moving objects distributed about the screen.

Cutting the screen into regions is much smarter, but is best done with some knowledge of the specific game. For example, if you have an interface area such as a radar or control panel with blinkies all over it, you may want to cover this area with its own change rectangle and use a separate rectangle for the rest of the screen.

Either way, it will be extremely helpful to take some time to have your program draw a frame around each rectangle it updates, so that you can see how much work it's doing and get a good feel for the efficiency of your partitioning scheme.

Using a Difference Buffer

Not all PC games take well to difference rectangles. Some redraw the entire screen every frame, even though they don't need to, while others have a very large number of small objects moving across the screen. Believe it or not, it can be significantly faster for games such as these to keep a complete copy of the previous frame around, and use it to compare against when deciding what to update during each frame.

As your pixel expansion routine looks at the image, have it take each longword in the original image and compare it against the corresponding longword in the difference buffer. If they are the same, move on to the next group of four pixels. If they are different, copy the new pixels into your difference buffer and perform the screen expansion.

This may seem like a lot of work, but it is much faster than expanding the entire screen each frame, especially if most of the game's screen is unchanged each frame.

Solution 2: Expanding Directly to the Screen

Unfortunately, some games do need to change the full screen every single frame. Flight simulators and dungeon-style shoot-'em-ups are good examples of games in which the PC version is likely to redraw everything all the time.

Games like these are extremely strong candidates for direct screen writing. By expanding the image directly to the screen instead of relying on `CopyBits()`, you will cut your total memory movement by almost half.

If expanding pixels directly to the screen is causing the old `CopyBits()`-style ripple effect, read on. Your solution awaits.

Solution 3: Line Skipping

Once you have decided that your game requires the ability to write directly to the Mac's screen, there is a nifty trick that will speed up your game substantially.

Simply skip every other line on the display, leaving it black. Normally, your game doubles a single PC pixel into 2 x 2 Mac pixels, spanning two lines on the destination pixel map. Each of these lines must be written separately, causing a large amount of bus traffic. Using line skipping, you just ignore the second line. Do not write new black pixels to the skipped lines; just leave them black (see figure 11.9).

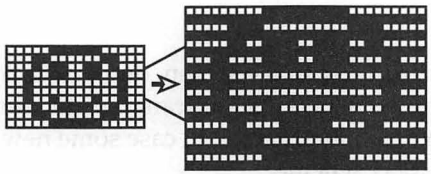


Figure 11.9 *Line skipping for speed.*

The following code is a modification of the standard pixel doubling routine given earlier in this chapter. Note that the only change made was to comment out the second line write.

```
   ;; expand 4 x 1 pixels into 8 x 2 pixels
   ;; note: PIXEL is defined as register d2
   ;; note: some lines are commented out for line skipping.
   move.l  (INC)+, d1          ;; get the source 4x1 pixels as one longword
   move.l  d1,d0               ;; make a copy
   swap    d1                  ;; d1 will be the other two pixels
   lsr.l   #8, d0              ;; these two lines "space out" two adjacent pixels
   lsr.w   #8, d0              ;;   into separate words.
   move.l  d0, PIXEL           ;; copy out "spaced" pixels
   lsl.l   #8, PIXEL           ;; shift the copy over
   or.l    d0, PIXEL           ;; …and recombine
;;;; move.l  PIXEL, 640(OUTC)  ;; commented out for line skipping
   move.l  PIXEL, (OUTC)+      ;; write this row and increment the write ptr.
   lsr.l   #8, d1              ;; space out the other two pixels…
   lsr.w   #8, d1              ;;
   move.l  d1, PIXEL           ;; make another shifted copy…
   lsl.l   #8, PIXEL           ;;
   or.l    d1, PIXEL           ;; and recombine.
;;;; move.l  PIXEL, 640(OUTC)  ;; commented out for line skipping
   move.l  PIXEL, (OUTC)+      ;; write this row and increment the write ptr.
```

This may sound strange, but give it a try. Simple to implement, it can make your game fast enough to run on some of the lower-end Macs. The results will probably be far better than you expect. In fact, the first time I ran one of my games at 640 x 480 with line skipping on, I had shown it to three other game programmers before anyone figured out what I had changed.

Why does it work? Several reasons. First of all, none of the image is actually lost, because the omitted line was really only a copy of the one above it. Second, computer users are accustomed to seeing raster lines in the display, so the eye tends to ignore the horizontal grouping. Finally, the high-resolution nature of the display compared to the game itself helps to hide the missing lines from the eye.

The important thing is that it works, and it cuts your pixel pushing better than in half. When you were expanding your pixels into an off-screen map and displaying via `CopyBits()`, the total data moved, including the initial writing of the 320 x 200 buffer, was 576,000 bytes. Now, with direct screen writing and line skipping, you have the same, full-screen image, for 192,000 bytes. Still a hefty sum, but down by exactly a factor of three. Remember, this can go down much further if you don't absolutely need to update the full screen every frame.

Direct Screen Writing

If your game doesn't need to write directly to the screen, don't do it. You'll save time and countless bug report pages. If your game gets a significant benefit from writing to the screen, make sure you provide the user with an alternative, in case some new breed of Mac should encounter problems with your display scheme.

In an attempt to convey the importance of using `CopyBits()` at least as a compatibility fallback, here's something which convinced me and which you may wish to try for yourself.

One evening, shortly after reaching alpha on a new game, I decided to test it out on a friend's nonstandard Mac IIci. This computer had three monitors: one standard 13" Apple RGB, one 12" grayscale, and an enormous black-and-white-only display. After launching the game, I dragged the window so that it spanned all three monitors simultaneously. The game continued to play. I had not written any one-bit display routines, any clipping routines, or anything to handle color-to-grayscale conversion. As we led our little pirate hero from one display to the other, it became obvious that we would need to start making offerings to the `CopyBits()` gods, so that they might do some more great things for us in the future.

Aspect Ratio

So now the game is playing into a 640 x 400 window and everything looks great, except that the aspect ratio is not right. There are forty empty lines above the image and forty below, and the whole image looks slightly squashed. All of your circles look like eggs, and if you happen to have eggs in your game, they look like circles. Your protagonist is a little hard to take seriously, at four-fifths of his originally intended height.

Why can't the Mac just use square pixels? It does. Blame this one on the original PC art, which was drawn with the tall pixels used by VGA boards (see figure 11.10).

While there are no perfect solutions to this problem, there are a few adequate ones.

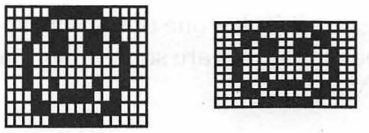


Figure 11.10 *Tall vs. square pixels.*

Solution: Line Padding

Line padding, when added into the rest of your pixel expansion code, can provide a nice touch to your game. The process is simple: For every five horizontal lines you draw to the screen, add an extra one, which is a copy of the one above it (see figure 11.11).

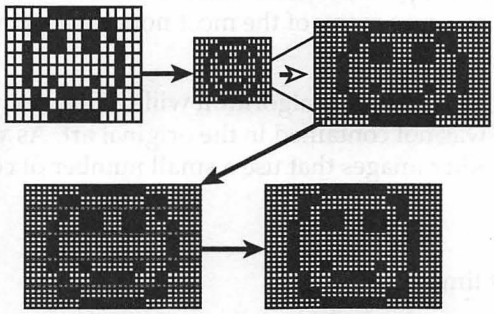


Figure 11.11 *Aspect correction using line padding.*

If the player looks closely, depending on the type of game, the extra lines may be visible. Once in a while, they will really stand out, especially as text hits them, or when this mode is combined with another alteration such as line skipping. More noticeable, however, is the fact that the game now fills the entire Mac screen, instead of the letterbox-looking stripe it used to occupy.

Silly Solution: Adjust Monitor "Vert" Control

Yes, it works. I've tried it. If you reach to the back of your monitor, it is likely to have a vertical scale adjustment. Of course, I know fewer than five people (all of them nerds like me) who would ever adjust their monitor for any game. Still, if line padding slows things down or looks too distorted, this option can at least be mentioned to the user. The most significant disadvantage to using this method is that, after playing the game, it is nearly impossible to get the monitor adjusted perfectly back to its original proportions.

Chunky Pixels

By now, you have undoubtedly noticed that the original, 320 x 200 artwork gives the entire game a chunky, "PC-port" feel. You may not mind. The PC programmers will probably not mind. The customers and game reviewers, however, may clobber you over this issue alone,

depending heavily on what type of game it is. It's one of the sore spots when it comes to playing ported computer games. Blocky graphics are something that these users left behind forever when they decided to buy a Mac.

Solution 1: Redrawing Some of the Art

This option, along with some of its positive and negative aspects, is discussed in the "Resolution Doubling" section earlier in this chapter.

Solution 2: On-the-Fly Pixel Smoothing

There is at least one way to smooth out the artwork as the game is playing. The following is a description of the real-time smoothing algorithm called "epx" (Eric's Pixel eXpansion).

While this solution to the blockiness problem has limited usefulness and is no replacement for high-resolution artwork, it can cure many of the most noticeable symptoms of low-resolution art.

As the pixels get expanded to the screen, this algorithm will smooth some of them out, creating a roundedness which was not contained in the original art. As you will see, it works especially well on text and on other images that use a small number of colors.

Strengths

- No extra art development time
- No extra storage or RAM space
- No preprocessing of data files
- Smoothes text well, removing the original blockiness
- Leaves dither patterns alone

Weaknesses

- Slower than normal pixel expansion
- Makes "guesses" about the art that are often wrong
- Can cause funky side effects, depending on the art
- Not as nice as totally redrawn artwork

Examples

The examples shown in figure 11.12 are taken from actual shipping games in which the algorithm is used to smooth all images onto the screen from a 320 x 200 off-screen pixel map during game play.

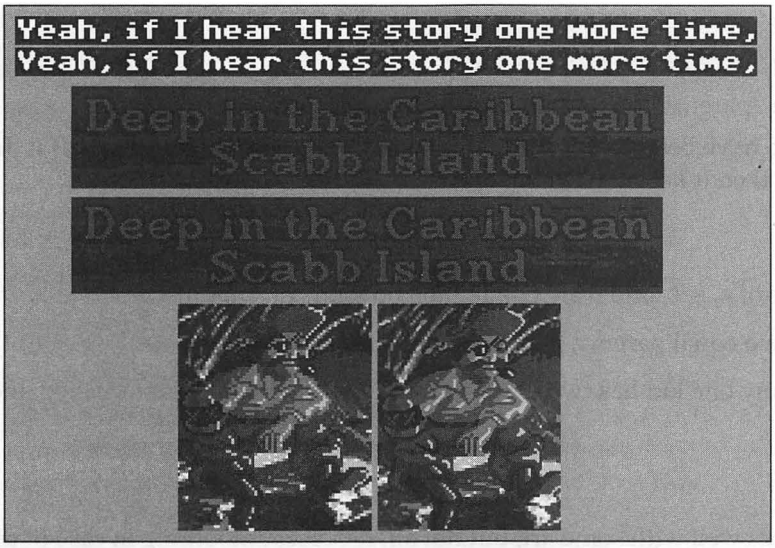


Figure 11.12 *Comparisons of some images with and without epx smoothing.* *(Pictures from Monkey Island 2: Le Chuck's Revenge ® and © 1991 LucasArts Entertainment Company. Used under authorization. All rights reserved.)*

The Algorithm Explained

Figure 11.13 is an explanation of the entire algorithm. There is nothing else to it, unless you can find a way to improve upon it. In order to get reasonable speed, assembly language is mandatory, at least on non-PowerPC machines.

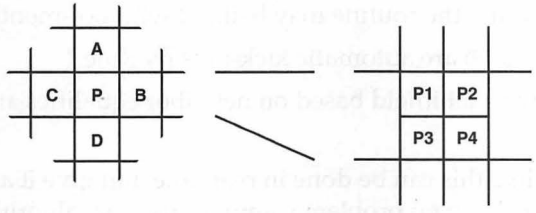


Figure 11.13 *epx pixel smoothing.*

Definitions

- `P` is the pixel we want to expand and smooth.
- `P1`, `P2`, `P3`, and `P4` are the "kids," the four pixels which result from the expansion of `P`.
- `A`, `B`, `C`, and `D` are the "neighbors" of `P`.
- `A` and `B` are the "parents" of `P2`.

- A and C are the "parents" of P1.
- C and D are the "parents" of P3.
- B and D are the "parents" of P4.
- A kid is said to have been "smoothed" if it takes on the color of its parent. If it does not get smoothed, then it keeps the value of P.

Rules

1. If at least three of A, B, C, and D are equal, then no kids get smoothed.
2. If no kid has two equal parents, then no kids get smoothed.
3. Otherwise, if any one kid has two equal parents, then that kid gets smoothed and the others do not.

Optional Rule

(This next rule works well with some art, but not all. It causes smoothing in case of dual diagonal equality. Try it out, but keep it optional.)

2.5 If A==B and C==D, then P2 and P3 get smoothed, P1 and P4 do not. Likewise, if A==C and B==D, then P1 and P4 get smoothed, P2 and P3 do not.

Implementation Tips

- Never ever look at the color table. Use pixel values only.
- Try to keep the previously read pixel in a register to use as a neighbor for the next pixel.
- To minimize the number of compares, notice that much of the time at least three neighbors will be equal. If they are, the routine may bail out with no smoothing, by Rule 1.
- Also notice that A==B and C==D are automatic kick-outs by Rule 1.
- It may be useful to construct a bitfield based on neighbor equalities and then switch based on its value.

I know that it does not seem like this can be done in real time, but give it a try. Several games currently available for the Mac have no problem running this exact algorithm on an original Mac LC, while still playing the game and carrying an eight-voice orchestra (see "Sound Issues" later in this chapter). Quadras and Power Macs will definitely have no problem keeping up.

This technique could also be used to presmooth art as a quick way for the art department to come up with a reasonable high-resolution version on limited time and money. On CD-ROM games, for example, the extra storage space is usually not a problem, and the "funky side effects" can be touched up by the artists.

Colors 0 and 255

It is significant, if not surprising, that the Mac and the PC cannot come to an agreement over the most fundamental issue in the universe: the distinction between black and white. On the PC, the pixel value 0 indicates a pixel of zero intensity, or black. On the Mac, 0 indicates a page which has not been written on, which leaves it white.

After the earlier discussion of color maps, the solution may seem obvious: Simply remap the Mac's color palette to match that of the PC. Maybe you've already tried this and noticed that the Mac cursor, menus, and fonts have a very difficult time with the swap. As they try to match colors, fonts get cached wrong, icons get half-updated, and the nice, solid look of the desktop begins to come apart. Even after your game puts the colors back and quits, there may be lasting effects which will persist until the player restarts or reconfigures the monitor setting. Many of the Mac's optimized drawing routines depend on having stable, omnipresent black-and-white indices.

The effect is even greater if your game plays within a window, not taking over the full screen. As the color entries are changed, the screen will flash, invert, and generally look like an out-of-control music video. Even if you have solved the Mac vs. PC black-and-white problem, if the PC game animates either of these colors, the desktop will animate as well, and it will not look good.

The overbearing rule that applies is the following: In any windowed graphics system, at least two colors (preferably black and white) must be left constant and unanimated, or the windowing system will not be predictable.

Solution 1: Don't Use Them

If you beg and plead and if you're not too late, you may be able to convince the programmers of the original PC game not to use colors 0 and 255 at all. It's worth the humiliation. If you don't have to worry about colors 0 and 255, you can simply have the desktop redrawn in black-and-white at the start of the game and it will stay that way, ready to be of service, until you give it the rest of its colors back, at which point it will gracefully redraw.

Of course this doesn't mean that PC games cannot display black or white at all, because they can simply make use of two of the other color slots. The only restriction placed on the developers is that they must make do with 254 colors instead of 256. If they resist at first, tell them that it will make their Windows version easier to do, and offer them chocolate. If it is really too late to change the project, you should at least make sure that they keep this problem in mind for their next project, which you may also need to port.

Solution 2: Pretend You Never Used Them

If the PC version of the game is already finished or if the PC programmers are just mean and stubborn, you may have a difficult time getting a version of the game that does not use colors 0 and 255. You may still have a way out, though.

If you can take some time to filter the artwork, the two troublesome colors may be eliminated from the game altogether. For example, suppose all of the PC art is done using a PC paint program and then converted, via some in-house tool, into bitmap files which the game engine reads and displays. If you can alter this tool to strip out colors 0 and 255 by matching them to the closest colors in the rest of the palette, you will end up with art that seems as though it were drawn without the offending colors in the first place. The best thing about this trick is that you will not have to bother the artists, who really don't want to hear about color index problems.

Be careful when you use this method, however. Depending on your particular game and the programming styles of the PC programmers, some or all of the following problems may arise:

- Color cycling mismatch. The program uses the color palette to animate and it happens to animate colors 0 or 255. As you have moved those pixel values over to another index, they will not animate when expected.
- Color cycling mismatch (inverse). The program uses the color palette to animate and it happens to animate colors which you have moved the 0 or 255 pixels over to. Because you have merged some new pixels into this group, they will animate when not expected.
- Code-based coloring. The lasers, radar blips, or other unartistic display elements are drawn by writing direct pixel values that are not contained in the art files. It may be difficult to catch all of these, and the retrofit coding may get messy.
- Index-dependent compression. The part of the program which decompresses the artwork uses some slick algorithm which counts on using the low 16 or high 16 colors. This is not as uncommon as it sounds, especially with the increasing use of compressed video.

In general, any quick solution to the above problems will introduce new problems of its own, starting an endless cycle of hack-patch programming. Think through any proposed solution as far as possible before implementing. If it begins to look like colors 0 and 255 absolutely must be used, read on.

Solution 3: Color Map and Pixel Inversion

If the PC game absolutely must make use of 0 and 255, there is a way to at least keep black and white the same across machines. This does not solve the color cycling animation problem, but it does make it simpler.

Your program should, as it draws, simply swap black and white pixels with each other, so that they match up with the Mac's convention. The tedious way to do this would be to check each pixel as it is drawn, looking for 0 or 255 pixels and swapping them. Too much work. This is the equivalent of inserting two `if` statements per pixel.

A slightly better solution would be to keep a look-up table for pixel values and re-look-up each pixel as it gets translated. This involves at least one extra memory access per pixel or pair of pixels. Although this solution is workable, there is still a better way.

A far more efficient solution than either of these, one which also will enable you to deal with multiple pixels at once, is to simply invert all pixels as they are brought over to the off-screen buffer. By flipping all bits in all pixels, 0 and 255 will be automatically swapped (see figure 11.14).

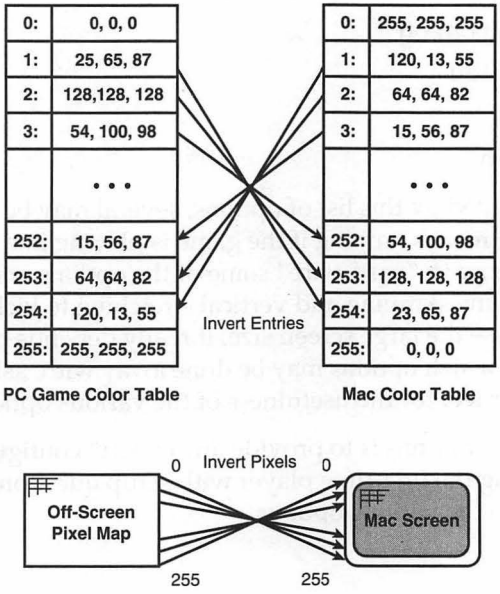


Figure 11.14 *Flipping the pixels and color map.*

So, what happens to the rest of the colors? No problem. When you assign their color table entries, invert the color index numbers except for 0 and 255. This way, all of the colors will match up in the color table, black will be black, white will be white, and no one will suspect a thing.

Letting the Player Decide

No matter how much time you spend writing code to decide which video modes to use, it will be wrong some of the time. The simple fact is that, a few weeks after you release your finished game, someone will release a product that interferes with it. The only way to make sure that the player has the best possible experience with your game is to let the player control some aspects of the display.

Video Options Gone Amok

You may be tempted to give the player access to all of the options discussed in this section. Please don't. Once all of the possible settings are taken into account, we have the following variables:

- Display size: 320, 480, or 640 pixels wide
- Direct screen writing on/off
- Line skipping on/off
- Vertical aspect stretching on/off
- Real-time smoothing on/off
- Desktop shown/hidden
- Menu bar shown/hidden

Of the 192 possibilities presented by this list of options, several may be removed by realizing that, for example, smoothing means nothing if the game is playing in a small window. Furthermore, it may make sense to "hard-wire" some of the options one way or the other. For example, you may want line skipping and vertical stretching to kick in automatically when the player chooses to use the large screen size. It really depends on the game in question. If you can't decide which options may be done away with, ask your testers. They've probably got a better feel for the usefulness of the various options than you do.

The alternative to hard-wiring options is to provide an "expert" configuration screen. This way, you will avoid drowning the first-time player with setup questions, while still giving the advanced player access to the various options.

Sound Issues

Every Macintosh computer, from the largest workstation to the lightest laptop, comes with the off-the-shelf ability to play sampled sounds. Many even come with microphones so that the user can easily record any noise as a replacement for the Mac's system beep. To game programmers, this nice built-in feature is something of a curse.

The game player, after belching into the microphone and hearing it played back, quickly deduces that the Mac can play any sound in the world. Why, then, do the fast-action games on the Mac make such sparse use of music, when games on the PC seem to fill the room with sound?

The answer is that the standard, off-the-shelf PC has almost no sound support whatsoever. The ancient internal square-wave speaker packed into most systems is easily outperformed by most touch-tone telephones. With this knowledge, the PC games enthusiast quickly goes out and buys a sound and music card for between $59 and $599, not including speakers. These cards not only enable the PC to make excellent sound effects and music, but they also take most of the work of producing the sounds off of the PC's shoulders.

Meanwhile, the Mac game player, having deduced that his computer is sonically superior from the outset, never even considers buying an add-in sound board.

Streaming Audio

Not including the various characteristics of the AV Mac line, the sound capabilities of the Macintosh basically boil down to the ability to play a sampled sound at a constant pitch.

Some PC games, especially new ones, are using "streamed audio" more and more. The ability to store sound on a CD-ROM and play it continuously is resulting in a higher use of very long voice and music tracks which are mixed in software.

If the game you are porting is one which uses streamed audio, you are in luck. The PC game expects to have a "DAC" (stands for "Digital to Analog Converter"), a channel to which it can dump a constant stream of bytes, which will play through the speaker. On the Mac, you can give it something just as good.

Solution: Maintaining a Double Buffer DAC

By using the system call `SndPlayDoubleBuffer()`, you can start the Mac on a continuous playback of two buffers which you allocate. The Mac will play one of the buffers from beginning to end and then switch over to the other. Each time it switches, you will have a chance, via a callback function, to put new data in either of the buffers.

By constantly filling the dormant buffer with the PC game's sound data, you can very easily get the Mac version to sound exactly like the original, with little or no modification to the game's sound code (see figure 11.15). Double buffering is covered in detail in chapter 6.

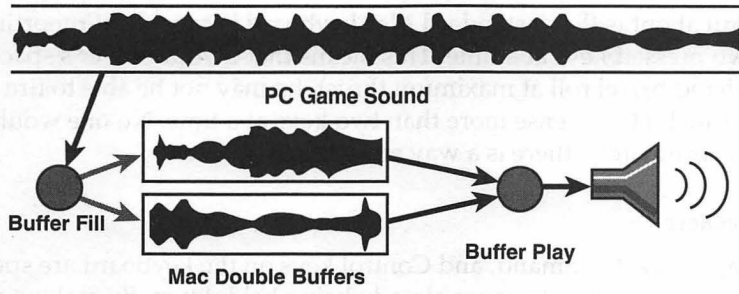


Figure 11.15 *Using a double buffer as a DAC.*

Unfortunately, the use of streamed audio is relatively new to the PC gaming world. Games that do not use streamed audio for their music rely heavily on the sound board's ability to carry a tune. Must be nice....

Music Mixing

On the Mac, with no special music hardware in sight, the CPU must take on the responsibility. This means loading sampled instruments into memory and performing note modulation and mixing "by hand."

Can't be done, right? Wrong. Even the original Mac LC has been known to hold an eight-voice sampled orchestra plus sound effects while playing a medium-speed animated adventure game. Quadras and Power Macs should not have a problem with it.

See chapter 6, "Audio," for details, but recognize that, no matter how you do it, the Mac is going to end up doing much more work than its PC counterpart in order to provide bone-tingling music.

Control Issues

At present, it is safe to say that most PC game players own joysticks and most Mac game players do not. In general, the generic Macintosh keyboard-and-mouse setup is exactly what most users have, and that is what the programmer should write for, if possible.

Using the Keyboard

Until more Mac users adopt joysticks, game makers will need to rely on the only nonmouse input device guaranteed to accompany any Mac: the keyboard. For most action-style games, the keyboard is by far the most popular input device.

One nice thing about keyboards is that, since PCs typically have them too, much of your code and control mechanisms may be the same as in the original version.

One thing to be careful about is that a standard Mac keyboard is capable of reporting a maximum of only two pressed keys at a time. This means that if your player's spaceship is pulling a hard right-hand barrel roll at maximum thrust, he may not be able to fire the lasers, due to the keyboard's inability to sense more than two keys at a time. No one would design a spaceship like that. Fortunately, there is a way around this problem.

Solution: Use the Modifier Keys

The Shift, Option, Caps Lock, Command, and Control keys on the keyboard are special. They get reported even if two other keys are already being held down. By making use of these for vital functions such as lasers, shields, boosters, or windshield wipers, you can add a significant amount of control and playability to your game.

The Keyboard as a Joystick

Many games currently on the PC require a joystick. Right on the box, it says "Requires 386 or better, 2 MB RAM, 150 MB free HD space, Joystick." If a Mac game requires a joystick, there had better be one in the box. Otherwise, it's back to the keyboard and mouse.

For some games, the keyboard is an adequate replacement for the joystick. If fine control is not important to game play, the numeric keypad can provide all necessary functionality.

For games such as flight simulators, however, fine control is important. Given no other option, the user will tend to tap the keys rapidly in order to achieve a similar effect to an actual joystick. Depending on the nature of the game, you might try to provide a variable "pull" based on how long a particular key has been held down. Experiment with various methods and have your playtesters tell you what they like best. For action games, they will probably find that even a fairly crude keyboard control is better than...

Using the Mouse

There are many mouse-based PC games. Adventure games, card games, and storybook CD-ROM games are just a few of the types which can benefit from the point-and-click nature of the mouse.

The Mouse as a Mouse

Obviously, mouse-based games translate nicely over to the overwhelmingly mouse-based Mac.

The only little catch involved in bringing mouse-based games is that most PC mice have two buttons. Games usually use the left mouse button for selecting and the right for performing an action, such as opening a door. This may sound just like the click/double-click distinction on the Mac. It is.

Double-clicking in a game, however, can cause problems. In an environment in which the player might click rapidly or even flail about in panic, unintentional double-clicks can be extremely frustrating. The two best solutions to this problem both involve the keyboard.

If the action is directed at an object on the screen, such as blasting a six-headed monster, you might want to use a modifier key such as Shift or Option to fire the weapon (in this case, a normal click might be used to move your character around the world, so simply clicking on the monster would put you within an easy tentacle's reach).

On the other hand, if the right mouse click is used to perform a general action, such as raising shields or transforming your heroic plumber into a jet plane, it may be better to simply tie the function to a key, so that simply pressing the key performs the action.

Most importantly, do not forget to change the manual. Once you see the words "right mouse click" in the manual, you know it's going to be a cheesy PC port.

The Mouse as a Joystick

There is a reason that real-life fighter planes use joysticks (okay, yokes) instead of mice. Besides the fact that mice lack tactile positional feedback and lightning-quick response time, they just aren't as cool.

There is only one way to make your game play as good as possible when using a mouse instead of a joystick. Sit down, play it, and make subjective modifications until it feels right. Then, give it to the testers and listen well to their feedback. For the initial up-and-running tests, you might want to try one or more of the following:

- Provide some kind of on-screen indication of where the "joystick" thinks it is, so that the user can see a direct "I move the mouse and the little box moves" reaction.
- Try out some nonlinear control effects, giving the player fine control within a "center box" and more dramatic changes further out.
- Make the control self-centering so that, left alone for a few seconds, it will leave the player flying straight ahead.
- If you choose not to have the control self-centering, consider providing a "straight ahead" key that the player can press to quickly find center.
- Make sure you get a chance to play the game using a trackball. Not only are they quite popular on the desktop, but they are all that exists for PowerBook users. (A few laptoppers have even been known to go so far as to stick a suction-cup dart onto the trackball to form a makeshift joystick. Try it out. It works pretty well.)

Joysticks

For PC games, reading the joystick's position is the most basic of functions. Any joystick that the player happens to buy will plug into the game port on the PC, and any game will have instant access to the controller's data.

When you buy a joystick for the Mac, it comes with at least one floppy disk containing the custom control panel and driver for that particular controller. The collection of settings in the control panel will allow the user to make the joystick behave like a mouse, a keyboard, or both. "Behave like a joystick," unfortunately, is not an option.

As joystick adoption increases on the Mac, there will hopefully be a standard game controller protocol, so that games and controllers may work with, not around, each other. Currently there is no such standard, and the Mac game programmer is left with the following two options:

- Obtain a developer's kit for each joystick you plan to support. Write custom software, using each company's distinctly different API, and hope that none of them change after your game hits the shelves.
- Forget about custom software. Start with the assumption that each joystick comes with sufficient software to enable it to control the cursor within the Mac's screen. Then, simply write one joystick routine which reads the cursor position.

While each method has its advantages and neither is an ideal solution, there is one prime benefit to the latter. If you use each joystick's control panel to fine-tune the stick's characteristics and button assignments, your testing crew can help you figure out what works and what doesn't with each stick. Furthermore, if the end user (you remember, the person who

actually buys your product) is dissatisfied with the joystick setup, he or she can easily change it as desired.

One Final Note

As the Intel-based computers move forward into high-resolution displays, windowed environments, generalized digital sound, and consistent user interfaces, they have begun to run into a series of problems which Mac programmers, yourself included, have already solved. If revenge is what you want, then there it is. If you're too nice to gloat and snicker, then help out by offering some of the solutions that have made games possible on the Mac. After all, it will make life easier when they eventually have to port your games over to the PC.

Profile: Robert Barris

Robert Barris started programming computer games at Quicksilver Software in 1986 with a Data East game called Tag Team Wrestling for the Commodore 64. His Macintosh games include MacPlay's Castles: Siege & Conquest and Alone in the Dark, which were both originally programmed for IBM computers. Robert wrested Castles: Siege & Conquest out of its IBM roots so well that it won a place in *MacWorld* Magazine's "Hall of Fame—Best Strategy Game."

Q: What decisions made Castles: Siege & Conquest a good Mac port?

A: Some decisions that helped out a lot were made before we ever thought about a Mac version. For example, the simulation code that deals with positioning castle pieces and projecting from that coordinate system to the screen was fairly well generalized. Thus, it was a straightforward matter to double the displayed resolution of everything without impacting any of the game calculations proper (troop maneuvers, collision detection, and so on).

The fact that the original PC version was 95 percent C code helped, although the usual number of endian/struct packing/character set issues appeared from time to time.

I tried to go "Mac crazy" in areas where I knew it would have the most impact. We pulled a lot of the jelly-bean data files into resources to cut down clutter. We used Halestorm for the music and that came out very good. We used QuickTime movies instead of our proprietary Codec, which was code-named "Little Caesar's"—I'll let the QuickTime nuts figure out where the inspiration for that name came from. And of course, the game needed a total user interface facelift—much more emphasis on Finder-style "select, then operate" and direct manipulation. You can really get your tasks and attacks going in a hurry on the Mac version; it's just so much more streamlined.

continued

The funny thing is, it's not like there's any magic Toolbox call that will make your UI more friendly (although `PopUpMenuSelect()` gets used a bit). We could have done that exact same UI for the original IBM version and it would have been so much better... but hindsight is always handy.

Q: Which part of Castles took the most time?

A: There were three big chunks, as best I can remember.

First, reworking the graphic engine to play nicely inside of a GWorld and still perform well when the user grabs the landscape and scrolls around. There were three or four pages of 68K assembler for the inner loops, and probably forty hours spent with the sampling profiler, learning which Toolbox calls caused the most overhead during blitting to the screen and figuring out how to use the Palette Manager and `CopyBits()` and still go fast.

Second, redoing the artwork. We actually wrote a conversion tool that scaled up the original PC sprites, and we did bilinear interpolation and alpha masking and stuff, but it just didn't come out that good (a painful lesson!). On the other hand, the artist's painstaking redrawing of the characters, based on the machine-converted versions, came out really good. Check out the battle scenes; you can see the little archer dudes stringing their arrows and everything. The PC version has a blinking "brick" where you can point and say "see, that was his arm moving." High-res graphics are a great thing.

Third, the user interface reconstruction. This wasn't very complicated since internally the UI was fairly well separated from the simulation engine, but it was a big task nonetheless.

Q: When porting a game from another platform to the Macintosh, is the best approach to just try to get the game running on the Mac exactly as it did on the other platform, and then tweak it to bring it up to speed on Mac features?

A: It really depends on the code you are working with. I think that given the choice, it's best to not port at all—write it crossplatform in the first place. This becomes easier and easier as time goes by.

If you're confronted with a large body of code whose heritage is unknown, and if the original programmer's not around (as I had to deal with in Alone in the Dark), then it's probably best to port and then tweak as stated.

Q: Castles: Siege & Conquest had a great user interface. Was it all designed in one big design meeting, or was it more of an iterative process designed along the way?

A: Both—the initial plan was pretty complete, but then changes got made along the way.

Q: What are your thoughts on the "look and feel" of Mac games as opposed to IBM games? People seem to hate direct ports on the Mac.

A: If it's an immersion or action title, make it look however you want. But the process of starting, configuring, and stopping the game should follow the usual pattern of any good Mac app.

We play DOOM 2 on the PC a lot; after about the tenth iteration of "How do I make the network activate on this computer?" we started getting pretty fed up with it. If the game is hard to get into, play, or get out of, it's weak in this regard. You have to be able to have a person who has never ever used it before sit down, double-click, and get the game playing with networking, sound, and all your goodies. If there's something missing, the game should speak up and say why.

A strategy game or game that is not "action-based" probably makes more sense having a conventional "WIMP" interface (windows, icons, menus, pointing), and this was the route we took in Castles. I don't think, however, that games need to fit into the typical Mac color scheme (black text on a paper-white background, gray scroll bars, etc.). In Castles, for example, we went with more of a gold text on black background motif that fits better with the character of the artwork and looks somewhat less high-tech. I was surprised how much difference those good Mac fonts make compared with the original 320-resolution PC stuff.

Q: When you port a game, do you have to go over the source code with a fine-tooth comb and understand every single function?

A: Yes and no. On Castles: Siege & Conquest, I was fortunate in that the programmer of the PC version, Byon Garrabrant, is a coworker and a good friend of mine, and so I had already gained a lot of indirect familiarity with how it worked during the development process.

Contrast that with what I did on Alone in the Dark: At the start of the project, I was given the C and assembler source code for the project, including about 50 pages of "machine-translated X86 code" that had been turned into a primitive 68K format. Many, many bugs. Comments and variable names in French!

I will say that in both cases, I set up MPW scripts to let me do rapid searches for keywords throughout the original untouched source as well as the current source. Being able to check out the original untouched source as the project went on was a lifesaver. Being able to do it fast was even more important—even if you're not an MPW programmer, you should have MPW in your bag of tricks; it really does have a great scripting system you can use.

Q: What sorts of optimizations do you find especially useful when writing Macintosh games?

continued

A: Tiptoe around the Toolbox: Use only the subset of calls that you absolutely need to achieve hardware independence and still run fast. It is really important to know which parts of the Toolbox you do *not* need to use.

Also, play games with `CopyBits()` to keep it in the maximum-speed mode. This includes structure packing and alignment stuff. And nowadays, watch out for the dreaded `MixedMode` switches between emulated 68K and PowerPC code.

Q: What advice would you give a beginning Mac game programmer?

A: Buy a Power Mac and CodeWarrior! Get on the Internet ASAP. Get connected to all the Usenet/Web/FTP stuff out there. Also, I think a good starting point is to "animate something smoothly." Trying to animate something will branch you into memory allocation and all sorts of other topics you'll have to learn about.

Q: Any advice for a Mac programmer who has had a good start already and wants to learn specifically about programming games?

A: I assume that means you already have the above items. :)

I have some sayings that I keep close at hand when programming:

"Question your assumptions." The #1 phrase I mutter when debugging.

"Measure first, optimize later."

"Premature Optimization is the root of all evil"—seen on Usenet.

"They're not magicians." If confronted with some dazzling technique that at first seems like magic, I repeat this phrase to myself until I figure out how they did it. On the other hand, if a client asks for something that is clearly not possible, I mutter, "I'm not a magician."

"It's a computer—I can do whatever I want." I often mutter this one when in a discussion about what the Toolbox is or isn't capable of or what Windows can or can't do. It's my way of reminding myself to always be discriminating in my usage of the system code available. Not everything needs be done through QuickDraw; in fact, most games hardly use it at all. I'm also pretty shy of the Dialog Manager and the Control Manager; they don't seem like they were really designed for game usage at all (nor should they have been).

Q: What development tools do you use?

A: Here's what I use:

Art: Director, Photoshop, Premiere, the "Expression" filter, DeBabelizer.

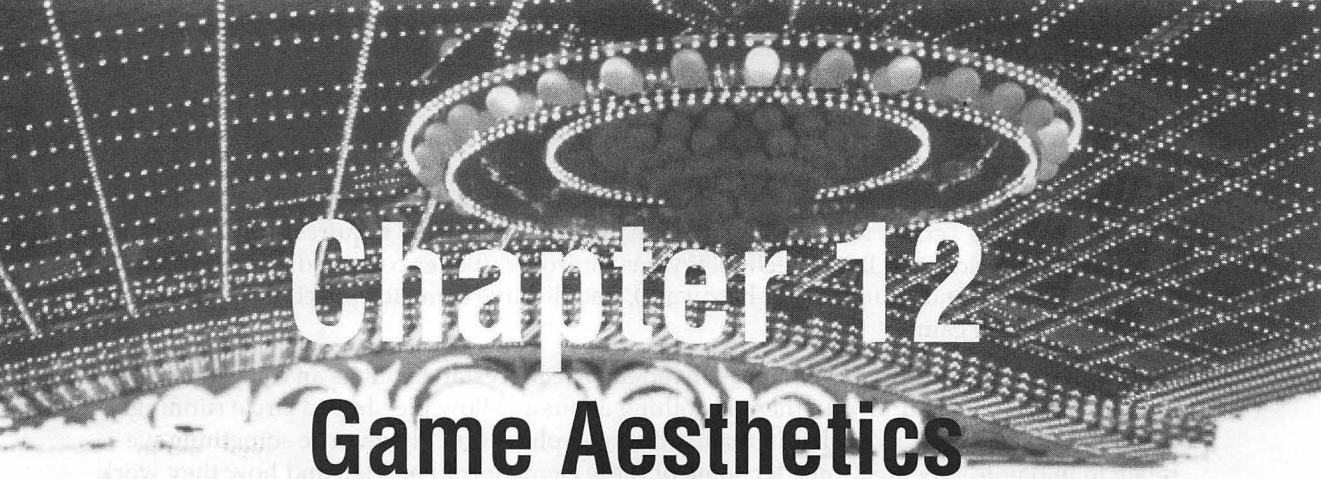
Coding: CodeWarrior, Think C/Symantec C.

Debugging/Tuning: MacsBug, ZoneRanger.

Administration: MPW, FullWrite, ClarisWorks, SoftWindows for PowerPC.

Q: What games do you like to play when not working?

A: DOOM 2 about 70 percent of the time. Still waiting for the Mac version :). In the arcade, I really like the high-powered driving simulators: SEGA Daytona, OutRun, and so on. Also the Namco space games such as Star Blade, Galaxian 3 (both 3-D space shoot-'em-ups). I am a big action fan and I think that will be the next kind of game that I write; I've been coding, reading, and learning 3-D polygon stuff for too long to not make any use of it!



Chapter 12
Game Aesthetics

This chapter is a bit of a departure from the other more technical chapters of this book. It may at first appear to be something of a hodgepodge of ideas jumbled together—even the word "aesthetics" is a little ambiguous. But this chapter focuses on some of the nontechnical—perhaps artistic—aspects of game development.

Some of the ideas in this chapter are things you should be thinking about when you're initially designing the game; other parts are more useful in the middle or toward the end of the game development. Topics include the idea of a game "metaphor," the role of artwork, simple vs. complex games, sexism and racism in games, game violence, and so on. When your head is swimming with algorithms and base 2 numbers, you might find this chapter a welcome departure.

You should probably already be aware that technical prowess alone does not a great game make. It probably isn't possible to describe what makes a game "fun," and this chapter won't try to do that, but there are still a whole range of nontechnical decisions that a game developer must make. You may be assisted to some degree by the following discussions.

If nothing else, you may find issues discussed that you never gave much thought to—perhaps this chapter can help you to take a step back from the source code and see your game from a new, and more holistic, perspective.

Game Metaphor

A game's metaphor is the imagery or story or look or feel that binds all the elements of a game together. It has little or nothing to do with the underlying code. The metaphor is the thread that runs throughout—it is the nebulous "look and feel" of the game.

Milestones in Game Metaphor

Some games (more so in the earlier days of computer games) had little or no metaphor. For example, the original Space Invaders game had an extremely shallow metaphor. Your "ship" in the game was a fairly nondescript cluster of pixels that skirted below quasirectangular "bunkers" firing lines (lasers, I suppose) up at marching grid of "aliens." I'm not knocking

the game for its time, and truth be told, games these days still don't *require* a metaphor to be a success (witness Tetris). Metaphor is a subtle thing that probably requires a certain amount of technological richness. But as technology has improved (better color and screen resolution, richer music and sound, faster hardware), people have come to expect more richness and depth from a game.

If we move forward in time from Space Invaders, Pac Man stands out as an early attempt at more of a metaphor. Mind you, there is nothing about a yellow pie-shaped circle running through a maze that is grounded in reality, but metaphors don't have to be something we relate to in day-to-day life. Consider some of these elements of Pac Man and how they work together: bright colors, cartoon-like characters, simple nursery-rhyme-like music, simple controls. All these elements together begin to suggest a consistent whole. In a "Gestaltian" way, something greater than the sum of its parts begins to emerge. Pac Man suggests a metaphor encompassing merry-go-rounds, amusement parks, childhood, and Saturday morning cartoons. Consider how incongruous it would have been to have a steely laser sound in Pac Man or haunting organ music. The current crop of Mario games still works off of this "childhood" metaphor.

Another milestone in game metaphor was the arcade game Joust. Here was a beautifully rendered game with armored knights on the backs of giant birds battling in a Precambrian world of lava pits and precipices. Mind you, this game had only 16 colors, but the oranges, reds, yellows, and browns chosen all added to the metaphor. The game was fire, magic, Paganism, the Middle Ages.... As in Pac Man, every element of the game worked together to reinforce the metaphor. Imagine now how jarring it would be if Joust had used "cute" music like that in Pac Man.

In any event, let's return to more contemporary titles (in case I'm losing some of the younger audience).... I mentioned Tetris as an example of a game with little or no metaphor. There's no denying its popularity, but how many other games like it can you name that have been as successful? Tetris is a sort of fluke—a game so simple and obvious that it should have been thought of 20 years ago. The number of simple, obvious games that have yet to be written is dwindling.

For a counterexample to Tetris, look at Myst. This game (also extremely popular) is nothing *but* metaphor. There are no brilliant, cutting-edge programming techniques used in this game. Sure, dozens of high-end machines slaved away for years to generate the beautiful artwork in the game, but technically the game itself does nothing innovative. Why, then, is the game so popular? It isn't Doom or Marathon, with cutting-edge number-crunching algorithms screaming away. If you played and enjoyed Myst, ask yourself what it is you liked about it.

The game is beautiful, the graphics haunting. The music itself contributes to the "myst"ery of the world—the slow, monotonous waves crashing on the shore—and you are there all alone—the DaVinci-esque machines and contraptions.... There is nothing (besides the rocket) that looks particularly futuristic (and even the rocket is a throwback to the Victorian or Art Deco concept of a rocket). There's a sense of a parallel world—a path this world might

have taken had it diverged several hundred years ago—perhaps a world that we've all visited in our dreams. And nothing dispels this metaphor; from the moment you descend into the game, the dream-like, Art Deco, alternate world metaphor is consistently reinforced. If there were a tricorder or laser rifle in the game, this delicate metaphor would be shattered.

Discovering a Metaphor

As you might imagine, there isn't any hard and fast method for arriving at a metaphor for any game. Since this is right-brain stuff, there are many different, equally acceptable ways to choose a metaphor. It may well be that the developers of Pac Man and similar games didn't consciously or intentionally think about what sort of metaphor they wanted for their game and then guide the decision-making process in developing it along that path. But decisions are made unconsciously as well, and often, whether it's one person or a few people of like mind, an unspoken metaphor emerges. Nonetheless, you may find it more helpful to be conscious of a metaphor, so you might be more focused in your decisions regarding choice of artwork, colors, sounds, interface, title, and so on.

The metaphor may suggest itself at some point in the development of the game. Glider 4.0, for example, has at its roots a very simple little program I knocked together on a Commodore 64 in the early 1980s (believe it or not). For whatever reason, I wrote a small program that drew a small floor blower (the type you see in central air-conditioned homes) and a small paper airplane. I wrote it such that when the paper glider was anywhere directly over the blower, it would lift; otherwise, it would float to the ground. By itself, it wasn't much of a game at all, but it was sort of compelling and even a little fun to play with—just watching the glider rise on the invisible column of air and float back down.

Still, this humble program was already a seed suggesting not only an entire game, but also the game's metaphor. Moving over the vent and off of it was clearly a rather tame thing—it didn't require keyboard-pounding speed or deftness. Also, it was already an indoor environment, suggesting aspects of a final metaphor. In fact, as the concept of the game started to evolve and grow more complex, the metaphor became clearer and clearer. When the goal of the game became to get out of the house, one aspect of the metaphor became *claustrophobia*. And since flying paper airplanes around indoors was something I might have done as a child to entertain myself on an otherwise boring rainy day, elements of *childhood* and *water* crept into the metaphor as well. Water and boredom suggested a rather drab color scheme with lots of cool colors, pastels and earth tones.

The initial impetus for writing your game, or perhaps the type of game it is, might suggest a metaphor as well. A game like Myst perhaps *is* the metaphor. It's difficult to imagine even beginning a game like Myst without some clear sense of the time and place and the look and feel of the game you're attempting. Any adventure games, in as much as most of them have at their roots J. R. R. Tolkien's *Lord of the Rings*, are going to already have a good sense of a metaphor to follow. A sports game might try to capture the look and feel of a sportscast (and if that's what you were aiming for, it wouldn't be unreasonable to even put commercials into the game—frightening thought though that is).

There is a saying that goes, "A donkey was a horse designed by committee," or, more popularly, "Too many cooks spoiled the broth." This is often extremely relevant to your developing a game metaphor and a rich and consistent look and feel to your game. When you work with a publisher or with a development team, very often many people suggest many ideas for things that will improve the game. Even after a game is complete, users themselves will suggest improvements. But if you already have a well-developed concept or sense of your game's metaphor, some of these ideas may run counter to the "spirit" of the game as you perceive it. In these cases, I would suggest that you stand behind your convictions. Sadly, however, this is probably the hardest aspect of the game to defend. If the publisher wants your game to support the Mac Plus and you're developing a cutting-edge texture-mapped flight simulator, it's fairly easy to explain the technical reasons why that would be impossible. But metaphor is a good deal more difficult to defend with words. If you have a clear "vision" for your game, stick with that vision and don't let your "horse" become a "donkey."

It may be, however, that no metaphor suggests itself before or early on in a game's development. In that case, the metaphor will probably suggest itself when the initial artwork falls into place.

Artwork Aesthetics

There are a lot of subtle effects you can achieve with artwork that can reinforce your metaphor—or even establish it. (They can also work against your metaphor if you're not careful.) I mentioned Myst previously. The type of ray-tracing techniques used to render the art in Myst strive for photo-realism, but the effect in Myst is not ultimately photo-realistic—it's one of otherworldliness. This is very appropriate in Myst; it is probably deliberate and certainly plays into the metaphor of the game.

At least three things contribute to Myst's otherworldliness.

One contribution comes from ray tracing technology's "state of the art." Ray-traced images still lack "something"—the degree to which they fail to perfectly render reality gives them an otherworldy feel. How often have you seen a commercial and recognized that it's ray-traced because you could sense that "something" is lacking? But for Myst, the qualities that ray tracing lacks in reproducing reality add to the "myst"ique and otherworldliness of the game. The shortcomings of ray tracing give Myst a dream-like quality.

The choice of colors, which is more deliberate, also contributes to Myst's otherworldliness—lots of washed-out colors, grays, and earth tones. The popular word to describe these colors is "pastels," but because of some of the connotations of that word, I prefer "low-intensity" colors. When an artist works with an opaque medium (such as oils or acrylics), you get low-intensity colors by adding white to any of the primary or "intense" colors. When using Apple's Color Picker, you select these colors by clicking nearer to the center of the color wheel. Another way you can get this effect in a paint program is by dithering any of the

primary or intense colors with grays. In Myst the effect is to add a look of "faded parchment" to the whole world. Imagine if Myst were filled with bright and primary colors—if it had bright red brick buildings or a bright blue rocket sitting on a silver launching pad. Those intense colors would change the overall feel of the game dramatically.

The third thing that contributes to Myst's otherworldliness look is the mist itself. You might know that there is an option with most rendering packages often called "fog." The idea is that all objects are seen through a haze that gets thicker and more pronounced the more distant an object is—as though the object were seen through a fog or mist. This is a convenient mechanism in Myst because it means you can never see clearly beyond some distance; you have to click and move forward to get closer to those shadows and shapes seen in the distance to see what they are and to interact with them. In addition, this haziness contributes to all the previously mentioned qualities of Myst's metaphor: mystique, otherworldliness, faded parchment, and so on.

I've always deliberately used color in a game to create a mood or establish a metaphor. If you've ever seen the original Glider 4.0, you've seen two deliberate uses of color. One use was to reinforce the game's metaphor. The working metaphor in that game might be described with the following words: autumn; stormy; indoors; claustrophobic; recalling those days as a child when rain kept you indoors and you tried to entertain yourself with paper airplanes, rubber bands, or whatever; exploring inside; evening; *The Wizard of Oz* (I know, that's a long list; but maybe you can see the thread that sort of ties all these elements together into a metaphor). Color, to this end, meant using lots of dark greens, browns, grays, and low-intensity colors (with only 16 colors in this game, I did a lot of dithering with grays to get this subdued effect).

As a more practical use of color (one less related to the metaphor), I made the prizes and foreground objects in the game (such as clocks and batteries) very clear, pure, intense colors. So the alarm clocks (prizes) were bright yellow, but the cracked plaster and exposed bricks in the wall (background) were washed-out beiges and terra-cotta. This helped players distinguish between objects they could interact with and "background clutter" that they could ignore.

Even though Glider PRO was a sequel to Glider 4.0, its metaphor was completely different. This time the metaphor was built up from: spring; sunny; in and out of doors; not claustrophobic; recalling the last day of school when summer began; *Little Nemo in Slumberland*; morning. Quite a reversal. And so, I used brighter colors throughout the game.

Once you've established a sort of metaphor for a game (if you do it early enough in the game development), every decision you make will follow from that. In fact, the whole game development process can simply "flow." What colors should I use? Bright, spring-like ones. What backgrounds? How about a tree house. What kind of music? Happy, upbeat music. Once I had the metaphor fairly coalesced in my mind for Glider PRO, all the little decisions became easy.

Interface Aesthetics

One way to view a game interface is as a continuum in which a bare-bones interface represents one end of the spectrum and Microsoft Word is at the other (just a joke—but you get the idea). You can make a case for any location along the continuum and I have personally leaned different ways at different times. A stripped-down, minimalist interface is elegant and easy to use and figure out. On the other hand, when a game begins to get old, having the ability to modify countless parameters of the game can breathe new life into it. Initially, I leaned toward the latter (opting for a complex game interface), but over the past three or four years, I've come around to appreciating and adopting a much more minimalist interface. I'll explain why by giving you several wrong reasons to make an interface complex.

One common reason (and a poor one) is that programmers don't know what the end user wants in a game. So they throw in everything and the kitchen sink. In Glider 4.0, there were no less than four ways to control the movement of the paper glider in the game. Pressed, I would confess that probably the only *real* way to control the paper glider was with the keyboard. In fact, although I had implemented a mouse and joystick method, it took months before even I could get barely proficient at playing the game using these methods. Furthermore, when visiting MacWorld Expos, I witnessed people new to the game coming up and attempting to fly the paper airplane (unsuccessfully) with the mouse—because it was an option. I learned from this that *more* is not necessarily *better*! Part of your job as a game developer is to make these decisions for the end user *before* he or she has to make them. If only one control method makes sense for your game, then stick with that alone.

There's a practical reason for taking some of the decision-making out of the user's hands (thereby also simplifying your interface). Occasionally, you may get a person asking you about your game, "Why can't you put something in so that I can set the difficulty of the game?" Or, "Why can't I set how many lives I have?" What if you did allow users to set the number of lives they begin with? You know how some games give an extra player every 20,000 points? You could make that a user-defined variable, too. The user could set how easy or difficult the opponents are, how many bonuses are sprinkled throughout the game, and even how many points the bonuses are worth....

See where this is going? There are two major flaws with this reasoning. First, it invalidates any kind of high scores list your game might have; a score of 1,385,290 is meaningless now (you may think that doesn't matter, but look at the number of people who post their high scores on the net or compare scores with one another). But the worst crime is that you have allowed players to make the game so easy that they may be able to "win" the game in a single day and will quickly lose interest in your game. So, make the hard decisions—lay down your game's rules—enforce them.

Another reason a program can get cluttered is that the programmer wants to show off. There's nothing wrong with showing off until it interferes with the elegance of your game or complicates the interface. Case in point: I had just figured out how to do hierarchical menus

when I began Glider 4.0. So, I put a hierarchical menu in it. I've come to loathe hierarchical menus. When you watch beginners (especially children) trying to use those things, you may come to the same realization that I did: They're evil—don't use them. For a word processor they're okay, but a game shouldn't be tedious to use. For every interface decision you make, ask yourself, "Why?" and "Do I need this?" In my case, I should have answered the question, "Do I need hierarchical menus?" with "No."

There are, however, some good reasons for adding to the complexity of the game interface (and I'll explain later how you can isolate this and maintain a simplified and elegant facade). User-definable keys is a good example. Keys that are comfortable for you to use when playing a game are not comfortable for everyone. An obvious example here is any game that requires two hands to play. If you're right-handed, will a left-handed person be comfortable playing your game? So make the keys user-definable. This is a good thing. But there's no reason to make *every* key user-definable. You wouldn't, of course, make the *quit* key user-definable—it should be Command-Q. Only the keys that require dexterity to use during the course of the game need be editable. Therefore, you could hard-code the pause key as well. Again, you might have arrived at this by asking yourself, "Do I need to make all the keys user-definable?" "Is there a good reason to do this?" Question every aspect of your interface. If you go through a questioning process like this, you can always defend every decision you made.

Note

A note about keys: As with other aspects of the user interface, less is better. Often, with some thought, you can completely drop a key or two. For my own tastes, I wouldn't dare go over five required keys to play a game if it can possibly be avoided. For a flight simulator, there may be nothing you can do, but in an arcade game, some clever changes to the game might remedy this. Want to add a "shields" key? A "hyperspace" key? A "smartbomb" key? You can see how easily it gets out of hand (no pun intended). One solution might be to make shields automatic, for example. If the player's ship has shields and is about to get hit, have the shields come on by themselves (of course they only last for so long). For the hyperspace and smartbomb, you could combine them into one key and allow the player to have only one or the other at any given time (this is essentially what I did with "batteries" and "helium" in Glider PRO).

Cleaning Up Your Menus

So you've made the hard decisions but still have a good deal of clutter to try to iron out of your interface. How to proceed? My favorite method is to consolidate the dialogs and menu items. Go through your menus and try to determine how you might reduce them. Figure 12.1 shows examples of menus that might appear in a game.

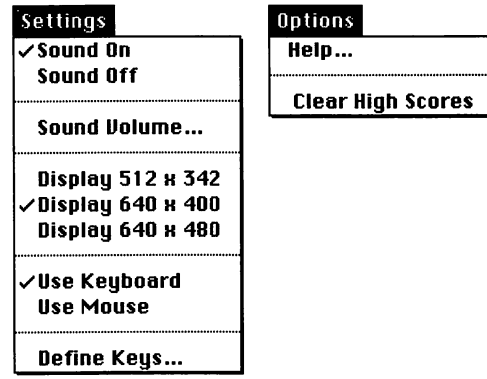


Figure 12.1 *A very cluttered interface.*

Okay, it's a worst case example. But before I start criticizing the easy mistakes, let me point something out. Items directly accessible in the menu bar ought to be items that users will be inclined to change often. If there's something there that users are likely to select only once—the first time they try the game—don't clutter your menu bar with it.

In the above example, Help should always be readily available and never hidden within a dialog box somewhere. But the Clear High Scores item below it is not only something the user won't be selecting often, but perhaps something even dangerous to have so easily accessible.

The menu on the left is the easiest one to simplify. One way we could do it is group all of these settings into a single dialog box and the menu item to access it could simply read Settings or Preferences. If you're worried about a cluttered dialog box, you may want to break it down a bit into three dialog boxes (Sound Settings, Display Settings, and Control Settings). Of the items listed in the left menu, though, a case might be made for making Sound On and Sound Off remain menu items, because depending upon the time of day or night, a player may be accessing these items frequently. Figure 12.2 shows our polished menu bar.

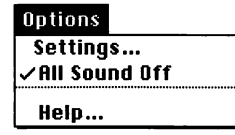


Figure 12.2 *A much neater interface.*

Because we were able to throw out so many menu items, there wasn't a need to have two separate menus. Presumably, Settings would now bring up a dialog box with radio buttons allowing you to select the display size, some method for raising and lowering the volume, a standard button for clearing the high scores, and so on. As you can see, I made the Sound On and Sound Off a single menu item that can be either checked or unchecked.

This should illustrate that an interface need not be intimidating to the user. Often an interface gets to be scary as a result of adding features to a game, bit by bit, throughout the entire development of the game. Finally, though, at least by the time you're polishing up the game, you need to sit back and reflect on your interface and ask yourself questions that might simplify it and make it more elegant.

Mac-Like or Not Mac-Like

If, as a developer, you're a born and bred Mac aficionado who stands up for The Macintosh Way, you may have found yourself in a bit of a quandary when developing a Macintosh game interface. The viziers and emirs at Apple would insist that you stick with the standard menu bar, menu items, dialogs, buttons, and all other components of the Mac standard interface. But when confronting the interface aesthetics for your Mac game, the "standard" nature of this interface can detract from the metaphor of your game—or worse still, can make your game resemble a word processor.

Once again, I propose that there is a continuum of sorts. At one end of this continuum might be SimpleText (or anyway a game that conforms very rigidly with the Human Interface Guidelines as laid down by Apple). At the other end of the continuum might be a coin-op arcade game—say Pac Man or whatever. Where within this continuum do you want to take a stand?

If you want to take a stand at either end of the continuum, the decisions you make as a developer are fairly simple: You'll either be driven by laissez faire or Apple's Human Interface Guidelines. Walking the tightrope between these two, however, is quite a balancing act. As a developer, I tend to prefer this middle ground; I want the user to feel fairly comfortable with my interface and recognize a number of "standard" features, but I don't want the look and feel of my game and my metaphor to be stifled by the conformity that is the "standard" interface. So every deviation becomes an issue. I'll present arguments and counterarguments for most of the decisions you're likely to have to make.

Question number one: Do you use a menu bar or not?

There are a good number of reasons to give the ol' menu bar a toss in a game. Right off the bat, a menu bar is 20 pixels of your screen real estate. If your game supports the little 9" black-and-white monitors on the Mac Plus and SE, the menu bar takes up six percent of the screen. Very often, 20 pixels can make the difference between your game feeling too crowded or spacious. If your game also has a scoreboard along the top showing number of points or whatever, then it looks redundant to have both a menu bar cutting across the top of the screen and a scoreboard right below it. The screen real estate occupied by the menu bar might make a nice place for your scoreboard.

Further, by denying the user access to the menu bar, you avoid a number of potential conflicts with the user changing the sound volume or screen depth from a control panel.

And finally, the menu bar just looks ugly—it looks too word processor-like.

On the other hand, the menu bar is the place a user looks in order to quit your game, read the help for your game, and so on. Having no menu bar can be very unnerving and alienate a user from your game.

My preferred solution is to make the menu bar available when the game is idle or in "splash screen" mode, but to hide the menu bar when a game is actually in progress. This way, when users launch your game, they have all the benefits of the Quit item, the help or preferences dialog for your game, and so on. You'll have to handle the potential dangers, such as the screen depth being switched on you. But having the menu bar hidden during the actual playing of your game buys you that extra 20 pixels to do with as you like; it makes your game look more "game-like." Code for hiding and showing the menu bar is included in the 12 - Game Aesthetics folder on the CD-ROM that accompanies this book.

Hiding the Menu Bar

I have seen people come up with some fairly complex functions for hiding and showing the menu bar. Many of the more complex ones try to handle the general case in which your application might have a window smaller than the screen; thus, when the menu bar is hidden, that portion of desktop revealed under the menu bar needs to be drawn. My own experience with games has been that I always have a main window that completely encompasses the entire monitor; when the menu bar is hidden, I need only be certain my window underneath the menu bar is updated. But anyway, let's move on to the code....

Essentially, there are only two things you want to accomplish when hiding the menu bar. You want to:

1. Tell the Mac that the menu bar height is zero (and save the previous height of the menu bar so that you can restore it later).
2. Make sure your windows `clipRgn` and `visRgn` reflect the new strip of area revealed beneath the menu bar. (Again, I'm assuming you have one main window that extends from the top of the monitor to the bottom—and the entire width as well.)

Setting the menu bar's height to zero is trivial. The Mac maintains a low-memory global that reflects the number of pixels tall the menu bar is. The word "low-memory global" should sound a warning to you, since Apple says in many, many places that you are *not* to change low-memory globals (since they may change in the future and your program would be rendered incompatible or buggy). Fortunately, however, Apple has released new header files (called Apple Universal Headers) that give you a future-compatible way to change some of the more commonly accessed low-memory globals. (Apple's Universal Headers should have shipped with your compiler.) The header that describes these routines is called LowMem.h (for C compilers) and in it you'll find a couple functions called `LMGetMBarHeight()` and `LMSetMBarHeight()`. Using these two routines, you can call `LMGetMBarHeight()` to get the menu bar height and save it off in one of your global variables. Then, you call `LMSetMBarHeight()`, passing it a value of 0 (zero) to flag the menu bar as hidden.

Unfortunately, that isn't all there is to it. In fact, if you were to try the above on its own, you would find that the menu bar is still there—no apparent change has taken place. As far as the Mac is concerned, the menu bar *is* gone, but it is still your responsibility to "cover it up," to erase it.

The problem you encounter at this point is that none of the QuickDraw commands seem to be able to touch the menu bar. You could try to `PaintRect()` over it, `CopyBits()`, or `EraseRect()`, but nothing appears to be able to touch the menu bar. The reason for this is that everything drawn to your window through QuickDraw is routed through your windows `clipRgn` and `visRgn`. Well, there are fortunately a number of region-manipulating functions such as `RectRgn()`, `SectRgn()`, and `OpenRgn()`, so we ought to be able to force the area beneath the menu bar back into our windows `visRgn` and `clipRgn`.

Once we have our windows regions set up properly to reflect no menu bar, any of the usual QuickDraw routines should work fine. Here is the code for hiding the menu bar:

```
short  mBarHigh; // a global variable used in both hide and show routines
void HideMenuBar (void)
{
       // This function hides the menu bar by
       // calling LMSetMBarHeight() and then painting
       // over the menu bar with black.

  Rect      theRect;
  RgnHandle    worldRgn, menuBarRgn;
  GrafPtr   wasPort, tempPort;

  if (LMGetMBarHeight() != 0)   // only hide if not hidden
  {
       // First, we're going to save a pointer to the old
       // port and create a temporary "scratch port" in
       // which to define our regions and paint over the
       // menu bar.

    GetPort(&wasPort);          // remember old port
    tempPort = (GrafPtr)NewPtrClear(sizeof(GrafPort));
    OpenPort(tempPort);         // create temporary new port
    SetPort((GrafPtr)tempPort);

       // We get a copy of the height of the menu bar and
       // save it off in the global mBarHigh (a short).

    mBarHigh = LMGetMBarHeight();

       // Now we tell the Mac the menu bar is gone.

    LMSetMBarHeight(0);         // sets low-mem global to 0

       // Now we need to get a region that describes the
       // familiar rounded rectangle of the main monitor.
       // In cases with multiple monitors, the global
```

```
        // qd.screenBits will be the one with the menu bar...
        // The one whose region we're concerned with
        // imitating.

      theRect = (**GetGrayRgn()).rgnBBox;
      UnionRect(&theRect, &qd.screenBits.bounds, &theRect);
      worldRgn = NewRgn();          // init new region
      OpenRgn();                    // open region for defining
      FrameRoundRect(&theRect, 16, 16);
      CloseRgn(worldRgn);           // just defined the screen

        // Although the region we've just defined represents
        // the entire monitor with the menu bar, we're going
        // to trim this down to a region that represents
        // just the menu bar -- the top strip of the screen.

      theRect = qd.screenBits.bounds;
      theRect.bottom = theRect.top + mBarHigh;
      menuBarRgn = NewRgn();        // define yet another region
      RectRgn(menuBarRgn, &theRect);

        // Now we have worldRgn = the whole screen and
        // menuBarRgn = just the top strip where the menu bar
        // is. By finding their intersection, we manufacture
        // an exact shape of the menu bar -- rounded corners
        // only on the top.

      SectRgn(worldRgn, menuBarRgn, menuBarRgn);
      DisposeRgn(worldRgn);         // no longer need this region

        // Now that we have our menu bar region, we need to
        // add it to our temporary ports' visRgn. In this
        // way we can use QuickDraw to paint over it.

      UnionRgn(tempPort->visRgn,
           menuBarRgn,
           tempPort->visRgn);

      DisposeRgn(menuBarRgn);       // no longer need this region

      PaintRect(&theRect);          // paint over the menu bar

      ClosePort(tempPort);          // clean up
      SetPort((GrafPtr)wasPort);
   }
}
```

Showing the menu bar is, unfortunately, just as complex (only in reverse):

```
void ShowMenuBar (void)
{
```

```
Rect        theRect;
GrafPtr     wasPort, tempPort;
RgnHandle   worldRgn, menuBarRgn;

if (LMGetMBarHeight() == 0)   // only do if menu bar hidden
{
   GetPort(&wasPort);         // do the port thing again
   tempPort = (GrafPtr)NewPtrClear(sizeof(GrafPort));
   OpenPort(tempPort);
   SetPort((GrafPtr)tempPort);

      // Restore the menu bar height in low-mem global

   LMSetMBarHeight(mBarHigh);

      // Create a screen region again (with rounded corners)

   theRect = (**GetGrayRgn()).rgnBBox;
   UnionRect(&theRect, &qd.screenBits.bounds, &theRect);
   worldRgn = NewRgn();
   OpenRgn();
   FrameRoundRect(&theRect, 16, 16);
   CloseRgn(worldRgn);

      // Create a rectangular menu bar region again

   theRect = qd.screenBits.bounds;
   theRect.bottom = theRect.top + mBarHigh;
   menuBarRgn = NewRgn();
   RectRgn(menuBarRgn, &theRect);

      // Intersect the 2 regions to arrive at a rounded
      // menu bar region again.

   SectRgn(worldRgn, menuBarRgn, menuBarRgn);
   DisposeRgn(worldRgn);

      // Once again we add it to our tempPort's visRgn

   UnionRgn(tempPort->visRgn,
      menuBarRgn,
      tempPort->visRgn);

      // But this time we "subtract" the menu bar region
      // from the tempPort's visRgn -- in this way, the
      // menu bar is removed and thus can't be drawn over.

   DiffRgn(tempPort->visRgn, menuBarRgn, tempPort->visRgn);
   DisposeRgn(menuBarRgn);

   ClosePort(tempPort);   // clean up
   SetPort((GrafPtr)wasPort);
```

```
        // At this point, all is well and good, but the Mac
        // hasn't got around to drawing the menu bar. We need
        // to tell it to do so.

      DrawMenuBar();
   }
}
```

The File and Edit Menus

If your game is up and the user has switched out into the Finder and chooses Shut Down, does your game quit? If you handle Apple Events correctly, it should. But it is my understanding that if you don't handle Apple Events, the Finder begins looking for a File menu and then a Quit item in the File menu. If you have renamed your File menu to something else or have no menu items at all (as in the previous discussion about having no menu bar at all), this may fail and an unhappy user will have to "manually" quit your game before the Mac can shut down. Of course, the best solution here is to properly support Apple Events; then you can rename the menus anything you want (realize, though, that Apple Events are a System 7-only thing and your game may still have problems under System 6).

But then there's that pesky Edit menu with its Cut, Copy, and Paste. If your game supports none of these features, my recommendation is that you drop it. Again, it is my understanding that this is a holdover from old System versions in which Desk Accessories (Apple menu items) may have expected the Edit menu item to be there. No one much cares to support it anymore, though, and there's nothing more confusing to a user than a whole menu list in your game that is always grayed out.

To summarize, you should support Apple Events and then, if the name File doesn't apply to your game, rename this menu as Game or Play or whatever is more appropriate. And toss the Edit menu altogether.

Dialog Boxes

It's almost a given that your game will have at least two dialog boxes—perhaps more (the two I'm thinking of are the About box and a Preferences dialog box). But like the menu bar, there is something just *too* familiar about the standard dialog boxes; after all, SimpleText and the Finder both use this same style. But do dialog boxes really detract from your game—specifically, the game's metaphor? They might—a touch, anyway. Can dialog boxes be embellished in such a way as to reinforce your game's theme? Again, probably—a touch.

Before I list things you can do to improve the dialog boxes, we should look at an example of a really ugly one (see figure 12.3).

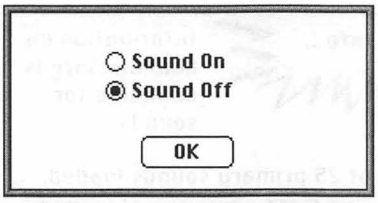


Figure 12.3 *A pretty boring dialog box.*

Beyond just how plain and boring it is (and all the other countless problems with it), this dialog probably inhibits your game metaphor — certainly, in no way could it be reinforcing the metaphor. Suppose your game is futuristic, with a *Blade Runner* look. Where's the grime? This dialog is too clean. Maybe your game is futuristic but with a *Star Trek: The Next Generation* look. So where are the cool bands of color or fancy font or black background? Starting to get the idea?

The simplest way to soup up the look of your game's dialog boxes is to add color. Color icons ('cicn's) or a nonwhite background ('dctb's) might be enough. Perhaps you could use custom buttons ('CDEF's).

You ought to be fairly familiar with ResEdit and recognize the above acronyms as resource types. It takes but seconds to select a custom background color for a dialog in ResEdit, which automatically adds a 'dctb' for you. And then the Mac automatically uses it without any additional coding from you. Experiment with color backgrounds in your dialog boxes.

Remember, though—back to the aesthetics thing—don't add color for color's sake. Going nuts with ResEdit can make your game look more like something ported from the Commodore 64. In most cases, if you add color to dialog boxes, you'll use one color—the same one—throughout all your dialog boxes. Also, the color should still make the text and controls legible (so use a light color probably) and in some way it should relate to or reinforce your game's metaphor.

'CDEF's (control definitions) are another thing altogether. Good 'CDEF's can do wonders for a game's metaphor; imagine buttons that look like steel in a submarine simulator or buttons that look like they're wood-textured in a medieval role-playing game. Writing and maintaining these little tidbits, however, can be a lot of work. If you come across some public domain 'CDEF's that seem reliable and well-crafted, go with those. For my tastes, in general, 'CDEF's are just something else that can go wrong (with System upgrades, etc.).

Color icons ('cicn's) or pictures ('PICT's) sprinkled about the dialog boxes is more my preference. They require very little work to implement (well, 'cicn's are no work at all; it's displaying the proper 'PICT' depending on the monitor depth and updating it in a custom dialog filter that adds the extra work). As I've stated earlier, if you depart too radically from the Mac interface, users can get uncomfortable. But throwing in a few graphics is just subtle enough to protect the users' comfort and still add something to your game's metaphor.

Let's check out a few examples of using subtlety (see figure 12.4).

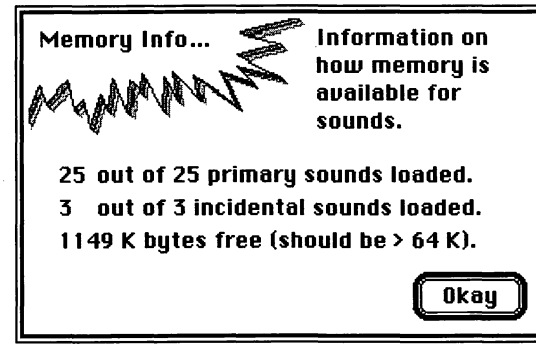


Figure 12.4 *A much more attractive dialog box.*

The metaphor for this game is loosely *American Gladiators* or wrestling. For that reason, the "POW" balloon in the upper left corner became the subtle graphic touch that, to greater or lesser degree, reinforces the game's metaphor.

Figure 12.5 is from the same game and illustrates the internal consistency maintained throughout the game. Once a "look" was decided upon for the dialog boxes, they were all done the same way.

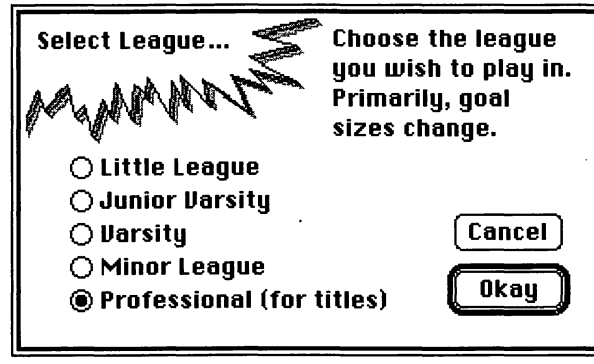


Figure 12.5 *Notice the consistent "look" with the last dialog box.*

The next example shows the "look" chosen for another game (see figure 12.6). Here, the sunset and horizon in the strip across the top tie in directly with the splash screen from the game; this sunset is a consistent theme throughout the game. Notice too that the strip across the top doubles as a sort of title bar for the dialog box. It's important to put titles at the top so that users remember what they selected.

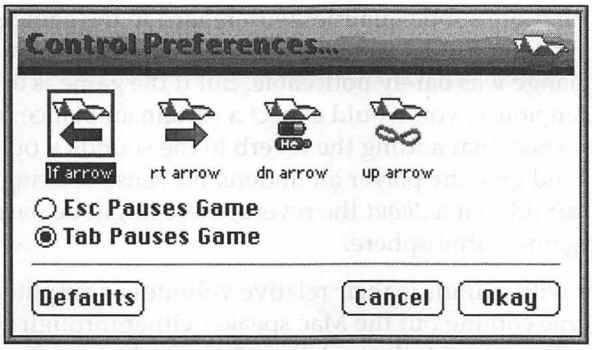


Figure 12.6 *This dialog shows a different "look."*

The moral of the story? Deciding how Mac-like or un-Mac-like to make your game is a delicate balancing act. On the one hand, conforming to the Mac standard can be stifling to a game developer and elicit jeers from the IBM crowd that all Mac games look like spreadsheets. On the other hand, departing too liberally from the Mac look alienates the player and will elicit jeers from the Mac crowd that your game is nothing but a cheap port of an Amiga game. But if you consider these issues, you're likely to find your tastes evolving and your games settling comfortably on a middle ground.

Sound Aesthetics

Like the artwork in a game, sounds have an aesthetic quality as well. Unlike the interface, though, less is not always better—but more on that later. Keeping with the idea of a game metaphor, sounds should contribute to the look and feel of the game. I'll use Glider again as my example, and show how decisions on sounds were made based on how they contribute to the overall metaphor of the game. You can apply the lines of reasoning I follow to your own projects.

The game Glider has a very familiar setting: rooms of a house, tables, basketballs, leaking ceilings, and so on. For this reason, I chose to keep the sounds familiar as well (when possible). For the sound of the leaking ceiling, I sampled the actual sound of water being dripped into a glass. But there are also some rather whimsical elements in Glider; for example, a paper airplane isn't going to *really* get extra power from colliding with a battery. For these whimsical events I chose to use whimsical sounds.

In the broadest sense, you can think of sounds as falling into two camps: whimsical (or goofy) sounds and realistic sounds. If a game is very abstract, such as Pac Man, you would want to lean toward the goofy sounds. But in a game like Myst, this would be blasphemous; realistic sounds are much more appropriate. Consider your game's intent when selecting sounds.

Your sounds can have even more subtle qualities with regard to the game's metaphor. After sampling all the sounds in Glider PRO, I went back through many of them and applied a reverb effect. Often, the change was barely noticeable. But if the game is to take place primarily in a large wooden house, you would expect a certain ambiance (reverb if you like) from the sounds. I didn't expect that adding the reverb to the sounds would make them suddenly sound *more real* and give the player an undeniable sense of being in a large wooden house, but I was afraid that *without* the reverb, there might be *something missing* that detracts a touch from the game's atmosphere.

Another thing to consider with sounds is their relative volumes or amplitude. Although a player can adjust the volume coming out the Mac speaker either through your game's preferences or through the Sound control panel, the relative volumes of one sound versus another are predetermined by you, the developer, when you sample the sounds.

One thing you could do is to let someone else play your game and you just simply walk around the room and listen. If one sound suddenly screams out too loudly, or one sound is almost lost altogether, you may want to go in with a sound editing program and increase or decrease the volumes of some of the individual sounds. There was a game recently released (I won't mention any names, but it was a sequel to an early popular game in which you try to collect all the crystals) in which there is one extremely loud and obnoxious sound that plays when you get on the high scores. That one sound is easily twice the volume of any other sound in the game.

However, you can use the relative volumes of sounds for effect. For example, if you have a minor sound (such as if you have a walking character in the game and there is a footstep sound), you may consider this an incidental sound and purposely drop its volume. Make other sounds, such as the sound of getting an extra player ("DING DING DING!") or an explosion, loud for effect. Along with all the other hats you wear when developing a game, you are also sound engineer in charge of mixing.

A final note about the quantity (not quality) of sounds. Like colors, the more varied your sounds are in a game, the richer your game is—sounds are the "audio palette" of a game. It's true that there are memory constraints and you have to be frugal with your sounds—but only up to a point. I prefer more, shorter sounds than fewer, long sounds. If there is a sound when you get bonus A, make it different from the sound of getting bonus B. If your primary weapon in a game is a laser and you're pounding on it throughout the entirety of a game, make three similar but slightly varied laser sounds and play the different sounds at random.

Very often sounds are overlooked by the game developer and assigned as merely an afterthought. But if you have ever played a good game with the sounds turned off, you know how much depth effective sounds can add to a game.

Packaging

I won't claim to be an expert on packaging; I've stayed away from the marketing end of game sales to a large degree (I did one box cover and had two proposals rejected, so...), so I'll be brief. It's likely that as developers, you won't be very involved in the packaging end of your game. Still, if your input is solicited, there are a few lessons I have learned and can pass along.

Always, always, always have screen shots on the back of the box. If your game doesn't call the Event Manager, these may be difficult to obtain. The simplest solution is to write a temporary function in your game that simply calls `GetNextEvent()`. This will enable Command-Shift-3 to function so you can make screen shots. Another thing you might consider is a special function in your game that also slows it down to a crawl. For a fast arcade game, you often miss the "great shot" by the time you go to hit the screen dump keys. So something like "slow if Caps Lock is down" can be extremely useful. A final solution is to "doctor up" a screen shot or set one up in the game. Since you have the source graphics of your game, you can use a paint program to paste together a pseudo-scene from the game (sure, I've done it).

But absolutely make sure you get screen shots on the box—presumably on the back of the box. I also feel you should have a screen shot in any advertisement as well. When a person walking through a store sees your game on the shelf, she will first be drawn by the box cover. But invariably she will flip the box over to look for screen shots (we all know that the cover of the game box doesn't represent what the actual game will look like). If there are no screen shots, I think the normal reaction is to be distrustful of the product—to assume that the game looks really poor.

The front of the box, as I reason it, is there to get the customer's attention. It should, obviously, reflect the premise of the game and hopefully coincide with the metaphor you've constructed for your game. Keep in mind, however, that the artists that render your box covers usually spend only a brief time with your game. It may be useful if you provide a "write up" that describes not only the premise of your game but also the metaphor you are trying to elicit.

As I said, the front of the box is important because it serves to get customers' attention so they pick up the box. A person's natural reaction, having just picked up your game, is to turn it over and scan the back of the box. Here then, on the back, you include the screen shots, a description of the game, hardware and system requirements, etc. I point this out because it reemphasizes the role of the front of the box; it need not describe the game in detail, it need not contain screen shots showing the customer how the game "actually" looks—it simply needs to get the person to pick up the box. Put the details on the back.

A final note on packaging: Even though software is an information good that you're selling (merely 1's and 0's, albeit laid out with so much care), there is often still a need to establish a tangible "value" to your product. If there is no perceived benefit to buying the game rather than pirating it, this increases the likelihood that your game will be pirated. For this reason, you see a WWII Navy Pilot's Manual included with the game Hellcats and star charts in Space Rogue. Note too that these items can play right up to your game's metaphor (too often though, they are a means to providing "look up" copy protection as well—which I abhor, but that's another topic).

Of course, all of this may be out of your control as a developer. The publisher will make the packaging decision. You have more interesting issues to attend to.

Sexism, Racism, and Violence

Who is your game's audience? You should ask yourself this question at some point during your game's evolution. A simple answer to this is "Myself." There's a lot to be said for writing a game that appeals to you. In fact, I can't imagine how you would write a game that was unappealing to you. Very likely, you're not going to enjoy your own game much when you finish it because you'll simply be sick of having to play it as you developed it (one of the tragedies of game development). But if the initial idea for the game was one you might have enjoyed, you'll be duly inspired throughout and have a clear sense of a direction for the game.

But this is where we get to the touchy subjects of sexism, racism, and violence.

I doubt many game developers consider themselves blatantly sexist or racist, but making the main character in the game a white male is likely going to shut out the largest segment of the population. It's not that a black female couldn't enjoy a game in which the protagonist is a white male, but she'll likely have more difficulty identifying with the main character of the game if she sees a white male forearm (the player's) holding up a pistol in the lower center of the screen. As a player you might get used to it, but if you are a white male, try to imagine how odd it would be if you saw a black woman's forearm (and this is supposed to be your arm) holding the pistol. LucasArts, in many of its games, has done an admirable job by allowing the player to select his or her gender—and then by reflecting this gender in the artwork throughout the game.

But the majority of the computer game audience *is* white male, you say. That's true, but it could be the chicken/egg thing. Perhaps the lack of identifying with any aspect of a computer game has alienated women from playing computer games; thus, it is assumed that males make up 90 percent of the game audience. Who knows?

Still, if it takes little effort on your part to give the player a choice, why not do so? Games such as Myst, SimCity, and Glider (and of course Tetris) don't have an obvious race or gender bias (the other humans in Myst are, of course, white males, but who's to say what

your race and gender are in the game?). I've been to half a dozen MacWorld Expos at which I was able to see, firsthand, people playing and enjoying my games (such as Glider). And I'm especially proud of the numbers of women, adults, and minorities I've seen enjoying Glider.

Glider is also a nonviolent game. Does that make it inferior? If that is an aspect of the game that contributes to the diversity of people who enjoy the game, then I'm glad for that too.

It's not selling out or succumbing to political correctness to take the path of nonviolence or gender/race neutrality in a game. If it broadens your game's appeal, and if it broadens your audience, then shouldn't you be even happier that even more people are enjoying your game?

However, that said, you may have a very specific game that requires violence or that is historically from the viewpoint of a white male —for example, a WWII flight simulator. No one would suggest you rewrite history and load up the P-51 Mustangs with blanks and pilot them with Native American women. But there are many games in which violence is more of a gimmick or a crutch or in which the unintentional racism or sexism is just a result of thinking narrowly.

There ought to be another term for "narrow thinking." A game in which the player is assumed to be a white male isn't really sexist or racist—not overtly. It's more sexist and racist by oversight or by "exclusion." I know, this sounds like political correctness. Just ask yourself, though, if you want your game to appeal to everyone; if so, if you were the "othersome," is there anything in your game that might act as a barrier or impediment to your enjoyment of the game?

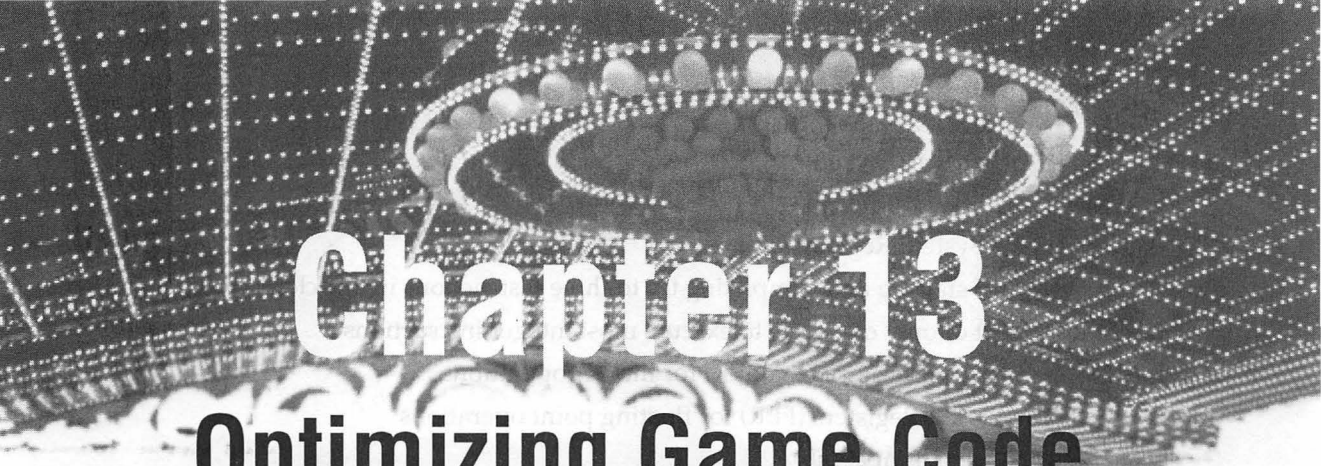
Finally, let's look at violence. Does violence add anything to your game? Is your game bereft of excitement without violence? Or rather, is there another mechanism you could use in place of violence to achieve the same goals? Say your game is one in which a person walks around killing people. It could be perceived as a touch less violent if you replaced the people as targets with something like robots. Have you ever heard anyone say, "This world isn't violent enough—we need more violence"? Sure, we played violent games as kids, we saw violent movies and cartoons, and we turned out okay—what harm is there in a little game, after all. The whole violence thing may be much ado about nothing. But still, why contribute to that culture if you can avoid it—if you pick up a newspaper or turn on the news, you'll be reminded that we didn't *all* turn out okay. Just in case these things do contribute in a small way to the violence in our culture, see if you can tone it down a bit or throw it out altogether. It's a safe bet that no one ever lost sleep for writing a nonviolent game.

If you would rather discuss the financial or "practical" matters, though, consider the violence in the games I've mentioned. Apart from Doom and Marathon, the best-selling games for computers are decidedly tame: SimCity, Myst, Tetris, and so on. Perhaps it's just chance that they are nonviolent and popular. Perhaps it's the nonviolence that keeps doors open and allows for more universal appeal; maybe *that* is what makes them so popular. Consider too that now there is a rating system that video games have come under; this came about in Congress because of society's concern about the growing violence in games. And if anything,

it appears that our society is moving ever further in a direction away from violence—perhaps by banning violent games altogether in the not-too-distant future. In Germany, violent computer games are illegal. Don't care about your game selling in Germany? What if a dozen or so more countries followed suit?

Sure, in some cases it seems that violence sells games—witness Doom and Street Fighter II. However, these games have only a narrow market—even though they might saturate that big market. Adolescent males make up a big market. If you've got *the* violent game for this crowd, you're a millionaire. But I would argue that this market has room enough only for one title at a time—and then only briefly. So if you're not *the* game for this crowd, you're nothing, and you have forfeited any appeal you might have had for the broader market. I believe that the broader market (everyone else) seems to have the capacity to embrace several titles at once. This makes it less of an all-or-nothing market. There's room for several big hits at once, the semihits, and even "cult" or "niche" games.

Still (as I step off my soap box), you ought to at least consider these issues. If you've thought through these issues and still want to write a violent game, at least it has been a thought-out and measured decision—not one made without consideration.



Chapter 13

Optimizing Game Code for PowerPC

Many of you may have already tried porting your applications to native PowerPC code. Some of you have noticed a significant improvement in performance, while others have found only minimal speed gains. A straight port of a 68K-based application to a PowerPC application will generally result in an overall performance increase of 50-200 percent, but I'm here to tell you that with careful attention to coding, you can actually achieve increases of 300-400 percent. I'm not talking about comparing a 68K application running on a Mac LC to a Power Mac 8100/110; rather, I'm talking about comparing high-end Quadra 800 performance to a lowly Power Mac 6100/60's performance.

General Macintosh applications can be very difficult and tedious to optimize for PowerPC execution, but game applications are different. All game applications have one common bottleneck: the blitter code. In almost every game, the most time-consuming part of the application is the blitter—the code that draws sprites and dumps video buffers to the screen. As such, this is where we will focus most of our attention.

The PowerPC is a RISC-based processor, and I have a saying about any RISC chip: KNOW YOUR PROCESSOR. The old 68K processors were just like any other CISC-based processor—they executed instructions in a linear and predictable manner. Each instruction required a certain number of bus cycles, and that was that. RISC is a whole different world, and no two RISC processors work the same. So, to get to know our processor, I'd like to introduce the features of the PowerPC 601 and 603.

PowerPC 601 and 603 Features

- Capability of executing up to five instructions simultaneously with the five independent execution units:
  - BPU—Branch Processing Unit
  - IU—Integer Unit
  - FPU—Floating Point Unit

  - LSU—Load/Store Unit
  - SRU—System Register Unit
- Capability of starting and completing up to three instructions in one clock cycle
- Requirement of only one cycle to execute most integer instructions
- 32 General Purpose Registers (GPR) for integer operations
- 32 Floating Point Registers (FPR) for floating point operations
- Zero-cycle branch capability
- Six-entry instruction queue

Believe it or not, we are going to make use of every one of these features to optimize our PowerPC-native games. For those of you looking ahead to the PowerPC 604 processor, here are some additional specifications.

PowerPC 604 Features

- Simultaneous starting of up to four instructions
- Simultaneous finishing of execution of as many as six instructions
- Capability of executing up to six instructions simultaneously with the six independent execution units: FPU, BPU, LSU, and three IUs

Optimizing the Blitter

Even though I am mainly going to focus on optimizing blitter routines, all of the optimization techniques you are about to learn can be easily applied to almost every area of your application. Once you see how much performance is gained by these optimizations, you'll be hacking though your code looking for things to fix!

The Old Way and the New Way

On the 68K-based Macintoshes, the fastest way to blit a sprite to the screen was with a series of `move.l` instructions which would copy 4 bytes (or four 8-bit pixels) per instruction. The fastest way to do this was with code which looked like this:

```
     jmp    @unRolledLoop(index)

@unRolledLoop
     move.l source+,dest+
     move.l source+,dest+
     move.l source+,dest+
     move.l source+,dest+
     move.l source+,dest+
```

```
move.l  source+,dest+
move.l  source+,dest+
move.l  source+,dest+
...
...
move.l  source+,dest+
```

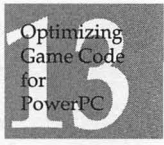
Jumping into an unrolled loop of `move.l` instructions with autoincrementing was always the fastest way to copy data on a 68K. On the PowerPC, we are not limited to copying only 4 bytes/pixels of data per instruction—we can copy 8! The FPU registers on the PowerPC are 64 bits wide—that's 8 bytes! By defining our source and destination pointers as `double *`, we can write C code which will automatically be about two times faster than its 68K assembly language counterpart.

```
/************** QUICK COPY ***************/
//
// This will quickly copy data from sourcePtr to destPtr
//

void QuickCopy(Ptr sourcePtr, Ptr destPtr, long numBytes)
{
double  *src, *dest;              // pointers to doubles
long i,numDoubles;

   numDoubles = numBytes / 8;     // calc # doubles
   for (i=0; i < numDoubles; i++)
      *dest++ = *src++;           // copy 8 bytes per loop
}
```

Copying with `doubles` is the heart of optimization for all PowerPC-native games, because using `doubles` yields the single greatest speed improvement. The only problem with using `doubles` is that your source data must be some multiple of 8 bytes in length. With the old `move.l` method, the data had to be some multiple of 4 bytes in length, so you may have to modify your data files to work.



Let's Make It Faster

Though the above code is far from complete, it is dramatically more efficient than any method possible on a 68K machine. We can demonstrate by writing a simple screen dump routine that is based on an old 68K-based blitter routine having no PowerPC optimizations written into it. We then write a "poor man's" profiler to test how long it takes to execute the routine. Then we start making optimizations to the routine to see how much it can be speeded up. The "poor man's" profiler will simply be a routine that calls `DumpBuffer()` 1,000 times with a `SysBeep()` at the beginning and at the end. You'll just need to time it on your watch. It's not the most accurate profiler in the world, but it works. The routine looks like this:

```
/***************** PROFILE IT ***************/
//
// Allocates memory for two pixel buffers which are both 640 x 480 pixels.
// Then it calls DumpBuffer(), which blits buffer 1 into buffer 2.
//

void ProfileIt(void)
{
long i;
Ptr  buffer1,buffer2;

     /* ALLOC MEMORY FOR 2 BUFFERS */

  buffer1 = NewPtr(640*480);     // buffers are 640 x 480
  buffer2 = NewPtr(640*480);

     /* SEE HOW LONG IT TAKES */

  SysBeep(0);                    // beep at beginning
  for (i=0; i < 1000; i++)
    BlitBufferToScreen(buffer1);
  SysBeep(0);                    // beep when done
}
```

The unoptimized blitter routine is as follows:

```
/*************** BLIT BUFFER TO SCREEN ***************/
//
// This is a completely unoptimized blitter routine which
// will blit the 640 x 480 buffer to the screen.
//
// On a Power Mac 6100/60, it takes approximately 23 seconds
// to execute this 1,000 times.
//

static void BlitBufferToScreen(void)
{
long x,y;
long *destPtr,*srcPtr;
Ptr  destStartPtr;

  srcPtr = (long *)gDrawBufferPtr;      // get ptr to start of buffer
  destStartPtr = gScreenAddr;           // get ptr to start of scan line

  for(y=0; y < 480; y++)
  {
    destPtr = (long *)destStartPtr;     // set destPtr to start of line

    for (x=0; x < (640/4); x++)         // copy a line with 4 byte longs
      *destPtr++ = *srcPtr++;
```

```
    destStartPtr += gScreenRowBytes;    // skip to next scan line
  }
}
```

As mentioned in the code's comments, this version of the blitter routine takes 23 seconds to execute 1,000 times on a Power Mac 6100/60. The routine has not been optimized for the PowerPC, since it is simply copying `longs` from the `srcPtr` to `destPtr`.

Now notice what happens when we modify the code to use `doubles` for copying the pixels:

```
/*************** BLIT BUFFER TO SCREEN ****************/
//
// This is a partially optimized blitter routine which
// will blit the 640 x 480 buffer to the screen.
//
// This has been optimized by using doubles to copy 8 bytes
// of data at a time.
//
// On a Power Mac 6100/60, it takes approximately 15 seconds
// to execute this 1,000 times.
//

static void BlitBufferToScreen(void)
{
long x,y;
double *destPtr,*srcPtr;
Ptr  destStartPtr;

  srcPtr = (double *)gDrawBufferPtr;        // get ptr to start of buffer
  destStartPtr = gScreenAddr;               // get ptr to start of scan line

  for(y=0; y < 480; y++)
  {
    destPtr = (double *)destStartPtr;       // set destPtr to start of line

    for (x=0; x < (640/8); x++)             // copy a line with 8 byte doubles
      *destPtr++ = *srcPtr++;

    destStartPtr += gScreenRowBytes;        // skip to next scan line
  }
}
```


This blitter routine executes 1,000 times in only 15 seconds! Technically, the pixels are copied twice as fast as the previous version, but because of the general overhead and bus limitations, our overall speed increase is less than double. The next few optimizations work on reducing this overhead. First, let's take a look at the PowerPC code that CodeWarrior generates for the above routine. Select Disassemble from CodeWarrior's Project menu (see figure 13.1).

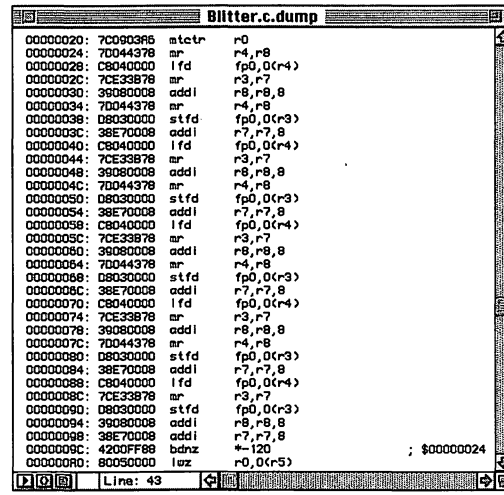


Figure 13.1 *Disassembly of the blitter routine.*

Even though the inner loop of this routine appears extremely tight in C, CodeWarrior generates 31 lines of PowerPC code for each iteration of the loop with full optimization enabled. Don't panic quite yet—it's not as bad as it looks. If you read the PowerPC code, you'll notice that CodeWarrior's optimizers have partially unrolled the loop, making each iteration appear larger than it is. In actuality, only six lines of code are generated for each copy:

```
lfd  fp0,0(r4)      // load 8 bytes into fp0 from r4
mr r3,r7            // move r7 into r3
addi r8,r8,8        // increment r8 by 8
mr r4,r8            // move r8 into r4
stfd fp0,0(r3)      // store the 8 bytes from fp0 to r3
addi r7,r7,8        // increment r7 by 8
```

Even though six lines are better than 31, we can do better, and this example doesn't even include the code needed to do the looping. I'm fairly certain that everyone reading this has been told by some non-game programmer that assembly language is obsolete on RISC-based machines and that C compilers can do a much better job of optimizing code than can be done by hand in assembly. Well, as an experienced CISC and RISC game programmer, I'm telling you that statement is not true. A good programmer can almost always do a better job of optimizing code than any compiler. The question is how much time the programmer can afford to spend manually optimizing code in assembly. The benefit of a compiler over an assembler is that a compiler will optimize code everywhere without the programmer lifting a finger. The average programmer would not be able to spend time writing assembly

language RISC code for every routine in a game, only the crucial ones. Luckily for us, sprite drawing and screen updating are the main speed concerns in most games, so we can afford to spend the time necessary to force our compiler to generate the code we want.

Using Indexes

The first thing to know about PowerPC machine code is that there is no such thing as autoincrementing. On the 68K, it was very efficient to use `*destPtr++ = *srcPtr++`, since the code generated for that line almost always looks like this:

```
move.l  a0+,a1+
```

The data was copied and the addresses were incremented in one line. The PowerPC does not work like that at all. Not only is there no autoincrementing, but you cannot copy directly from one address to another. Instead, you must do the following:

1. Load data from memory to a register.
2. Copy data from the register to memory.
3. Increment the source pointer.
4. Increment the destination pointer.

The best way to cut down these steps from four to two is to do two things. First, we want to completely unroll the inner loop of our blitter routine. Second, we want to do all of our loads and stores as indexed arrays rather than autoincrementing pointers. Once we do this, our new routine is as follows:

```
/*************** BLIT BUFFER TO SCREEN ****************/
//
// This is a partially optimized blitter routine which
// will blit the 640 x 480 buffer to the screen.
//
// This has been optimized by using doubles to copy 8 bytes
// of data at a time.
//
// On a Power Mac 6100/60, it takes approximately 14 seconds
// to execute this 1,000 times.
//

static void BlitBufferToScreen(void)
{
long y,rowBytes;
double  *destPtr,*srcPtr;

   srcPtr = (double *)gDrawBufferPtr;    // get ptr to start of buffer
   destPtr = (double *)gScreenAddr;      // get ptr to start of scan line
```



```
rowBytes = gScreenRowBytes/8;              // calc rowBytes in terms of doubles

for(y=480; y ; y—)
{
   destPtr[0] = srcPtr[0];
   destPtr[1] = srcPtr[1];
   destPtr[2] = srcPtr[2];
   destPtr[3] = srcPtr[3];
   destPtr[4] = srcPtr[4];
   destPtr[5] = srcPtr[5];
   destPtr[6] = srcPtr[6];
   destPtr[7] = srcPtr[7];
   destPtr[8] = srcPtr[8];
   destPtr[9] = srcPtr[9];

   destPtr[10] = srcPtr[10];
   destPtr[11] = srcPtr[11];
   destPtr[12] = srcPtr[12];
   destPtr[13] = srcPtr[13];
   destPtr[14] = srcPtr[14];
   destPtr[15] = srcPtr[15];
   destPtr[16] = srcPtr[16];
   destPtr[17] = srcPtr[17];
   destPtr[18] = srcPtr[18];
   destPtr[19] = srcPtr[19];

   destPtr[20] = srcPtr[20];
   destPtr[21] = srcPtr[21];
   destPtr[22] = srcPtr[22];
   destPtr[23] = srcPtr[23];
   destPtr[24] = srcPtr[24];
   destPtr[25] = srcPtr[25];
   destPtr[26] = srcPtr[26];
   destPtr[27] = srcPtr[27];
   destPtr[28] = srcPtr[28];
   destPtr[29] = srcPtr[29];

   destPtr[30] = srcPtr[30];
   destPtr[31] = srcPtr[31];
   destPtr[32] = srcPtr[32];
   destPtr[33] = srcPtr[33];
   destPtr[34] = srcPtr[34];
   destPtr[35] = srcPtr[35];
   destPtr[36] = srcPtr[36];
   destPtr[37] = srcPtr[37];
   destPtr[38] = srcPtr[38];
   destPtr[39] = srcPtr[39];
```

```
        destPtr[40] = srcPtr[40];
        destPtr[41] = srcPtr[41];
        destPtr[42] = srcPtr[42];
        destPtr[43] = srcPtr[43];
        destPtr[44] = srcPtr[44];
        destPtr[45] = srcPtr[45];
        destPtr[46] = srcPtr[46];
        destPtr[47] = srcPtr[47];
        destPtr[48] = srcPtr[48];
        destPtr[49] = srcPtr[49];

        destPtr[50] = srcPtr[50];
        destPtr[51] = srcPtr[51];
        destPtr[52] = srcPtr[52];
        destPtr[53] = srcPtr[53];
        destPtr[54] = srcPtr[54];
        destPtr[55] = srcPtr[55];
        destPtr[56] = srcPtr[56];
        destPtr[57] = srcPtr[57];
        destPtr[58] = srcPtr[58];
        destPtr[59] = srcPtr[59];

        destPtr[60] = srcPtr[60];
        destPtr[61] = srcPtr[61];
        destPtr[62] = srcPtr[62];
        destPtr[63] = srcPtr[63];
        destPtr[64] = srcPtr[64];
        destPtr[65] = srcPtr[65];
        destPtr[66] = srcPtr[66];
        destPtr[67] = srcPtr[67];
        destPtr[68] = srcPtr[68];
        destPtr[69] = srcPtr[69];

        destPtr[70] = srcPtr[70];
        destPtr[71] = srcPtr[71];
        destPtr[72] = srcPtr[72];
        destPtr[73] = srcPtr[73];
        destPtr[74] = srcPtr[74];
        destPtr[75] = srcPtr[75];
        destPtr[76] = srcPtr[76];
        destPtr[77] = srcPtr[77];
        destPtr[78] = srcPtr[78];
        destPtr[79] = srcPtr[79];

        destPtr += rowBytes;                // skip to next scan line
        srcPtr += 80;
    }
}
```

Our tiny little blitter routine has gone from a few dozen lines to around 100. Take a look at the PowerPC code generated by CodeWarrior now (see figure 13.2).

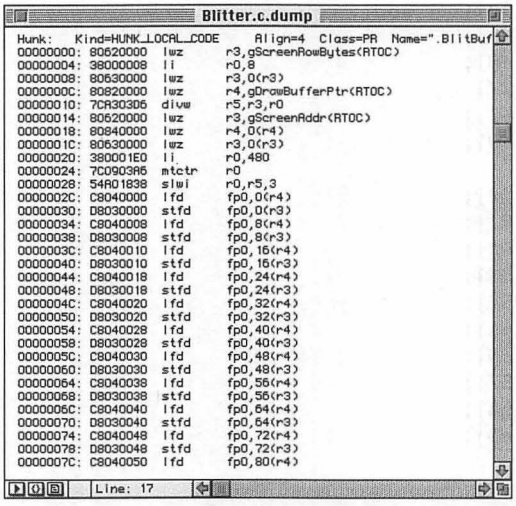


Figure 13.2 *Disassembly of the blitter routine.*

Note that only two lines of code are generated for each copy:

```
lfd  fp0,0,(r4)      // load fp0 with data @ r4+0
stfd fp0,0(r3)       // store fp0 to r3+0
```

Looks much better than the old six-line version, doesn't it? Don't get too happy, because the punch line is that our routine which took 15 seconds now takes... drum roll please... 14.5 seconds. "What!?" you may be asking. "How could it have cut off only a half a second after we cut the code from six lines to two?" The answer is simple: parallel execution.

Remember that the low-end PowerPCs have five execution units that can work simultaneously and complete up to three instructions per cycle. The six-line version of our blitter routine uses three execution units simultaneously:

```
lfd  fp0,0(r4)       // uses the LSU (3 cycles)
mr r3,r7             // uses the SRU ( 1 cycle)
addi r8,r8,8         // uses the IU (1 cycle)
mr r4,r8             // uses the SRU (1 cycle)
stfd fp0,0(r3)       // uses the LSU (3 cycles)
addi r7,r7,8         // uses the IU (1 cycle)
```

The first three lines of code execute simultaneously in three cycles and the next three lines of code execute simultaneously in three cycles for a total of six cycles. Note that the `mr` and `addi`

instructions actually complete *before* the `ldf` instruction, since they execute in only one cycle each. Now look at the new code again:

```
lfd  fp0,0,(r4)     // uses the LSU (3 cycles)
stfd fp0,0(r3)      // uses the LSU (3 cycles)
```

Since only one execution unit is being used (the LSU), it takes three cycles to execute each line. Therefore, this version also takes six cycles to complete a single copy. The end result is that it takes just as long to copy a `double` now as it did before; the only savings we get result from completely unrolling the loop, but since CodeWarrior had already partially unrolled the loop for us in the previous listing, our savings are minimal. But minimal or not, any savings in such a critical section of code is a good thing. If you make enough of these kinds of tiny optimizations throughout your code, they will all add up and your game will run much faster.

Even though in this example we did not achieve great performance gains by unrolling the loop and using indexed arrays, other parts of your application may benefit more from this kind of operation, especially if the operations are more than just copying data from one place to another. If your routine is loading data and then modifying it before saving it out, you may find that you can obtain a more noticeable performance gain. For example, the following code, which draws part of a sprite with a mask,...

```
for (i=0; i < 640; i++)
  *destPtr++ = (*srcPtr++) ¦ (*destPtr & mask);
```

...would most likely generate faster code if written as an unrolled loop of...

```
destPtr[0] = srcPtr[0] ¦ (destPtr[0] & mask);
destPtr[1] = srcPtr[1] ¦ (destPtr[1] & mask);
destPtr[2] = srcPtr[2] ¦ (destPtr[2] & mask);
...
destPtr[79] = srcPtr[79] ¦ (destPtr[79] & mask);
```


Another thing to note here is that for an operation involving logical operators such as AND and OR, we cannot use `doubles` because there is no way to perform these logical operations on floating point registers. Blitter code that uses logical operations must be written using longs.

Alignment Issues

Here is the one big bummer about the PowerPC: Misaligned accesses can cost you more cycles than you ever imagined. A misaligned access occurs when you load a short, long, or `double` at an address that is not aligned to its respective data type. In other words, a short needs to be aligned to an address that is a multiple of two, a long to an address that is a

multiple of four, and a `double` to an address that is a multiple of eight. Reading or writing to an address that does not follow these rules is a misaligned access.

The PowerPC 601 has hardware support for all misaligned accesses; therefore, the performance hit is very minimal. In most cases, this will cost you only one cycle. On a PowerPC 603 or 604, however, there is hardware support only for misaligned shorts and longs. A misaligned `double` causes an interrupt and the misalignment is corrected in software, not hardware! What this means is that writing a misaligned `double` on a 603 or 604 can cost you up to 300 cycles! Usually it's less than that (around 40 cycles), but the interrupt may cause your instruction cache and data cache to get blown. Your simple data move went from a few cycles to up to 300 just because it was misaligned by maybe 1 byte.

While writing the game Power Pete, I ran into this problem. Because of the way the scrolling engine works in that game, screen blitters are misaligned almost 90 percent of the time. The game, which runs full screen at 31 frames/second on a Power Mac with a 60 MHz PowerPC 601, ran at about 15 frames/second on a Power Mac equipped with a 120 MHz PowerPC 604!

What I had to do to get around this problem was to write two blitter routines, one for PowerPC 601 chips and one for the 603 and 604. The version for the 603 and 604 blitted using longs, because on the 603 and 604, long-misaligned accesses are handled in hardware, so the cycle penalty is negligible. This new version of the blitter was only half as efficient as its 601 counterpart using `doubles`, but it was the only good solution to the problem. Luckily, the PowerPC 603 and 604 chips are plenty fast enough to make up for the difference in efficiencies of the blitter routines.

The average game does not have the kind of massive misalignment problem that I had with Power Pete, but it is extremely important to realize that misaligned accesses can utterly bring your application to a screeching halt. Be aware of where you read and write your data!

Optimizing with Floating Point Math

We've done about all that we can for the blitter routine, but another important area of optimization deals with floating point math. Prior to the 68K, programmers were always afraid to use floating point because either it was too slow or the user's computer didn't have an FPU. The PowerPC is about as floating point-friendly as you could ever hope for. First of all, *every* PowerPC has an FPU, so you can count on every Power Mac owner being able to run your game. Secondly, the FPU on the Power Mac is extremely fast. Pipelining allows up to three instructions to be executing in the FPU concurrently, and most floating point instructions execute in only three cycles.

In the old days, programmers were forced to use fixed point math instead of floating point. As you may already know, fixed point math is based on the principle that a set number of bits in a value is defined as the fractional part and another set for the integer part. For example, in one possible fixed point model, the value 0x1000 might actually represent 1.0 if

the lower 12 bits were the fractional part of the number. The common notation for describing the format of a fixed point number looks like this:

(# of sign bits, # of integer bits, # of fractional bits)

Therefore, the representation (1, 19, 12) means a 32-bit fixed point number with 1 bit for the sign bit, 19 bits of integer, and 12 bits of fraction.

The main problem with fixed point math is the overhead. When you multiply two fixed point numbers together, you have to shift the resulting value back down to keep the decimal point in the correct bit position. For example, if we have two (0, 24, 8) values and we multiply them...

0x50 * 0x30 (5.0 x 3.0)

...we get...

0xF00 (240.0)

This is clearly wrong. We need to shift the result right by 8 bits to correct the value:

0xF00 >> 8 = 0xf0 (15.0)

Put bluntly, don't ever, ever do this on the PowerPC—it's a waste of cycles! Floating point on the PowerPC is much faster and much more accurate than fixed point. The speed savings don't come just from the FPU itself, but from the simple fact that while the FPU is processing its three instructions, the other four execution units can be off doing their own thing. While three floating point instructions are being worked on, you may have executed several load, branch, add, and register instructions. In some respects, you are getting the floating point calculations for free because you are still able to perform other tasks while the FPU is at work.

In addition, by not using fixed point math, you free up some general purpose registers (GPRs) that can be more efficiently used for other things in your code. Remember that the FPU uses the 32 floating point registers (FPRs) while the IU uses the 32 GPR registers. By using floating point math, you give yourself a total of 64 registers to use, but by limiting yourself to fixed point math, you have only 32 registers to use.

It should be noted that the blitter routine we worked on uses only floating point registers, but it does not actually use the FPU. The Load/Store Unit (LSU) does all of the loading and storing to and from the floating point registers, and the LSU generally takes two to three cycles to execute one instruction.

One thing to be careful of while working in floating point: Divides are very slow. Whereas a multiply may take only three cycles to execute, a divide can take 18 to 32 cycles! Whenever possible, it's better to multiply by a fraction than to divide. For example, the expression

```
newNum = number / 100;
```

could be better written as

```
newNum = number * 0.01;
```

The multiply version executes up to ten times faster than the divide version, but the results are identical. Divides are the only floating point instructions to be wary of. All of the other important floating point instructions execute in three cycles.

Optimization Potpourri

Knowing exactly how your RISC processor functions is essential to optimizing RISC-based applications. The single best reference for PowerPC optimizations is the *PowerPC 604 RISC Microprocessor User's Manual* from Motorola. To fully understand the PowerPC, this book is a must. There are, however, many little optimization techniques for RISC processors that do not demand an in-depth comprehension of PowerPC specifics.

`static` Declarations

The first such optimization has already been used in the sample code. It is good practice to define all file-local functions and file-local global variables as `static`. Defining functions and global variables as `static` limits their scope to the current file. The benefit of doing this is that it may eliminate the need for certain jump vectors and dispatch tables which are generated by the linker. It's a minor optimization, but one that should be used wherever possible. In many ways, this optimization applies more to 68K applications than PowerPC applications, but nonetheless, it will possibly cut a cycle here and there.

Avoid Local Structs and Arrays

In C, local structs and arrays are always assigned to memory rather than to registers. For example, if you define a `Rect` structure in a function...

```
void MyRoutine(void)
{
Rect myRect;

   myRect.top = 100;
   ...
   ...
   ...
}
```

...storage for that struct is allocated on the stack and your routine will constantly access RAM whenever the `Rect` is used. Remember that accessing RAM is a great way to waste two to three cycles, but accessing register variables is usually one cycle (or free if your pipelining is good).

You should avoid using local structs as much as possible. Obviously, many Macintosh Toolbox calls take structs as parameters; therefore, you have to use them some of the time, but you should avoid using structs at all other times. Instead of having a `Rect` structure, you

should define separate variables for `top`, `bottom`, `left`, and `right`. The same goes for members of an array. It's bad to define small local arrays such as `indexList[5]`. Rather, you should manually define five separate variables: `index1`, `index2`, `index3`, `index4`, and `index5`. True, your code is going to look a bit uglier with everything all separated out, but your routines can become no-memory-access routines, which can execute many times faster than a routine whose local variables are stored in RAM. Not having to access RAM all the time can also improve data cache performance.

One final note on this topic: There are only 32 general purpose registers. To avoid using them all up, you should try to define variables within the scope of their use. For example, if you have a routine with three embedded loops, define each loop variable within the previous loop:

```
long i;

   for (i=0; i < 100; i++)
   {
      long j;
      for (j=0; j < 100; j++)
      {
         long k;
         for (k=0; k < 100; k++)
         {

         }
      }
   }
```

This way, the register used by variable `k` can be reused later in the function. A good compiler attempts to do this for you, but to be safe, you should do it by hand anyway. And once again, using floating point variables increases the number of integer registers available; thus, when it seems appropriate, use floating point.

Limit Function Parameters

On the old 68K machines, function parameters were passed on the stack. On the PowerPC, because of the huge number of registers, function parameters are passed in registers. This improves function-calling performance tremendously, because no memory accesses are required to pass parameters. There is, however, a limit to the number of parameters that can be passed this way. When that limit is exceeded, all remaining parameters are pushed onto the stack, which can cause a substantial performance hit. CodeWarrior appears to allow eight parameters passed in registers (which is quite a lot of parameters). If you have a function that requires more than eight parameters, you should find another way to call the function. That ninth parameter will cause a substantial speed hit, especially if the function is called often.

Consistent Data Sizes

The next little optimization technique is very simple: Be consistent with data sizes. Try not to mix variable sizes in your code, because that causes the compiler to generate additional masking or memory-clearing operations to match the variables. The best way to avoid size mixing is to always use longs whenever a short or char is not absolutely needed. There is no speed penalty for using a long variable over a short or char; the PowerPC processes both operations equally as fast.

For example, the following technique is bad:

```
long a,b,c;
short  d,e;
char f;

   a = (b*e)+(c/f) - d;
```

On the other hand, this is good:

```
long a,b,c,d,e,f;

   a = (b*e)+(c/f) - d;
```

My advice is that you try to always use longs, but there are times when memory is more important than speed. Large data structures may require that you use chars or shorts in order to save memory, but remember that mixing data types can cause a performance hit. Be careful when deciding whether you want to save a few bytes of memory or a few cycles of processing time. For example, suppose you have a massive table like this:

```
   char gMyList[6000];
```

This array uses only 6K of memory because each element is 1 byte. Now let's suppose we need to use the elements in this code to perform some large calculation:

```
void MyRoutine(long someNumber)
{
long largeResult;

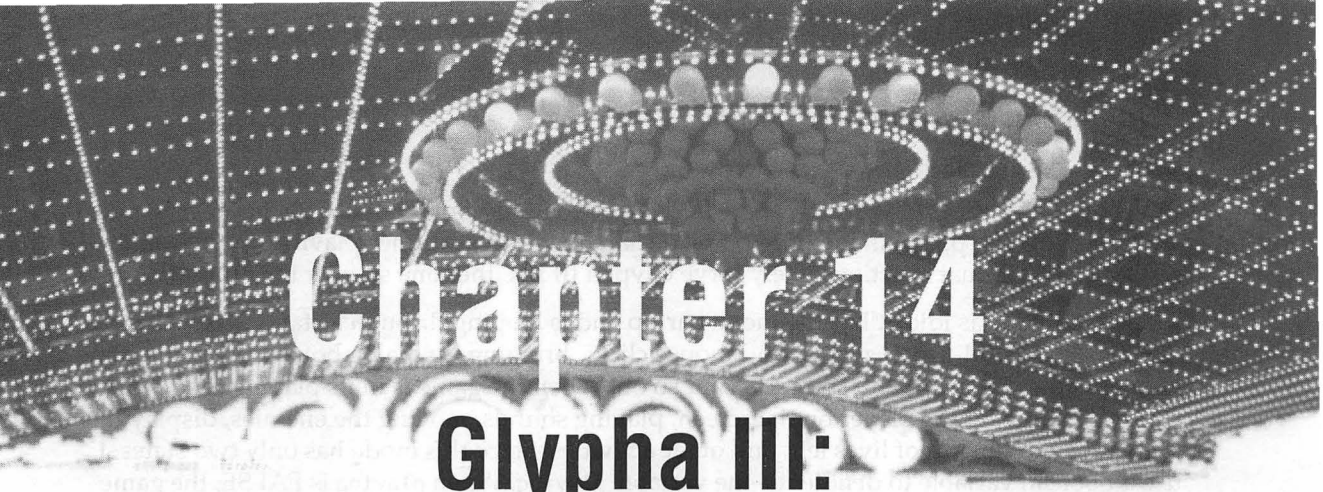
   largeResult = someNumber * gMyList[50];
}
```

In this example, we are multiplying a long by a char, which is not the fastest way to do this, but we saved a large amount of memory by keeping `gMyList` as an array of chars and not an array of longs. If our array was an array of longs, then we would be multiplying a long by a

long, which is the fastest way to perform this calculation. In cases like this, you have to ask yourself if it's worth saving 24K of RAM at the cost of a few cycles, or whether you really need those critical cycles and 24K is a negligible cost.

The PowerPC family of processors are phenomenal RISC chips. You should gather as much technical information as you can find on each chip's specifications, because as I've said before, "KNOW YOUR PROCESSOR!" Knowing the intricate details about each chip will help you get out of little problems like the one I had with Power Pete, and it will help you gain an intuitive feel for ways to optimize your games.

Always remember that a cycle saved in one place is a cycle you can blow off somewhere else when you get too lazy to do any more optimizations! That's Law of Programming #153.



Chapter 14

Glypha III: A Case Study

On the CD-ROM that accompanies this book, you'll find the arcade game Glypha III and the complete source code for the game. The source code is in very straightforward C and extensively commented. Although Metrowerks CodeWarrior was the compiler used to compile Glypha III, there's no reason you couldn't compile the game (with minor changes) with Symantec THINK C or Apple's MPW.

You are encouraged, of course, to examine the source code of Glypha III—nonetheless, sometimes it's difficult to get a sense for the overall structure of a program by simply looking at the many, many lines of code. In this chapter, I'll begin by explaining how Glypha III was set up and how it works within the very broadest of scopes. I'll then start narrowing my scope and explain in increasing detail what the various parts do.

I hope this in-depth look at a real game will help you see how the parts fit together. Because this game was written while the rest of this book was under development, you have a great advantage over me—you can optimize your own game with the information you found in the other chapters! I can't tell you how much I wish this book had been available when I started writing games.

It would probably benefit you to have played the game Glypha III a few times (or at least familiarized yourself with its interface) before reading what follows.

Top Down

When you begin a game (or any programming project for that matter), it's extremely beneficial if you understand how your program is going to fit together before you ever begin. Very often I look at how the final game can be broken down in terms of "modes." Modes are well-suited to computers; we have all sorts of programming conventions set up to handle modes—switch (case) statements, conditional (if/while) statements, and so on. So considering the broadest scope at the top-most level, is there a compelling way to organize your game? Any way to see it in one of a number of "modes" or "states"?

In the broadest sense, Glypha III can be in one of two modes: playing or idle. In other words, we can divide the program into two parts, declare two states—either playing or idle—and say furthermore that functions likely serve Glypha III in either one state or the other.

When Glypha III is idle, it has the menu bar up and is looping through a standard event-driven loop, handling keystrokes and mouse clicks, bringing up dialog boxes, displaying the help screen, handling menu selection, and so on. When the game is in its playing mode (or state), it is moving the player on the screen, playing sounds, moving the enemies, displaying the score and number of lives left, and other activities. Since this mode has only two states, I use a Boolean variable to denote it—the variable `playing`. When `playing` is FALSE, the game is in idle mode; when it's TRUE, Glypha III is in play mode.

Initialization

When Glypha III is launched for the first time, a number of variables must be initialized, then the game falls into its idle mode, waiting for the user to select New Game to switch over to playing mode.

First and foremost, every Macintosh program must initialize the Toolbox. I have a fairly standard-looking function that handles this. Perhaps the only thing noteworthy is my initialization of `randSeed`. Because most games rely heavily on random numbers, it is important that you randomize `randSeed` so that your game doesn't behave in a repeatable manner (perhaps you recall that the arcade game Pac Man had "patterns" you could follow and the game would then behave in a deterministic, consistent, nonrandom manner—I don't care for that).

Initialization should also set up any off-screen pixmaps you're going to need in the game. It's slow and scrambles memory to create and destroy these structures before and after every game. Since we can reuse them throughout a game and from game to game, we should create them only once—when the game launches. Many of the off-screen pixmaps will be used to hold the source graphics the game uses for the various enemy sphinxes, the player, eggs, and so on. These graphics are read in from 'PICT' resources and then drawn to the off-screen pixmaps. Reading in large 'PICT' resources can be slow, and if you tried to do this before every game, you'd find the delay unacceptable.

Along similar lines, it's best to preload all sounds you're going to need as well. Sounds are read in from 'snd ' resources. Anything you can read in from the hard drive and store in memory will improve your game's performance. There's nothing worse than having to hit the hard drive to grab a graphic or a sound in the middle of a game.

I should point out, though, that sometimes there's no way around doing just that—hitting the hard drive in the middle of a game. If your game has dozens of large background graphics (but only one is needed at any one time), you may find that the memory requirements of your game (should it preload every single graphic) would exceed 8 MB or whatever.

Incidentally, you should have a target platform in mind when you're writing a game (a sort of minimum configuration you're assuming the user has). I write commercial games and therefore try to encompass as many machines as I can. Nonetheless, it is 1995 as I write this, and in the back of my mind I'm more or less assuming a 4 MB machine with at least a 13" 256-color display and System 7. I usually add additional code to allow for a slightly "inferior" setup (for example, I might make my drawing routines capable of handling 4-bit color or, if the machine doesn't have memory for all the sounds, I may load in only those that I can). Remember, however, that the target platform is a moving target. By 1997, I think it would be entirely reasonable to have a PowerPC machine as your target machine.

Idle Mode

There really isn't much to say about the idle mode. If you're familiar at all with the standard Mac event loop, I deviate very little from that in Glypha III. The menu bar is there; I have Command-key shortcuts that work in the normal fashion; there are real dialog boxes that you can bring up—pretty much Mac 101 material. If, however, even these more mundane topics are new to you, you might study the Glypha III code and consult a general beginner's Mac programming book such as Hayden's *Programming Starter Kit for Macintosh* (ISBN: 1-56830-174-X).

Let's talk about a few minor deviations from the norm. Just for a diversion, I take content clicks (mouse clicks in the main window's content area) and strike lightning at the cursor. When I was writing the lightning function, I wanted it to be able to draw two jaggy bolts of lightning from the tips of the obelisks to any arbitrary point on the screen. Needless to say, a good way to test that my function worked was to take mouse clicks in the window and try to strike them with lightning. When I was satisfied that the function worked correctly, I decided to leave the test code in. Like I said, it's a little diversion.

Another slight deviation I introduce to the standard event loop is the handling of a few specific non-Command-key strokes. Okay, let me explain. Typically, unless you have a word processor program or such, the keyboard is treated as little more than a "menu shortcut" device. Generally, when you detect a `keyDown` event, you test to see if the Command key also is down, and if it's not, do nothing (if the Command key is down, you pass the event to a menu-handling routine). I like to look for a handful of "normal" keys such as the Help key on an extended keyboard. You'll notice as well that when the help screen is up, I handle the Up/Down cursor keys and Page Up/Page Down keys.

Play Mode

As you might imagine, the real action occurs when the game is in play mode. At this point, it's helpful to further break the action of the game down into submodes of sorts—or at least into smaller "steps" or "units."

First, however, let me explain the concept of "frames." This isn't rocket science, but I should clarify first that an arcade game in action is really a series of individual frames run together one after another—just like a movie or cartoon animation. The idea is, you look to see what keys are down (or whatever your input device is), move the player, move the enemies, fire off a few sounds, draw the whole "scene" on screen, and then do the whole thing over again—as fast as is possible.

So the steps or units I referred to are things like I just mentioned: get user input, move player, move enemies, and so on. In Glypha III, the steps are roughly as follows: move the player, move the enemies, deal with the mummy hand in the lava, deal with the floating eye, draw the whole scene, handle any lightning, and then get the user's input. Sound is handled throughout these steps on a case-by-case basis.

Scaled Coordinates

To get fluidity of motion, I invented a sort of arbitrary coordinate system in which the player and enemies move that is not the same as the screen's coordinate system. In fact, it differs by a multiple of 16. If the player moves 32 "units" to the right, on the screen the player appears to move only two pixels. Further, if the player moves only one unit, the player probably doesn't appear to move at all on the screen because of round-off.

Why do that? Why complicate things that way? Well, it does indeed complicate things, and sometimes simpler is better. Sometimes I'll use my scaled system and sometimes I'll use the pixel scale system—it just depends on what I'm doing. In a moment I'll explain the real advantage (in the case of Glypha III) of using a scaled coordinate system, but first we need a quick background in physics (this may be rocket science).

Accept that a single frame in Glypha III takes some fixed amount of time. In that frame (that amount of time), the player may move some distance (scaled or otherwise). What determines how far the player moves? The player's velocity does. In fact, if you use numbers within an acceptable range, you can calculate the player's new location by simply adding the player's velocity to its old position (this is what I do in Glypha III). And what determines the player's velocity? Well, when the player flaps its wings, some amount of impulse is applied to the player's velocity in the upward direction (sorry if this is sounding rocket-science-ish—it's really simple, but in explaining it I'm trying not to offend the physics majors out there). In other words, the player is accelerated upward—its velocity upward is increased. If we are adding the player's velocity to its location to determine its new location, you would expect that as the player flaps and increases its upward velocity, the player would begin moving faster and faster upward.

Well, a couple of things work against the player's flapping. The primary factor is gravity. Regardless of whether the player flaps or not, gravity is always there, working to pull the player downward (or decrease its upward velocity). So we have a dynamic setup now in which the player's flapping accelerates the player upward, while gravity is accelerating the player downward.

If you think about it for a second, it should be obvious to you that the player's "flap impulse" needs to be greater than the pull of gravity—or the player could never get off the ground. Even if they were equal, the player would need to flap every single frame in order to only break even with gravity.

Well, given the 640 x 480 pixel nature I'm assuming for the screen, we find that for 30 frames per second (what Glypha III tries to maintain), even a gravitational attraction as small as one pixel per frame makes the player appear to be on Jupiter. The player will fall like a rock! So what we really want is a gravitational pull smaller than one, so that the descent from the top to the bottom of the screen is more fluid. Using integers, however, we don't get smaller than one without going on to zero (and a zero gravity is, of course, no gravity at all).

The solution is to stick with a gravity of one or so but scale up everything else. In other words, a velocity of 64 doesn't move the player 64 pixels on the screen, but (in the case of Glypha III) moves the player only four pixels.

Another nice thing about this solution is that you can get velocities of less than one pixel per frame for the player. How does that look to the person playing the game? Well, pixels are very rigid things—you can't move something 1/2 a pixel. Nonetheless, if you moved an object one pixel every second frame, that would certainly appear to be equivalent to a speed of 1/2 pixel per frame. Using a scaled coordinate system, when you go to draw an object to the screen, you are now forced to use the pixel coordinate system and have to scale down. This scaling down does the rounding off, but in as much as the original scaled values are unchanged, even if they increase by only a small amount each frame, eventually they will round off to a different value and the object will appear to move.

So, the player's velocities, the player's position, the flap impulse, gravity—all these things are scaled up by 16. The bounding rectangle of the player (the pixel position of the player) is calculated by dividing the scaled position of the player by 16.

Player Modes

The concept of modes extends to the player, the sphinx enemies, and the hand and the eye as well. The player can either be:

- Idle (`kIdle`)—A new player has just appeared on a platform. While idle, a player cannot be killed (and is not even sought) by the enemies. A player has only a limited time it is allowed to be idle. Hitting a key (to flap, for example) drops the player immediately out of this mode.
- Walking (`kWalking`)—The player is determined to be standing on a platform. A player drops into this mode when getting within a certain region above a platform—at which point a graphic showing the player with legs extended (landing gear lowered) is used. Left/right movement is also very different when a player is in walking mode. A player drops out of this mode if it is killed, if it walks off the edge of a platform, or if it flies off it.

- Flying (`kFlying`)—This, of course, is an airborne player. There are some "submodes" within this mode, such as whether or not the player is being held by the giant mummy hand.
- Killed and Falling (`kFalling`)—The player has been struck dead by an enemy and is falling downward. The player's bones fall and bounce until they come to rest on a platform or slip beneath the lava.
- Pile of Bones (`kDeadAndGone`)—This is the player who was just Killed and Falling and has now come to rest as a pile of bones on a platform. This mode is timed (just to give the player a few seconds to take stock of his plight). When the timer expires, if the player has any extra lives, a new player is born and we begin back in Idle mode.

I didn't go into all the intricacies of each of the above modes. I list these in order to explain why I use modes at all.

Consider that, within the Glypha III code, there is a function that detects collisions between the player and the enemy. If you look at the above modes, you can reasonably argue that really only two of those modes need even bother calling the collision routine—Walking and Flying. When the player is dead or idle, it can be ignored. By breaking the player handling down into smaller categories depending on the player's mode, we get more organized code, quicker game playing (because we don't needlessly call unrelated functions), and no hassle with all those conditional exceptions.

Rendering

I've adopted a rather elegant method for handling screen drawing for Glypha III. It has its drawbacks, but the advantages seem to outweigh them.

Here is how the off-screens are handled:

From the player's point of view, there is a lavish background (the stone walls and platforms of the temple) and small sphinxes and such moving across this background.

From a programmer's standpoint, the sphinxes and such must be moved every frame and their old locations erased. Moving the graphics is easy—you just redraw the sphinx in a new location. Erasing the old graphic of the sphinx, however, is a bit trickier.

Were the background a solid color—like, say, a solid black—then erasing the old graphic would be much simpler. We could merely paint black over the old sphinx before we draw the new sphinx. Besides the fact that this solution doesn't work for a complex background, doing it in the order I suggested—erase old graphic first, then draw new graphic—has the added problem of introducing flicker (the interval between when the graphic is erased and redrawn will be detectable)—trust me.

So, we don't want flicker, and we have a complex background. How to cope? The trick is to do everything off-screen. In other words, do your erasing and redrawing off-screen. When

everything is as it should be, copy it all to the main screen. Sure, there will be "flicker" off-screen, but no one sees what goes on off-screen. The only disadvantage to this technique is the added overhead, both in memory requirements (for the off-screens) and in time (having to do the additional copy to the main screen).

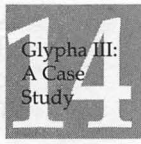
The way the off-screens are set up is like this: One 640 x 480 off-screen holds the entire background graphic. This off-screen is merely a source off-screen, used for "erasing" sphinxes and such by copying the original background right over them. For this to be viable throughout a game, the background off-screen must never be marred—it must always represent a perfect copy of the screen sans enemies, etc. When the background changes, however (from level to level with the addition and removal of platforms), this background off-screen must be updated to reflect the new background on screen.

A second 640 x 480 screen is also employed as a "scratch screen" on which the next frame of the game can be prepared. It is to here that the flicker is moved. It is on this off-screen that old sphinxes are erased and new sphinxes are drawn. It is from the graphics on this screen that the main window is refreshed.

Managing all these off-screens and swatches of graphics and their locations can be tedious. This is where the system used in Glypha III shines. As far as rendering goes, all that's important are the bounding rectangles for the various copy operations. Which rectangle are we to copy from and which rectangle are we copying to? All of the enemies have data structures that maintain their bounding rectangles on screen. There are, however, sphinxes, eggs, the player, torch flames, a giant eye, a hand, and so on. What would be more elegant is if, rather than all these different types of data structures, we had a simple array of rectangles that we could just pass to the rendering routine and say, "Copy these rectangles from screen A to B."

For simplicity, I used an array of rectangles of a fixed size in Glypha III (the array size is fixed—not the size of the rectangles). When the various animated elements of the game (such as the sphinx) are handled, there is a simple routine they can call, passing a rectangle, that will store the rectangle in the next empty slot in the array for later rendering. When all the enemies and the player have been taken care of, the rendering function can just walk down the array of rectangles copying from one off-screen to another.

An added benefit is that the rendering is all done in a simple loop. I chose to use the Toolbox routine `CopyBits()` in this loop because Glypha III doesn't require more horsepower than that, but you could replace it with your own code to draw direct to the screen and get a significant boost in speed. If the copying routines were scattered throughout the code, you would have a much more difficult job trying to replace it all.



As a final note, erasing the old graphics (sphinxes and such) requires that you remember where the old objects were. Since we built up this array of rectangles from the last frame, it seems sensible to introduce some way to hang on to this list from one frame to the next. You don't care about where the enemies were two frames or ten frames ago, but where they were just one frame ago is needed for erasing. Glypha III actually maintains two separate arrays of

rectangles. Every other frame, we switch between which one gets the new rectangle data and which one acts as storage for the last frame's rectangles. Take a look at the Glypha III source code in this chapter's folder on the CD-ROM to see exactly how it's done.

Speed Governing

The final comment I have about the code deals with speed issues. As I mentioned, Glypha III is not a real horsepower-dependent game. The graphics moving about are relatively small; I've set a cap so that there are never too many enemies on the screen at one time; the background doesn't scroll; and so on. So, there really isn't a problem with Glypha III running too slowly on one machine versus another—the problem is that there is a danger of it running too fast on a fast machine! Perhaps you've tried to play a shareware game that suffered from this malady.

In Glypha III, I found a sort of minimum acceptable frame rate. It turns out to be 30 frames per second. At 30 frames per second, the game looks smooth—all the values for the strength of the gravity, the impulse of the wings when flapping, and so on have been adjusted to look smooth at 30 frames per second.

This 30 fps then becomes the maximum fps as well. To ensure the game never runs faster than 30 fps, we take advantage of the Toolbox routine `TickCount()`. Ticks are 1/60th of a second (roughly) on all Macs, so when two Ticks have elapsed, we have our 1/30th of a second. The trick is to read the `TickCount()` before you go into your play loop and then store it off in a variable. When you complete the game loop, sit in a small loop reading `TickCount()` until 2 Ticks elapse since you took the reading at the beginning of the play loop.

In other words, before beginning all of your moving/drawing, check the Ticks. Let's say your Mac has been on for a couple hours, and you get a number back like 7845. Okay, you drop down through your function, moving the enemies, moving the player, reading the player input, playing sounds, drawing, and so on. When you have (finally) executed one frame, you want to make sure that you don't continue back up and begin another frame before two Ticks have passed. So you drop into a small subloop in which you repeatedly read `TickCount()` until a number greater than or equal to 7847 comes up. Then you repeat.

In this way, for example, on a PowerPC a single frame of your game may in fact execute in a quarter of a Tick or so. The rest of the 1 3/4 Ticks are spent waiting for the clock to catch up. Two Ticks were chosen for Glypha III because most machines (perhaps even a IIsi) can execute the loop in about that much time. Using a larger value (like, say, four Ticks for 15 frames per second) makes an arcade game look slow and feel sluggish or unresponsive.

I tried to provide an overview of the source code here and explain in better detail some of the more cryptic aspects of the code. If you walk through the source code, you'll find it lavishly commented. The comments will tell you what is happening moment by moment in Glypha III. To some degree, I think the commenting is a little too exhaustive. I confess that

normally I don't tend to comment code so thoroughly as I did for Glypha III. Often, if the code itself is self-explanatory, leave it be (the standard event loop is perhaps a good example). Also, lengthy, descriptive names for your variables can go a long way toward doing your commenting for you (for example, instead of `i` use `soundIndex`). Lengthy, descriptive function names are a good idea for the same reason (in this day of simple copy and paste, who cares how many characters the function name is long—you're not typing it every time, are you?).

I might say, lastly, two things: If you're having difficulty following the Glypha III source code, try making changes to different parts of it and seeing what the effect is. I have often done this when trying to understand how a program works—substitute in your own graphics and sounds or change some of the constants. And feel free to copy any functions you like from Glypha III to include in your own endeavors. I have to tell you, though, that you're obligated as a programmer to improve upon them— either to make them more robust or to tailor them to your own personal tastes. Share what you learn with others! Have fun!

Sample Glypha Code

```
//============================================================================
//----------------------------------------------------------------------------
//                      Interface.c
//----------------------------------------------------------------------------
//============================================================================

// I put all interface-related code in here. Interface would include event
// handling, menus, dialog boxes, etc. All the user interaction that takes
// place before and after an actual game is in play.

#include "Externs.h"
#include <Sound.h>


#define kAppleMenuID      128
#define iAbout            1
#define kGameMenuID       129
#define iNewGame          1
#define iPauseGame        2
#define iEndGame          3
#define iQuit             5
#define kOptionsMenuID    130
#define iSettings         1
#define iHelp             2
#define iHighScores       3
#define kAboutPictID      132


void DoAppleMenu (short);
void DoGameMenu (short);
```

```
void DoOptionsMenu (short);
void UpdateMainWindow (void);
void HandleMouseEvent (EventRecord *);
void HandleKeyEvent (EventRecord *);
void HandleUpdateEvent (EventRecord *);
void HandleOSEvent (EventRecord *);
void HandleHighLevelEvent (EventRecord *);
void DoAbout (void);
void DoGameSettings (void);


Rect        mainWindowRect;
WindowPtr   mainWindow;
MenuHandle  appleMenu, gameMenu, optionsMenu;
Boolean     switchedOut, quitting, canPlay, openTheScores;

extern  prefsInfo thePrefs;
extern  Rect    backSrcRect, workSrcRect;
extern  CGrafPtr  backSrcMap, workSrcMap;
extern  Boolean pausing, playing, helpOpen, scoresOpen;


//============================================================  Functions
//------------------------------------------------------------  MenusReflectMode()

// Depending on whether a game is in progress (paused) or not, I want
// menu items grayed out in one case and not grayed out in the other.
// This function, when called, displays the menus correctly depending
// on the mode we're in (playing or not playing, pausing or not).

void MenusReflectMode (void)
{
  if (playing)                          // If a game is in progress…
  {
    DisableItem(gameMenu, iNewGame);    // Cannot begin another New Game.
    EnableItem(gameMenu, iPauseGame);   // Can Pause Game.
    if (pausing)                        // If we are paused…
      SetItem(gameMenu, iPauseGame,
           "\pResume Game");            // Rename item "Resume Game."
    else                                // If we are not paused…
      SetItem(gameMenu, iPauseGame,
           "\pPause Game");             // Rename item "Pause Game."
    EnableItem(gameMenu, iEndGame);     // Can End Game.
    DisableItem(optionsMenu, 0);        // Cannot change game settings.
  }
  else                                  // Else, if Glypha is idle…
  {
    EnableItem(gameMenu, iNewGame);     // Can begin a New Game.
    DisableItem(gameMenu, iPauseGame); // Cannot Pause Game.
    SetItem(gameMenu, iPauseGame,
         "\pPause Game");               // Rename item "Pause Game."
    DisableItem(gameMenu, iEndGame);    // Cannot End Game.
```

```
    EnableItem(optionsMenu, 0);        // Can change game settings.
  }
}

//-------------------------------------------------------------- DoAppleMenu()

// This function takes care of handling the Apple menu (Desk Accessories and the
// About box).

void DoAppleMenu (short theItem)
{
  Str255    daName;
  GrafPtr wasPort;
  short     daNumber;

  switch (theItem)                     // Depending on the item selected…
  {
    case iAbout:                       // If the About item was selected…
    if ((scoresOpen) ¦¦ (helpOpen))    // If high scores or help screens up…
    {
      CloseWall();                     // hide them.
      scoresOpen = FALSE;              // High scores no longer open.
      helpOpen = FALSE;                // Help screen is no longer open.
                                       // Uncheck help & high scores menu items.
      CheckItem(optionsMenu, iHelp, helpOpen);
      CheckItem(optionsMenu, iHighScores, scoresOpen);
    }
    DoAbout();                         // Bring up the About dialog.
    break;

    default:                                // If any other item was selected (DA)…
    GetItem(appleMenu, theItem, daName);    // Get the name of the item selected.
    GetPort(&wasPort);                      // Remember our port.
    daNumber = OpenDeskAcc(daName);         // Launch the Desk Accessory.
    SetPort((GrafPtr)wasPort);              // When we return, restore port.
    break;
  }
}

//-------------------------------------------------------------- DoGameMenu()

// This function handles a user's interaction with the Game menu. Quitting
// Glypha, starting a new game, resuming a paused game are handled here.

void DoGameMenu (short theItem)
{
  switch (theItem)                  // Depending on menu item selected…
  {
    case iNewGame:                  // If user selected New Game item…
    if ((scoresOpen) ¦¦ (helpOpen)) // If high scores or help screen is up,
    {                               // close them first.
      CloseWall();
```

```
        scoresOpen = FALSE;
        helpOpen = FALSE;
        CheckItem(optionsMenu, iHelp, helpOpen);
        CheckItem(optionsMenu, iHighScores, scoresOpen);
      }
      InitNewGame();                // Initialize variables for a new game.
      MenusReflectMode();           // Properly gray out the right menu items.
      break;

      case iPauseGame:              // If user selected Pause Game item…
      if (pausing)                  // If we are paused, resume playing.
      {
        pausing = FALSE;            // Turn off pausing flag.
        DumpBackToWorkMap();        // Restore off screen just in case.
      }                             // Actually pausing a game (not resuming)
      break;                        // is not really handled here. It's handled
                                    // directly within the main game loop.

      case iEndGame:                // Ending a game in progress isn't really
      break;                        // handled here -- this is a dummy item.
                                    // Ending a game is handled within the main
                                    // game loop by looking for the Command
                                    // and E key explicitly.

      case iQuit:                   // If user selected Quit item…
      quitting = TRUE;              // Set quitting flag to TRUE.
      break;
    }
}

//-------------------------------------------------------------  DoOptionsMenu()

// This function handles the Options menu. Options include game settings,
// displaying the high scores, and bringing up the Help screen.

void DoOptionsMenu (short theItem)
{
  switch (theItem)              // Depending on which item the user selected…
  {
    case iSettings:             // If user selected Game Settings item…
    if ((scoresOpen) || (helpOpen)) // Close high scores or help screen.
    {
      CloseWall();
      scoresOpen = FALSE;
      helpOpen = FALSE;
      CheckItem(optionsMenu, iHelp, helpOpen);
      CheckItem(optionsMenu, iHighScores, scoresOpen);
    }
    DoGameSettings();           // Bring up game settings dialog.
    break;

    case iHelp:                 // If user selected Help item…
```

```
      if (helpOpen)                // If Help open, close it.
      {
        CloseWall();
        helpOpen = FALSE;
      }
      else                         // Else, if Help is not open -- open it.
      {
        if (scoresOpen)            // If the High Scores are up though,…
        {
          CloseWall();             // Close them first.
          scoresOpen = FALSE;
          CheckItem(optionsMenu, iHighScores, scoresOpen);
        }
        OpenHelp();                // Now open the Help screen.
      }
      CheckItem(optionsMenu, iHelp, helpOpen);
      break;

      case iHighScores:            // If user selected High Scores…
      if (scoresOpen)              // If the High Scores are up, close them.
      {
        CloseWall();
        scoresOpen = FALSE;
      }
      else                         // If the High Scores are not up…
      {
        if (helpOpen)              // First see if Help is open.
        {
          CloseWall();             // And close the Help screen.
          helpOpen = FALSE;
          CheckItem(optionsMenu, iHelp, helpOpen);
        }
        OpenHighScores();          // Now open the High Scores.
      }
      CheckItem(optionsMenu, iHighScores, scoresOpen);
      break;
    }
}

//-------------------------------------------------------------  DoMenuChoice()

// This is the main menu-handling function. It examines which menu was selected
// by the user and passes on to the appropriate function, the item within that
// menu that was selected.

void DoMenuChoice (long menuChoice)
{
  short     theMenu, theItem;

  if (menuChoice == 0)             // A little error checking.
    return;
```

```
    theMenu = HiWord(menuChoice); // Extract which menu was selected.
    theItem = LoWord(menuChoice); // Extract which item it was that was selected.

    switch (theMenu)              // Now, depending on which menu was selected…
    {
      case kAppleMenuID:          // If the Apple menu selected…
      DoAppleMenu(theItem);       // Call the function that handles the Apple menu.
      break;

      case kGameMenuID:           // If the Game menu selected…
      DoGameMenu(theItem);        // Call the function that handles the Game menu.
      break;

      case kOptionsMenuID:        // If the Options menu selected…
      DoOptionsMenu(theItem);     // Call the function that handles the Options menu.
      break;
    }

    HiliteMenu(0);                // "De-invert" menu.
}

//-------------------------------------------------------------  UpdateMainWindow()

// This is a simple function that simply copies the contents from the
// background off-screen pixmap to the main screen. It is primarily
// called in response to an update event, but could be called any time
// when I want to force the screen to be redrawn.

void UpdateMainWindow (void)
{
    CopyBits(&((GrafPtr)backSrcMap)->portBits,
         &(((GrafPtr)mainWindow)->portBits),
         &mainWindowRect, &mainWindowRect,
         srcCopy, 0L);
}

//-------------------------------------------------------------  HandleMouseEvent()

// Mouse clicks come here. This is standard event handling drivel. No different
// from any other standard Mac program (game or otherwise).

void HandleMouseEvent (EventRecord *theEvent)
{
    WindowPtr whichWindow;
    long    menuChoice;
    short     thePart;
                              // Determine window and where in window.
    thePart = FindWindow(theEvent->where, &whichWindow);

    switch (thePart)                // Depending on where mouse was clicked…
    {
      case inSysWindow:             // In a Desk Accessory.
```

```
        SystemClick(theEvent, whichWindow);     // (Is this stuff obsolete yet?)
        break;

        case inMenuBar:                 // Selected a menu item.
        menuChoice = MenuSelect(theEvent->where);
        if (canPlay)                    // Call menu handling routine.
          DoMenuChoice(menuChoice);
        break;

        case inDrag:                    // Like the lazy bastard I am…
        case inGoAway:                  // I'll just ignore these.
        case inGrow:                    // But, hey, the window isn't…
        case inZoomIn:                  // moveable or growable!
        case inZoomOut:
        break;

        case inContent:                 // Click in the window itself.
        FlashObelisks(TRUE);            // Do lightning animation.
        LogNextTick(3);                 // Lightning will hit cursor location.
        GenerateLightning(theEvent->where.h, theEvent->where.v - 20);
        StrikeLightning();
        WaitForNextTick();
        StrikeLightning();
        LogNextTick(2);
        WaitForNextTick();
        PlayExternalSound(kLightningSound, kLightningPriority);
        LogNextTick(3);
        GenerateLightning(theEvent->where.h, theEvent->where.v - 20);
        StrikeLightning();
        WaitForNextTick();
        StrikeLightning();
        LogNextTick(2);
        WaitForNextTick();
        LogNextTick(3);
        GenerateLightning(theEvent->where.h, theEvent->where.v - 20);
        StrikeLightning();
        WaitForNextTick();
        StrikeLightning();
        LogNextTick(2);
        WaitForNextTick();
        PlayExternalSound(kLightningSound, kLightningPriority);
        LogNextTick(3);
        GenerateLightning(theEvent->where.h, theEvent->where.v - 20);
        StrikeLightning();
        WaitForNextTick();
        StrikeLightning();
        LogNextTick(2);
        WaitForNextTick();
        FlashObelisks(FALSE);
        break;
    }
}
```

```
//-------------------------------------------------------------  HandleKeyEvent()

// More standard issue. This function handles any keystrokes when no game is
// in session. Command-key strokes handled here too.

void HandleKeyEvent (EventRecord *theEvent)
{
  char    theChar;
  Boolean   commandDown;

  theChar = theEvent->message & charCodeMask;            // Extract key hit.
  commandDown = ((theEvent->modifiers & cmdKey) != 0);   // See if Command key down.

  if (commandDown)                                  // If Command key down, call menu
  {                                                 // handling routine.
    if (canPlay)
      DoMenuChoice(MenuKey(theChar));
  }
  else
  {
    if (helpOpen)                                   // Handle special keys if the help
    {                                               // screen is up.
      if (theChar == kUpArrowKeyASCII)              // Up Arrow key scrolls help down.
      {
        if (theEvent->what == autoKey)
          ScrollHelp(-3);
        else
          ScrollHelp(-1);
      }
      else if (theChar == kDownArrowKeyASCII) // Down Arrow key scrolls help up.
      {
        if (theEvent->what == autoKey)
          ScrollHelp(3);
        else
          ScrollHelp(1);
      }
      else if (theChar == kPageDownKeyASCII)  // Handle Page Down for help screen.
      {
        ScrollHelp(199);
      }
      else if (theChar == kPageUpKeyASCII)    // Handle Page Up for help.
      {
        ScrollHelp(-199);
      }
      else if ((theChar == kHelpKeyASCII) && (!playing))
      {                                             // Hitting Help key closes help
        CloseWall();                                // (if it's already open).
        helpOpen = FALSE;
        CheckItem(optionsMenu, iHelp, helpOpen);
      }
```

```
    }
    else if ((theChar == kHelpKeyASCII) && (!playing))
    {                                   // Else, if help not open and Help
      if (scoresOpen)                   // key is hit, open Help.
      {                                 // Close high scores if open.
        CloseWall();
        scoresOpen = FALSE;
        CheckItem(optionsMenu, iHighScores, scoresOpen);
      }
      OpenHelp();                       // Open help.
      CheckItem(optionsMenu, iHelp, helpOpen);
    }
  }
}

//-------------------------------------------------------------  HandleUpdateEvent()

// This function handles update events. Standard event handling stuff.

void HandleUpdateEvent (EventRecord *theEvent)
{
  if ((WindowPtr)theEvent->message == mainWindow)
  {
    SetPort((GrafPtr)mainWindow);       // Don't forget this line, BTW.
    BeginUpdate((GrafPtr)mainWindow);   // I did once and it took me
    UpdateMainWindow();                 // ages to track down that bug.
    EndUpdate((GrafPtr)mainWindow);     // Well, it took me a week I think.
    canPlay = TRUE;
  }
}

//-------------------------------------------------------------  HandleOSEvent()

// Handle switching in and out events. Standard event handling stuff.

void HandleOSEvent (EventRecord *theEvent)
{
  if (theEvent->message & 0x01000000)     // If suspend or resume event…
  {
    if (theEvent->message & 0x00000001)   // Specifically, if resume event…
      switchedOut = FALSE;                // I keep thinking I should do more here.
    else                                  // Or if suspend event…
      switchedOut = TRUE;                 // What am I forgetting?
  }
}

//-------------------------------------------------------------  HandleHighLevelEvent()

// Again, it's a fact I'm lazy. Apple Events are fairly easy to implement but
```

```
// a nightmare to try and explain. Filling out the below function is left as
// an exercise to the reader.

void HandleHighLevelEvent (EventRecord *theEvent)
{
// theErr = AEProcessAppleEvent(theEvent);
}

//-------------------------------------------------------------  HandleEvent()

// Standard event stuff. This is the calling function that calls all the above
// functions. It looks for an event and if it detects one, it calls the appropriate
// function above to handle it.

void HandleEvent (void)
{
  EventRecord  theEvent;
  long    sleep = 1L;
  Boolean   itHappened;
                                  // See if an event is queued up.
  itHappened = WaitNextEvent(everyEvent, &theEvent, sleep, 0L);

  if (itHappened)                 // Ah, an event. I live for events!
  {
    switch (theEvent.what)        // And what kind of event be ya'?
    {
      case mouseDown:             // Aiy! Y' be a mouse click do ya'?
      HandleMouseEvent(&theEvent);
      break;

      case keyDown:               // Key-down, key-held-down events.
      case autoKey:
      HandleKeyEvent(&theEvent);
      break;

      case updateEvt:             // Something needs redrawing!
      HandleUpdateEvent(&theEvent);
      break;

      case osEvt:                 // Switching in and out events.
      HandleOSEvent(&theEvent);
      break;

      case kHighLevelEvent:       // Hmmmm. A "what" event?
      HandleHighLevelEvent(&theEvent);
      break;
    }
  }
  else if (openTheScores)         // Check for "auto open" flag.
  {                               // If TRUE, set the flag back to
    openTheScores = FALSE;        // FALSE and open the high scores.
    OpenHighScores();
```

```
    }
}

//-------------------------------------------------------------- DoAbout()

// This handles the About dialog. It brings up the About box in a
// simple centered window with no drag bar, close box or anything.
// Leaving the dialog is handled with a simple mouse click.

void DoAbout (void)
{
   Rect    aboutRect;
   WindowPtr aboutWindow;

   SetRect(&aboutRect, 0, 0, 325, 318); // Bring up centered window.
   CenterRectInRect(&aboutRect, &qd.screenBits.bounds);
   aboutWindow = GetNewCWindow(129, 0L, kPutInFront);
   MoveWindow((GrafPtr)aboutWindow, aboutRect.left, aboutRect.top, TRUE);
   ShowWindow((GrafPtr)aboutWindow);
   SetPort((GrafPtr)aboutWindow);
   LoadGraphic(kAboutPictID);            // Draw About dialog graphic.

   do                                    // Make sure button not down
   {                                     // before proceeding.
   }
   while (Button());                     // Proceed.
   do                                    // And now wait until the mouse
   {                                     // is pressed before closing the
   }                                     // window (About dialog).
   while (!Button());

   FlushEvents(everyEvent, 0);           // Flush the queue.

   if (aboutWindow != 0L)
      DisposeWindow(aboutWindow);        // Close the About dialog.
}

//-------------------------------------------------------------- DoGameSettings()

// This one however is a good and proper dialog box. It handles the meager
// preference settings for Glypha. Nothing fancy here to report. Just a
// straightforward dialog-calling routine.

void DoGameSettings (void)
{
   #define   kGameSettingsDialogID 133
   DialogPtr theDial;
   long    newVolume;
   short     i, item;
   Boolean   leaving;

   CenterDialog(kGameSettingsDialogID); // Center dialog, then call up.
```

```
    theDial = GetNewDialog(kGameSettingsDialogID, 0L, kPutInFront);
    SetPort((GrafPtr)theDial);
    ShowWindow((GrafPtr)theDial);          // Make visible (after centering).
    DrawDefaultButton(theDial);            // Draw border around OK button.
    FlushEvents(everyEvent, 0);
                                           // Put in a default sound volume.
    SetDialogNumToStr(theDial, 3, (long)thePrefs.wasVolume);
    SelIText(theDial, 3, 0, 1024);         // Select it.

    leaving = FALSE;

    while (!leaving)
    {
      ModalDialog(0L, &item);              // Simple modal dialog filtering.

      if (item == 1)                       // Did user hit the OK button?
      {                                    // We'll see if volume entered is legal.
        GetDialogNumFromStr(theDial, 3, &newVolume);
        if ((newVolume >= 0) && (newVolume <= 7))
        {                                  // If it is legal, we'll note it and quit.
          thePrefs.wasVolume = (short)newVolume;
          SetSoundVol((short)newVolume);
          leaving = TRUE;                  // Bye.
        }
        else                               // Otherwise, the volume entered is wrong.
        {                                  // So we'll Beep, enter the last legal
          SysBeep(1);                      // value and select the text again.
          SetDialogNumToStr(theDial, 3, (long)thePrefs.wasVolume);
          SelIText(theDial, 3, 0, 1024);
        }
      }
      else if (item == 2)                  // Did the user hit the "Clear Scores"
      {                                    // button?
        for (i = 0; i < 10; i++)           // Walk through and zero scores.
        {
          PasStringCopy("\pNemo", thePrefs.highNames[i]);
          thePrefs.highScores[i] = 0L;
          thePrefs.highLevel[i] = 0;
          openTheScores = TRUE;            // Bring up scores when dialog quits.
        }
        DisableControl(theDial, 2);        // Gray out Clear Scores button.
      }
    }

    DisposDialog(theDial);                 // Clean up before going.
}
```

```
//============================================================================
//----------------------------------------------------------------------------
//                      Enemy.c
//----------------------------------------------------------------------------
//============================================================================

// This file contains all enemy-related functions (enemy "AI"). It handles
// the enemy decision-making process, moves the enemies, etc.

#include "Externs.h"


#define kEnemyImpulse       8

#define kOwlMaxHVel         96
#define kOwlMaxVVel         320
#define kOwlHeightSmell     96
#define kOwlFlapImpulse     32

#define kWolfMaxHVel        128
#define kWolfMaxVVel        400
#define kWolfHeightSmell    160
#define kWolfFlapImpulse    48

#define kJackalMaxHVel      192
#define kJackalMaxVVel      512
#define kJackalHeightSmell  240
#define kJackalFlapImpulse  72


Boolean SetEnemyInitialLocation (Rect *);
void SetEnemyAttributes (short);
short AssignNewAltitude (void);
void InitEnemy (short, Boolean);
void CheckEnemyPlatformHit (short);
void CheckEnemyRoofCollision (short);
void HandleIdleEnemies (short);
void HandleFlyingEnemies (short);
void HandleWalkingEnemy (short);
void HandleSpawningEnemy (short);
void HandleFallingEnemy (short);
void HandleEggEnemy (short);
void ResolveEnemyPlayerHit (short);


handInfo  theHand;
eyeInfo   theEye;
Rect      grabZone;
short     deadEnemies, spawnedEnemies, numEnemiesThisLevel, numOwls;
```

```
extern  playerType   thePlayer;
extern  enemyType    theEnemies[kMaxEnemies];
extern  Rect         platformRects[6], enemyInitRects[5];
extern  long         theScore;
extern  short        numLedges, lightningCount, numEnemies, countDownTimer;
extern  short        levelOn, lightH, lightV;
extern  Boolean      evenFrame, doEnemyFlapSound, doEnemyScrapeSound;


//======================================================  Functions
//------------------------------------------------------  SetEnemyInitialLocation()

// When a new enemy is about to be "born," this function is called to determine
// the enemy's starting location. The only thing important here is that the enemy
// appears on a valid platform for the particular level we're on. As well, which
// platform he (it) appears on should be random.

Boolean SetEnemyInitialLocation (Rect *theRect)
{
   short     where, possibilities;
   Boolean   facing;

   possibilities = numLedges - 1;      // Determine number of valid platforms.
   where = RandomInt(possibilities);   // Choose one at random.
   *theRect = enemyInitRects[where];   // Initially place enemy at default location.

   switch (where)                      // Determine if enemy facing left or right.
   {                                   // It depends upon which platform they're on.
      case 0:                          // These are the left-most platforms.
      case 2:
      facing = TRUE;                   // Enemy will face right.
      break;

      case 3:                          // Special case for the center platform.
      if (RandomInt(2) == 0)           // Enemy randomly faces either left or right.
         facing = TRUE;
      else
         facing = FALSE;
      break;

      default:                         // Catch remaining (right-most) platforms.
      facing = FALSE;                  // Enemy will face left.
      break;
   }

   if ((levelOn % 5) == 4)             // Handle special case for Egg Wave
   {                                   // Redefine enemy bounds.
      theRect->left += 12 + RandomInt(48) - 24;
      theRect->right = theRect->left + 24;
      theRect->top = theRect->bottom - 24;
   }

   return (facing);
}
```

```
//------------------------------------------------------------ SetEnemyAttributes()

// Depending upon the type of enemy this function is passed (there are three
// types of sphinx enemies), this function sets up that enemy's various
// attributes -- such as maximum vertical velocity, etc.

void SetEnemyAttributes (short i)
{
  short   h;
                                    // Point enemy toward center of screen.
  h = (theEnemies[i].dest.left + theEnemies[i].dest.right) >> 1;
  if (h < 320)                      // If enemy in left half of screen…
    theEnemies[i].facingRight = TRUE;  // the enemy will face to the right.
  else                              // Otherwise, if in right half of screen…
    theEnemies[i].facingRight = FALSE; // face to the left.

  switch (theEnemies[i].kind)       // Okay, depending upon what "kind" of enemy
  {                                 // we're dealing with…
    case kOwl:                      // The owl is the simplest (wimpiest) enemy.
    if (theEnemies[i].facingRight)  // Choose which graphic to use.
      theEnemies[i].srcNum = 0;
    else
      theEnemies[i].srcNum = 2;
                                    // Set owl's velocity limitations.
    theEnemies[i].maxHVel = kOwlMaxHVel;
    theEnemies[i].maxVVel = kOwlMaxVVel;
                                    // This is the distance within which he will
                                    // pursue the player (it's strictly Y ➥
distance).
    theEnemies[i].heightSmell = kOwlHeightSmell;
                                    // This is how powerful the owl's "flap" is.
    theEnemies[i].flapImpulse = kOwlFlapImpulse;
    break;

    case kWolf:                     // The wolf sphinx is of medium difficulty.
    if (theEnemies[i].facingRight)  // Choose which graphic to use.
      theEnemies[i].srcNum = 4;
    else
      theEnemies[i].srcNum = 6;
                                    // Set wolf's velocity limitations.
    theEnemies[i].maxHVel = kWolfMaxHVel;
    theEnemies[i].maxVVel = kWolfMaxVVel;
                                    // This is the distance within which he will
                                    // pursue the player (it's strictly Y ➥
distance).
    theEnemies[i].heightSmell = kWolfHeightSmell;
                                    // This is how powerful the wolf's "flap" is.
    theEnemies[i].flapImpulse = kWolfFlapImpulse;
    break;

    case kJackal:                   // The jackal is the swiftest, toughest ➥
enemy.
    if (theEnemies[i].facingRight)  // Choose which graphic to use.
      theEnemies[i].srcNum = 8;
```

```
      else
        theEnemies[i].srcNum = 10;
                                        // Set jackal's velocity limitations.
      theEnemies[i].maxHVel = kJackalMaxHVel;
      theEnemies[i].maxVVel = kJackalMaxVVel;
                                        // This is the distance within which he will
                                        // pursue the player (it's strictly Y distance).
      theEnemies[i].heightSmell = kJackalHeightSmell;
                                        // This is how powerful the jackal's "flap" is.
      theEnemies[i].flapImpulse = kJackalFlapImpulse;
      break;
    }
}

//-------------------------------------------------------------  AssignNewAltitude()

// The sphinxes "patrol" specific altitudes in the arena. After wrapping around
// the screen a few times, they randomly select a new altitude to patrol (this
// keeps the player from finding a "safe" place to stand. This function chooses
// a new altitude for the enemy to patrol.

short AssignNewAltitude (void)
{
  short     which, altitude;

  which = RandomInt(4);          // There are only 4 "patrol altitudes."
  switch (which)                 // Depending on which random number came up…
  {
    case 0:                      // This is just below the ceiling.
    altitude = 65 << 4;
    break;

    case 1:                      // This is below the top platforms but above the
    altitude = 150 << 4;         // center platform.
    break;

    case 2:                      // This is just below the center platform.
    altitude = 245 << 4;
    break;

    case 3:                      // This is straight across the lava pit.
    altitude = 384 << 4;
    break;
  }

  return (altitude);
}

//-----------------------------------------------------------------  InitEnemy()

// This resets an enemy's info. It is called when a new enemy is to be born.
// It is called if an egg is about to hatch, if a new level has begun, or if
// if it is simply time to add a new enemy.
```

```
void InitEnemy (short i, Boolean reincarnated)
{
  Boolean   facing;

  if (spawnedEnemies < numEnemiesThisLevel)  // New enemy to appear (in other words
  {                                          // this enemy is not hatched).
                                             // Call function to set new location.
    facing = SetEnemyInitialLocation(&theEnemies[i].dest);
    theEnemies[i].wasDest = theEnemies[i].dest;
    theEnemies[i].h = theEnemies[i].dest.left << 4;
    theEnemies[i].v = theEnemies[i].dest.top << 4;
    theEnemies[i].wasH = theEnemies[i].h;   // Reset "old locations" variables.
    theEnemies[i].wasV = theEnemies[i].v;
                            // Assign the "patrol altitude."
    theEnemies[i].targetAlt = theEnemies[i].v - (40 << 4);
    theEnemies[i].hVel = 0;                 // Zero velocity variables.
    theEnemies[i].vVel = 0;
    theEnemies[i].pass = 0;                 // Zero number of times wrapped around.
    if ((levelOn % 5) == 4)                 // If this is an Egg Wave…
      theEnemies[i].mode = kEggTimer;       // set enemy in "wait to hatch" mode.
    else                                    // Otherwise, just put enemy in
      theEnemies[i].mode = kIdle;           // idle mode.
    if (i < numOwls)                        // Determine what kind of enemy.
      theEnemies[i].kind = kOwl;
    else if (i > (numOwls + 6))
      theEnemies[i].kind = kJackal;
    else
      theEnemies[i].kind = kWolf;
    theEnemies[i].facingRight = facing;
    SetEnemyAttributes(i);                  // Initialize enemy attributes.

    if (reincarnated)                       // If this is an egg that will hatch…
      theEnemies[i].frame = RandomInt(48) + 8 + (numOwls * 32);
    else
      theEnemies[i].frame = RandomInt(48) + 32 + (64 * i) + (numOwls * 32);

    if ((levelOn % 5) == 4)                 // If this is an Egg Wave
      theEnemies[i].kind--;                 // Decrement "kind" (since it's ➥
incremented
                                            // when they hatch).
    spawnedEnemies++;                       // Keep track of number of enemies ➥
active.
  }
}

//-------------------------------------------------------------  GenerateEnemies()

// This function is called only for a new level. It goes through and
// intializes a whole host of enemies in one go.

void GenerateEnemies (void)
{
  short     i;
```

```
  if ((levelOn % 5) == 4)      // If this is an Egg Wave…
  {
    numEnemies = kMaxEnemies;  // we insist upon the maximum number of enemies.
    numEnemiesThisLevel = numEnemies;
  }
  else                         // If not an egg wave, use a formula to determine
  {                            // the max number of enemies that are to be active.
    numEnemies = ((levelOn / 5) + 2) * 2;
    if (numEnemies > kMaxEnemies)
      numEnemies = kMaxEnemies;
    numEnemiesThisLevel = numEnemies * 2;
  }

  deadEnemies = 0;             // No dead enemies yet.

                               // Use formula to determine the number of owls
                               // to appear. This number goes down as the levels
                               // increase. It is used not merely to determine
                               // how many owls are to appear, but also how many
                               // of the more advanced enemies. For example, when
                               // numOwls goes down to zero, all the enemies will
                               // be of the more advanced breed (wolves and jackals).
  numOwls = 4 - ((levelOn + 2) / 5);
  if (numOwls < 0)
    numOwls = 0;

  spawnedEnemies = 0;          // No enemies have been "born" yet.
                               // Go through and set up all the enemies.
  for (i = 0; i < numEnemies; i++)
    InitEnemy(i, FALSE);
}

//-------------------------------------------------------  CheckEnemyPlatformHit()

// This is the enemy counterpart to a similarly named function that tests for
// player collisions with the platforms.

void CheckEnemyPlatformHit (short h)
{
  Rect   hRect, vRect, whoCares;
  short    i, offset;

  for (i = 0; i < numLedges; i++)          // Test all platforms.
  {                                        // Do a simple bounds test.
    if (SectRect(&theEnemies[h].dest, &platformRects[i], &whoCares))
    {                                      // If the enemy has hit the platform…
      hRect.left = theEnemies[h].dest.left; // Determine if enemy hit platform ➥
sides.
      hRect.right = theEnemies[h].dest.right;
      hRect.top = theEnemies[h].wasDest.top;
      hRect.bottom = theEnemies[h].wasDest.bottom;
                                     // Test this new special rect to see if
                                     // the enemy hit one of the platform sides.
```

```
if (SectRect(&hRect, &platformRects[i], &whoCares))
{                                   // If enemy hit from side, see which side.
                                    // We handle left and right separately
                                    // so that there's no ambiguity as to
                                    // what the new velocity and location
                                    // of the enemy is. If we did not do it
                                    // this way, there is the chance that an
                                    // enemy gets "stuck" on the edge of
                                    // a platform (due to round-off errors).
  if (theEnemies[h].h > theEnemies[h].wasH)
  {                                 // Enemy was moving right (hit left side).
    offset = theEnemies[h].dest.right - platformRects[i].left;
                                    // Slide enemy "off" platform.
    theEnemies[h].dest.left -= offset;
    theEnemies[h].dest.right -= offset;
    theEnemies[h].h = theEnemies[h].dest.left << 4;
    theEnemies[h].wasH = theEnemies[h].h;
                                    // Bounce enemy (negate velocity).
    if (theEnemies[h].hVel > 0)
      theEnemies[h].hVel = -(theEnemies[h].hVel >> 1);
    else
      theEnemies[h].hVel = theEnemies[h].hVel >> 1;
  }
  if (theEnemies[h].h < theEnemies[h].wasH)
  {                                 // Enemy was moving left (hit right side).
    offset = platformRects[i].right - theEnemies[h].dest.left;
                                    // Slide enemy "off" platform.
    theEnemies[h].dest.left += offset;
    theEnemies[h].dest.right += offset;
    theEnemies[h].h = theEnemies[h].dest.left << 4;
    theEnemies[h].wasH = theEnemies[h].h;
                                    // Bounce enemy (negate velocity).
    if (theEnemies[h].hVel < 0)
      theEnemies[h].hVel = -(theEnemies[h].hVel >> 1);
    else
      theEnemies[h].hVel = theEnemies[h].hVel >> 1;
  }
  doEnemyScrapeSound = TRUE;        // Play a collision sound.
                                    // Flip enemy to face opposite direction.
  theEnemies[h].facingRight = !theEnemies[h].facingRight;
}
else                                // Enemy didn't hit from side.
{                                   // See if enemy hit top/bottom.
  vRect.left = theEnemies[h].wasDest.left;
  vRect.right = theEnemies[h].wasDest.right;
  vRect.top = theEnemies[h].dest.top;
  vRect.bottom = theEnemies[h].dest.bottom;
                                    // Special "test rect" for top/bottom hit.
  if (SectRect(&vRect, &platformRects[i], &whoCares))
  {                                 // If hit the top/bottom of platform…
    if (theEnemies[h].mode == kFalling)
    {                               // Was the enemy a falling egg?
```

```
                        // Bounce egg (with some inelasticity).
    theEnemies[i].hVel -= (theEnemies[i].hVel >> 3);
                        // When the egg's velocity is between
                        // +/- 8, consider the egg at rest.
    if ((theEnemies[i].hVel < 8) && (theEnemies[i].hVel > -8))
    {
      if (theEnemies[i].hVel > 0)
        theEnemies[i].hVel--;
      else if (theEnemies[i].hVel < 0)
        theEnemies[i].hVel++;
    }
  }
                        // Specifically, did enemy hit the top?
  if (theEnemies[h].v > theEnemies[h].wasV)
  {                     // Enemy heading down (hit platform top).
    offset = theEnemies[h].dest.bottom - platformRects[i].top;
                        // Move enemy up off platform.
    theEnemies[h].dest.top -= offset;
    theEnemies[h].dest.bottom -= offset;
    theEnemies[h].v = theEnemies[h].dest.top << 4;
    theEnemies[h].wasV = theEnemies[h].v;
    if (theEnemies[h].vVel > kDontFlapVel)
      doEnemyScrapeSound = TRUE;
                        // "Bounce" enemy.
    if (theEnemies[h].vVel > 0)
      theEnemies[h].vVel = -(theEnemies[h].vVel >> 1);
    else
      theEnemies[h].vVel = theEnemies[h].vVel >> 1;
    if ((theEnemies[h].vVel < 8) && (theEnemies[h].vVel > -8) &&
        (theEnemies[h].hVel == 0) && (theEnemies[h].mode == kFalling))
    {                   // Here we handle an egg come to rest.
      if (((theEnemies[h].dest.right - 8) > platformRects[i].right) &&
          (theEnemies[h].hVel == 0))
      {                 // Special case where egg right on edge.
        theEnemies[h].hVel = 32;
      }
      else if (((theEnemies[h].dest.left + 8) < platformRects[i].left) &&
          (theEnemies[h].hVel == 0))
      {                 // Special case where egg right on edge.
        theEnemies[h].hVel = -32;
      }
      else              // If egg not on the edge of platform…
      {                 // switch to "timer" mode.
        theEnemies[h].mode = kEggTimer;
        theEnemies[h].frame = (numOwls * 96) + 128;
        theEnemies[h].vVel = 0;
      }
    }
  }
  if (theEnemies[h].v < theEnemies[h].wasV)
  {                     // Enemy was rising -- hit bottom of platform.
    offset = theEnemies[h].dest.top - platformRects[i].bottom;
```

```
                                    // Slide enemy off platform.
                theEnemies[h].dest.top -= offset;
                theEnemies[h].dest.bottom -= offset;
                theEnemies[h].v = theEnemies[h].dest.top << 4;
                theEnemies[h].wasV = theEnemies[h].v;
                                    // Play collision sound.
                doEnemyScrapeSound = TRUE;
                                    // "Bounce" enemy downward from platform.
                if (theEnemies[h].vVel < 0)
                  theEnemies[h].vVel = -(theEnemies[h].vVel >> 2);
                else
                  theEnemies[h].vVel = theEnemies[h].vVel >> 2;
                if ((theEnemies[h].vVel < 8) && (theEnemies[h].vVel > -8) &&
                     (theEnemies[h].hVel == 0) && (theEnemies[h].mode == kFalling))
                {
                  theEnemies[h].mode = kEggTimer;
                  theEnemies[h].frame = (numOwls * 96) + 128;
                  theEnemies[h].vVel = 0;
                }
              }
            }
          }
        }
    }
}

//-------------------------------------------------------  CheckEnemyRoofCollision()

// Like the player counterpart, this function checks to see if an enemy has hit
// the ceiling or the lava. It handles the consequences of both cases.

void CheckEnemyRoofCollision (short i)
{
  short     offset;

  if (theEnemies[i].dest.top < (kRoofHeight - 2))
  {                         // If enemy has hit the ceiling…
    offset = kRoofHeight - theEnemies[i].dest.top;
                            // Move enemy down to a "legal" altitude.
    theEnemies[i].dest.top += offset;
    theEnemies[i].dest.bottom += offset;
    theEnemies[i].v = theEnemies[i].dest.top << 4;
                            // Play a collision sound.
    doEnemyScrapeSound = TRUE;
                            // Bounce enemy downward.
    theEnemies[i].vVel = -(theEnemies[i].vVel >> 2);
  }
  else if (theEnemies[i].dest.top > kLavaHeight)
  {                         // If enemy has fallen into lava…
                            // kill that enemy.
    theEnemies[i].mode = kDeadAndGone;
    deadEnemies++;
```

```
                                // Play a splash sound.
      PlayExternalSound(kSplashSound, kSplashPriority);
                                // Call up another from the ranks.
      InitEnemy(i, TRUE);
    }
  }

  //-------------------------------------------------------------  HandleIdleEnemies()

  // The following functions handle the various enemy modes. Enemies are
  // considered to be in a specific mode and each mode is handled differently.
  // Idle enemies are ones who are "invisible" -- not yet born. While idle, a
  // timer is ticking down -- when it reaches zero, the enemy appears.

  void HandleIdleEnemies (short i)
  {
    theEnemies[i].frame--;              // Decrement timer.
    if (theEnemies[i].frame <= 0)       // If timer is zero or less…
    {
      theEnemies[i].mode = kSpawning;   // enemy is "born."
      theEnemies[i].wasH = theEnemies[i].h;
      theEnemies[i].wasV = theEnemies[i].v;
      theEnemies[i].hVel = 0;
      theEnemies[i].vVel = 0;
      theEnemies[i].frame = 0;
      SetEnemyAttributes(i);            // Initialize enemy attributes.
      PlayExternalSound(kSpawnSound, kSpawnPriority);
    }
  }

  //-------------------------------------------------------------  HandleFlyingEnemies()

  // Once an enemy takes off from a platform, they will always be in flying mode
  // unless they should be killed. This function handles the flying mode.

  void HandleFlyingEnemies (short i)
  {
    short     dist;
    Boolean   shouldFlap;
                        // Take into account gravity pulling enemy down.
    theEnemies[i].vVel += kGravity;
                        // Get absolute difference in enemy/player altitude.
    dist = thePlayer.dest.top - theEnemies[i].dest.top;
    if (dist < 0)
      dist = -dist;
                        // See if the player is within the enemy's "seek" range.
    if ((dist < theEnemies[i].heightSmell) &&
        ((thePlayer.mode == kFlying) || (thePlayer.mode == kWalking)))
    {                   // Enemy will actively seek the player.
      if (thePlayer.dest.left < theEnemies[i].dest.left)
      {                 // Determine if quicker to go left or right to get player.
        dist = theEnemies[i].dest.left - thePlayer.dest.left;
```

```
      if (dist < 320)   // Closest route is to the left.
        theEnemies[i].facingRight = FALSE;
      else              // Closest route is to the right.
        theEnemies[i].facingRight = TRUE;
    }
    else if (thePlayer.dest.left > theEnemies[i].dest.left)
    {                   // Determine if quicker to go left or right to get player.
      dist = thePlayer.dest.left - theEnemies[i].dest.left;
      if (dist < 320)   // Closest route is to the right.
        theEnemies[i].facingRight = TRUE;
      else              // Closest route is to the left.
        theEnemies[i].facingRight = FALSE;
    }
                        // Seek an altitude 16 pixels above player.
    if (((theEnemies[i].v + 16) > thePlayer.v) && (evenFrame))
      shouldFlap = TRUE;
    else
      shouldFlap = FALSE;
  }
  else                  // Else, player not within enemy's "seek" altitude.
  {                     // Flap if necessary to maintain "patrol altitude."
    if ((theEnemies[i].v > theEnemies[i].targetAlt) && (evenFrame))
      shouldFlap = TRUE;
    else
      shouldFlap = FALSE;
  }

  if (shouldFlap)       // If the enemy has determined that it needs to flap…
  {                     // Give the enemy lift & play the flap sound.
    theEnemies[i].vVel -= theEnemies[i].flapImpulse;
    doEnemyFlapSound = TRUE;
  }
                        // Enemy never hovers -- must move right or left.
  if (theEnemies[i].facingRight)
  {                     // If enemy facing right -- move enemy to the right.
    theEnemies[i].hVel += kEnemyImpulse;
    if (theEnemies[i].hVel > theEnemies[i].maxHVel)
      theEnemies[i].hVel = theEnemies[i].maxHVel;
                        // Determine correct graphic for enemy.
    switch (theEnemies[i].kind)
    {
      case kOwl:
      if (shouldFlap)
        theEnemies[i].srcNum = 12;
      else
        theEnemies[i].srcNum = 13;
      break;

      case kWolf:
      if (shouldFlap)
        theEnemies[i].srcNum = 16;
      else
```

```
            theEnemies[i].srcNum = 17;
         break;

         case kJackal:
         if (shouldFlap)
            theEnemies[i].srcNum = 20;
         else
            theEnemies[i].srcNum = 21;
         break;
      }

   }
   else              // If enemy not facing right (left) move to the left.
   {
      theEnemies[i].hVel -= kEnemyImpulse;
      if (theEnemies[i].hVel < -theEnemies[i].maxHVel)
         theEnemies[i].hVel = -theEnemies[i].maxHVel;
                     // Determine correct graphic for enemy.
      switch (theEnemies[i].kind)
      {
         case kOwl:
         if (shouldFlap)
            theEnemies[i].srcNum = 14;
         else
            theEnemies[i].srcNum = 15;
         break;

         case kWolf:
         if (shouldFlap)
            theEnemies[i].srcNum = 18;
         else
            theEnemies[i].srcNum = 19;
         break;

         case kJackal:
         if (shouldFlap)
            theEnemies[i].srcNum = 22;
         else
            theEnemies[i].srcNum = 23;
         break;
      }
   }
                        // Move enemy horizontally based on horizontal velocity.
   theEnemies[i].h += theEnemies[i].hVel;
   theEnemies[i].dest.left = theEnemies[i].h >> 4;
   theEnemies[i].dest.right = theEnemies[i].dest.left + 64;
                        // Move enemy vertically based on vertical velocity.
   theEnemies[i].v += theEnemies[i].vVel;
   theEnemies[i].dest.top = theEnemies[i].v >> 4;
   theEnemies[i].dest.bottom = theEnemies[i].dest.top + 40;
                        // Check for wraparound.
   if (theEnemies[i].dest.left > 640)
```

```
    {                   // If off right edge, wrap around to left side.
      OffsetRect(&theEnemies[i].dest, -640, 0);
      theEnemies[i].h = theEnemies[i].dest.left << 4;
      OffsetRect(&theEnemies[i].wasDest, -640, 0);
      theEnemies[i].pass++;       // Increment number of "wraparounds" for this enemy.
      if (theEnemies[i].pass > 2) // After two screen passes (wraparounds)
      {                           // enemy patrols a new altitude.
        theEnemies[i].targetAlt = AssignNewAltitude();
        theEnemies[i].pass = 0;
      }
    }
    else if (theEnemies[i].dest.right < 0)
    {                   // If off left edge, wrap around to right side.
      OffsetRect(&theEnemies[i].dest, 640, 0);
      theEnemies[i].h = theEnemies[i].dest.left << 4;
      OffsetRect(&theEnemies[i].wasDest, 640, 0);
      theEnemies[i].pass++;
      if (theEnemies[i].pass > 2)
      {
        theEnemies[i].targetAlt = AssignNewAltitude();
        theEnemies[i].pass = 0;
      }
    }
                        // Throw a touch of friction into the mix.
    theEnemies[i].vVel -= theEnemies[i].vVel >> 4;
                        // Keep enemies from moving excessively fast.
    if (theEnemies[i].vVel > theEnemies[i].maxVVel)
      theEnemies[i].vVel = theEnemies[i].maxVVel;
    else if (theEnemies[i].vVel < -theEnemies[i].maxVVel)
      theEnemies[i].vVel = -theEnemies[i].maxVVel;

    CheckEnemyRoofCollision(i); // Check for lava/ceiling collisions.
    CheckEnemyPlatformHit(i);   // Check for platform collisions.
}

//------------------------------------------------------------  HandleWalkingEnemy()

// This is a brief mode for an enemy. When an enemy has hatched from an egg, it
// walks only for 8 game frames, at which point it takes off and flies for the rest
// of its life.

void HandleWalkingEnemy (short i)
{
    if (theEnemies[i].facingRight)   // If enemy facing right, walk to the right.
    {
      theEnemies[i].dest.left += 6; // Move enemy to right.
      theEnemies[i].dest.right += 6;
      switch (theEnemies[i].kind)   // Determine correct graphic for walking enemy.
      {
        case kOwl:
        theEnemies[i].srcNum = 1 - theEnemies[i].srcNum;
        break;
```

```
      case kWolf:
      theEnemies[i].srcNum = 9 - theEnemies[i].srcNum;
      break;

      case kJackal:
      theEnemies[i].srcNum = 17 - theEnemies[i].srcNum;
      break;
    }
    theEnemies[i].hVel = 6 << 4;
  }
  else                    // If enemy not facing right (left), walk to the left.
  {
    theEnemies[i].dest.left -= 6; // Move enemy to left.
    theEnemies[i].dest.right -= 6;
    switch (theEnemies[i].kind)   // Determine correct graphic for walking enemy.
    {
      case kOwl:
      theEnemies[i].srcNum = 5 - theEnemies[i].srcNum;
      break;

      case kWolf:
      theEnemies[i].srcNum = 13 - theEnemies[i].srcNum;
      break;

      case kJackal:
      theEnemies[i].srcNum = 21 - theEnemies[i].srcNum;
      break;
    }
    theEnemies[i].hVel = -6 << 4;
  }

  theEnemies[i].frame++;          // Increment number of frames it has walked for.
  if (theEnemies[i].frame >= 8)   // If over 8, enemy takes off and flies.
  {
    theEnemies[i].mode = kFlying; // Switch to flying mode.
    theEnemies[i].frame = 0;      // Reset "frame" variable.
    switch (theEnemies[i].kind)   // Determine correct graphic for flying enemy.
    {
      case kOwl:
      if (theEnemies[i].facingRight)
        theEnemies[i].srcNum = 12;
      else
        theEnemies[i].srcNum = 14;
      break;

      case kWolf:
      if (theEnemies[i].facingRight)
        theEnemies[i].srcNum = 16;
      else
        theEnemies[i].srcNum = 18;
      break;
```

```
        case kJackal:
        if (theEnemies[i].facingRight)
          theEnemies[i].srcNum = 20;
        else
          theEnemies[i].srcNum = 22;
        break;
      }
                          // Resize enemy bounds to a "flying" size.
    theEnemies[i].dest.left -= 8;
    theEnemies[i].dest.right += 8;
    theEnemies[i].dest.bottom = theEnemies[i].dest.top + 40;
    theEnemies[i].h = theEnemies[i].dest.left * 16;
    theEnemies[i].v = theEnemies[i].dest.top * 16;
  }
}

//-------------------------------------------------------- HandleSpawningEnemy()

// This is an enemy "rising out of a platform." Either an egg has just hatched
// or a brand new enemy has been introduced. Regardless, the sphinx is born.
// When the enemy is at its full height, it will begin to walk.

void HandleSpawningEnemy (short i)
{
  theEnemies[i].frame++;          // Advance timer.
  if (theEnemies[i].frame >= 48)  // If timer >= 48, enemy begins to walk.
  {
    theEnemies[i].mode = kWalking;
    theEnemies[i].frame = 0;

    switch (theEnemies[i].kind)   // Determine appropriate graphic.
    {
      case kOwl:
      if (theEnemies[i].facingRight)
        theEnemies[i].srcNum = 0;
      else
        theEnemies[i].srcNum = 2;
      break;

      case kWolf:
      if (theEnemies[i].facingRight)
        theEnemies[i].srcNum = 4;
      else
        theEnemies[i].srcNum = 6;
      break;

      case kJackal:
      if (theEnemies[i].facingRight)
        theEnemies[i].srcNum = 8;
      else
        theEnemies[i].srcNum = 10;
      break;
```

```
      }
    }
    else                // If not full height, use "timer" to determine height.
      theEnemies[i].dest.top = theEnemies[i].dest.bottom - theEnemies[i].frame;
}

//-------------------------------------------------------------  HandleFallingEnemy()

// A "falling" enemy is an airborne egg. The enemy was killed, turned into an egg,
// and the egg is in freefall. If the egg comes to rest, it will begin a countdown
// until it is hatched.

void HandleFallingEnemy (short i)
{                       // Take into account gravity -- accelerate egg down.
  theEnemies[i].vVel += kGravity;
                        // Don't allow velocities to skyrocket.
  if (theEnemies[i].vVel > theEnemies[i].maxVVel)
    theEnemies[i].vVel = theEnemies[i].maxVVel;
  else if (theEnemies[i].vVel < -theEnemies[i].maxVVel)
    theEnemies[i].vVel = -theEnemies[i].maxVVel;

  if (evenFrame)        // Apply friction on even frames (who knows).
  {                     // "Friction" is 1/32nd of the velocity.
    theEnemies[i].hVel -= (theEnemies[i].hVel >> 5);
    if ((theEnemies[i].hVel < 32) && (theEnemies[i].hVel > -32))
    {
      if (theEnemies[i].hVel > 0)
        theEnemies[i].hVel--;
      else if (theEnemies[i].hVel < 0)
        theEnemies[i].hVel++;
    }
  }
                        // Move egg horizontally.
  theEnemies[i].h += theEnemies[i].hVel;
  theEnemies[i].dest.left = theEnemies[i].h >> 4;
  theEnemies[i].dest.right = theEnemies[i].dest.left + 24;
                        // Move egg vertically.
  theEnemies[i].v += theEnemies[i].vVel;
  theEnemies[i].dest.top = theEnemies[i].v >> 4;
  theEnemies[i].dest.bottom = theEnemies[i].dest.top + 24;
                        // Check for wraparound.
  if (theEnemies[i].dest.left > 640)
  {
    OffsetRect(&theEnemies[i].dest, -640, 0);
    theEnemies[i].h = theEnemies[i].dest.left << 4;
    OffsetRect(&theEnemies[i].wasDest, -640, 0);
  }
  else if (theEnemies[i].dest.right < 0)
  {
    OffsetRect(&theEnemies[i].dest, 640, 0);
    theEnemies[i].h = theEnemies[i].dest.left << 4;
    OffsetRect(&theEnemies[i].wasDest, 640, 0);
```

```
    }

    CheckEnemyRoofCollision(i); // See if egg hit ceiling or lava.
    CheckEnemyPlatformHit(i);   // Handle platform hit (it is here it determines if
                                // egg has come to rest and should begin countdown).
}

//-------------------------------------------------------------  HandleEggEnemy()

// This is the "idle" egg mode. This is a static egg, sitting peacefully on
// a platform. Waiting patiently so it might hatch into a death-sphinx and
// slaughter the player.

void HandleEggEnemy (short i)
{
    short     center;

    theEnemies[i].frame--;            // Decrement the egg timer!
    if (theEnemies[i].frame < 24)     // When it falls below 24, egg starts shrinking.
    {                                 // Use "frame" to determine height of egg.
        theEnemies[i].dest.top = theEnemies[i].dest.bottom - theEnemies[i].frame;
        if (theEnemies[i].frame <= 0) // When the egg is completely flat (gone)
        {                             // then BOOM! a sphinx is spawned!
            theEnemies[i].frame = 0;
            PlayExternalSound(kSpawnSound, kSpawnPriority);
            center = (theEnemies[i].dest.left + theEnemies[i].dest.right) >> 1;
                                      // Resize enemy bounds to new "walking enemy" ➥
size.
            theEnemies[i].dest.left = center - 24;
            theEnemies[i].dest.right = center + 24;
            theEnemies[i].wasDest = theEnemies[i].dest;
            theEnemies[i].h = theEnemies[i].dest.left << 4;
            theEnemies[i].v = theEnemies[i].dest.top << 4;
                                      // Set up all other enemy variables.
            theEnemies[i].wasH = theEnemies[i].h;
            theEnemies[i].wasV = theEnemies[i].v;
            theEnemies[i].hVel = 0;
            theEnemies[i].vVel = 0;
            theEnemies[i].mode = kSpawning;
            theEnemies[i].kind++;
            if (theEnemies[i].kind > kJackal)
                theEnemies[i].kind = kJackal;
            SetEnemyAttributes(i);
        }
    }
}

//-------------------------------------------------------------  MoveEnemies()

// This is the "master" enemy function. It goes through all the enemies
// and calls the above functions depending upon an enemy's mode.
```

```
void MoveEnemies (void)
{
   short     i;

   doEnemyFlapSound = FALSE;   // Intially, assume no flap or scrape sounds.
   doEnemyScrapeSound = FALSE;
                               // Go through each enemy.
   for (i = 0; i < numEnemies; i++)
   {
      switch (theEnemies[i].mode)
      {                        // Handle enemy according to mode it is in.
         case kIdle:           // Enemy not born yet.
         HandleIdleEnemies(i);
         break;

         case kFlying:         // Enemy airborne.
         HandleFlyingEnemies(i);
         break;

         case kWalking:        // Enemy just born, walking off platform.
         HandleWalkingEnemy(i);
         break;

         case kSpawning:       // Enemy growing from a platform.
         HandleSpawningEnemy(i);
         break;

         case kFalling:        // Enemy is an egg in flight.
         HandleFallingEnemy(i);
         break;

         case kEggTimer:       // Enemy is a patient, idle, silent egg.
         HandleEggEnemy(i);
         break;

         case kDeadAndGone:    // Enemy no more -- gone for good this level.
         break;
      }
   }
                               // If any sounds were flagged, play them.
   if (doEnemyFlapSound)
      PlayExternalSound(kFlap2Sound, kFlap2Priority);
   if (doEnemyScrapeSound)
      PlayExternalSound(kScrape2Sound, kScrape2Priority);
                               // See if enough enemies were killed to advance to
                               // next level (wave).
   if ((deadEnemies >= numEnemiesThisLevel) && (countDownTimer == 0))
      countDownTimer = 30;
}
```

```
//-------------------------------------------------------------  InitHandLocation()

// This simply sets up the hand. Puts it deep in the lava (off bottom of screen).

void InitHandLocation (void)
{
   SetRect(&theHand.dest, 0, 0, 56, 57);
   OffsetRect(&theHand.dest, 48, 460);
}

//-------------------------------------------------------------  HandleHand()

// This is the hand "AI." The hand, like the sphinx enemies, has modes.

void HandleHand (void)
{
   Rect    whoCares;
   short     hDiff, vDiff, pull, speed;

   switch (theHand.mode)
   {
      case kLurking:      // Hand is down, waiting for player to stray near.
      if ((thePlayer.mode == kFlying) && (SectRect(&thePlayer.dest, &grabZone,
&whoCares)))
      {                   // If player flies near, hand begins to reach out.
         theHand.mode = kOutGrabeth;
         InitHandLocation();
      }
      break;

      case kOutGrabeth:   // Hand is either coming after or has a hold of player.
      case kClutching:
      if (SectRect(&thePlayer.dest, &grabZone, &whoCares))
      {                   // See if player in the "grab/clutch zone."
         hDiff = theHand.dest.left - thePlayer.dest.left;
         vDiff = theHand.dest.top - thePlayer.dest.top;
                           // Ah! Player caught. Move player to correct
                           // location relative to the hand (so the player
                           // appears to, in fact, be held).
         if (thePlayer.facingRight)
            hDiff -= 3;
         else
            hDiff -= 21;
         vDiff -= 29;
                           // How hard/fast the hand moves depends on level.
         speed = (levelOn >> 3) + 1;
         if (hDiff < 0)
         {
            theHand.dest.left += speed;
            theHand.dest.right += speed;
         }
         else if (hDiff > 0)
```

```
    {
      theHand.dest.left -= speed;
      theHand.dest.right -= speed;
    }
    if (vDiff < 0)
    {
      theHand.dest.top += speed;
      theHand.dest.bottom += speed;
    }
    else if (vDiff > 0)
    {
      theHand.dest.top -= speed;
      theHand.dest.bottom -= speed;
    }
                    // Determine absolute distance player is from hand.
    if (hDiff < 0)
      hDiff = -hDiff;
    if (vDiff < 0)
      vDiff = -vDiff;

    if ((hDiff < 8) && (vDiff < 8))
    {               // If player in the "hot zone," player is nabbed!
      theHand.mode = kClutching;
      thePlayer.clutched = TRUE;
                    // Player's movement is severely dampened.
      thePlayer.hVel = thePlayer.hVel >> 3;
      thePlayer.vVel = thePlayer.vVel >> 3;
                    // Hand pulls player down (strength is greater on
                    // higher levels).
      pull = level0n << 2;
      if (pull > 48) // Set an absolute limit on hand strength.
        pull = 48;
                    // Pull player down!
      thePlayer.vVel += pull;
      theHand.dest.top = thePlayer.dest.top + 29;
      theHand.dest.bottom = theHand.dest.top + 57;
      if (thePlayer.facingRight)
        theHand.dest.left = thePlayer.dest.left + 3;
      else
        theHand.dest.left = thePlayer.dest.left + 21;
      theHand.dest.right = theHand.dest.left + 58;
    }
    else            // If player not in "sweet spot," hand is seeking.
    {
      thePlayer.clutched = FALSE;
      theHand.mode = kOutGrabeth;
    }
  }
  else              // Player not even close to hand…
  {                 // Hand sinks back down into lava.
    theHand.dest.top++;
    theHand.dest.bottom++;
                    // When hand is off screen, hand resumes lurking.
```

```
        if (theHand.dest.top > 460)
          theHand.mode = kLurking;
        else
          theHand.mode = kOutGrabeth;
        thePlayer.clutched = FALSE;
      }
      break;
    }
}

//-------------------------------------------------------------- InitEye()

// This initializes all the eye's variables.

void InitEye (void)
{
  SetRect(&theEye.dest, 0, 0, 48, 31);
  OffsetRect(&theEye.dest, 296, 97);
  theEye.mode = kWaiting;
  theEye.frame = (numOwls + 2) * 720;
  theEye.srcNum = 0;
  theEye.opening = 1;
  theEye.killed = FALSE;
  theEye.entering = FALSE;
}

//-------------------------------------------------------------- KillOffEye()

// This function handles a "slain" eye!

void KillOffEye (void)
{
  if (theEye.mode == kStalking)
  {
    theEye.killed = TRUE;
    theEye.opening = 1;
    theEye.entering = FALSE;
    if (theEye.srcNum == 0)
      theEye.srcNum = 1;
  }
  else
    InitEye();
}

//-------------------------------------------------------------- HandleEye()

// But of course, the eye has modes as well. This function handles the eye
// depending upon the mode it is in.

void HandleEye (void)
{
  short     diffH, diffV, speed;
```

```
if (theEye.mode == kStalking) // Eye is alive!
{
   speed = (levelOn >> 4) + 1; // How fast it moves depends on level.
   if (speed > 3)
      speed = 3;
                                 // When eye appears or dies, it is stationary.
   if ((theEye.killed) || (theEye.entering))
   {
      speed = 0;
   }
   else if ((thePlayer.mode != kFlying) && (thePlayer.mode != kWalking))
   {
      diffH = theEye.dest.left - 296;
      diffV = theEye.dest.bottom - 128;
   }
   else
   {
      diffH = theEye.dest.left - thePlayer.dest.left;
      diffV = theEye.dest.bottom - thePlayer.dest.bottom;
   }
                                 // Find direction to player (no wraparound for eye).
   if (diffH > 0)
   {
      if (diffH < speed)
         theEye.dest.left -= diffH;
      else
         theEye.dest.left -= speed;
      theEye.dest.right = theEye.dest.left + 48;
   }
   else if (diffH < 0)
   {
      if (-diffH < speed)
         theEye.dest.left -= diffH;
      else
         theEye.dest.left += speed;
      theEye.dest.right = theEye.dest.left + 48;
   }
   if (diffV > 0)
   {
      if (diffV < speed)
         theEye.dest.bottom -= diffV;
      else
         theEye.dest.bottom -= speed;
      theEye.dest.top = theEye.dest.bottom - 31;
   }
   else if (diffV < 0)
   {
      if (-diffV < speed)
         theEye.dest.bottom -= diffV;
      else
         theEye.dest.bottom += speed;
      theEye.dest.top = theEye.dest.bottom - 31;
```

```
}

theEye.frame++;          // Increment eye frame (timer).
                         // Determine correct graphic for eye.
if (theEye.srcNum != 0)
{
   if (theEye.frame > 3) // "Eye-closing frame" holds for 3 frames.
   {
      theEye.frame = 0;
      theEye.srcNum += theEye.opening;
      if (theEye.srcNum > 3)
      {
         theEye.srcNum = 3;
         theEye.opening = -1;
         if (theEye.killed)
            InitEye();
      }
      else if (theEye.srcNum <= 0)
      {
         theEye.srcNum = 0;
         theEye.opening = 1;
         theEye.frame = 0;
         theEye.entering = FALSE;
      }
   }
}
else if (theEye.frame > 256)
{
   theEye.srcNum = 1;
   theEye.opening = 1;
   theEye.frame = 0;
}
                            // Get absolute distance from eye to player.
diffH = theEye.dest.left - thePlayer.dest.left;
diffV = theEye.dest.bottom - thePlayer.dest.bottom;
if (diffH < 0)
   diffH = -diffH;
if (diffV < 0)
   diffV = -diffV;
                            // See if player close enough to be killed!
if ((diffH < 16) && (diffV < 16) && (!theEye.entering) &&
     (!theEye.killed))      // Close enough to call it a kill.
{
   if (theEye.srcNum == 0)  // If eye was open, player is killed.
   {                        // Strike lightning (hit the player).
      if (lightningCount == 0)
      {
         lightH = thePlayer.dest.left + 24;
         lightV = thePlayer.dest.bottom - 24;
         lightningCount = 6; // Strike 6 times!
      }
                            // Player is smokin' bones!
```

```
            thePlayer.mode = kFalling;
            if (thePlayer.facingRight)
               thePlayer.srcNum = 8;
            else
               thePlayer.srcNum = 9;
            thePlayer.dest.bottom = thePlayer.dest.top + 37;
            PlayExternalSound(kBoom2Sound, kBoom2Priority);
         }
         else                        // If the eye was "blinking," IT was killed!
         {                           // Player killed the eye!
            if (lightningCount == 0)
            {                        // Strike the eye with lightning!
               lightH = theEye.dest.left + 24;
               lightV = theEye.dest.top + 16;
                                     // Hit 'er with 15 bolts!
               lightningCount = 15;
            }
            theScore += 2000L;       // A big 2000 points for killing the eye!
            UpdateScoreNumbers();    // Refresh score display.
            PlayExternalSound(kBonusSound, kBonusPriority);

            KillOffEye();            // Slay eye!
         }                           // Hey, anyone remember that giant eye from
      }                              // Johnny Socko and his Flying Robot?
   }                                 // As a kid, I thought that was cool!
   else if (theEye.frame > 0)        // Eye has not yet appeared, but waits, lurking!
   {
      theEye.frame--;                // Decrement eye timer.
      if (theEye.frame <= 0)         // When timer hits zero, eye appears!
      {
         theEye.mode = kStalking; // The eye is after the player!
         if (lightningCount == 0) // Strike lightning at eye!
         {
            lightH = theEye.dest.left + 24;
            lightV = theEye.dest.top + 16;
            lightningCount = 6;
         }
         theEye.srcNum = 3;
         theEye.opening = 1;
         theEye.entering = TRUE;
      }
   }
}

//-------------------------------------------------------  ResolveEnemyPlayerHit()

// Okay, a bounds test determined that the player and an enemy have collided.
// This function looks at the two and determines who wins or if it's a draw.

void ResolveEnemyPlayerHit (short i)
{
   short     wasVel, diff, h, v;
```

```
   if ((theEnemies[i].mode == kFalling) || (theEnemies[i].mode == kEggTimer))
   {                        // Okay, if the enemy is an egg…
      deadEnemies++;        // simple -- the enemy dies.

      theEnemies[i].mode = kDeadAndGone;
      theScore += 500L;     // Add that to our score!
      UpdateScoreNumbers();
      PlayExternalSound(kBonusSound, kBonusPriority);
      InitEnemy(i, TRUE);   // Reset the enemy (I guess you could say they're ➥
reincarnated).
   }
   else                     // Now, here's a real, live sphinx enemy.
   {                        // Get their difference in altitude.
      diff = (theEnemies[i].dest.top + 25) - (thePlayer.dest.top + 19);

      if (diff < -2)        // Player is bested.  :(
      {                     // Strike player with lightning.
         if (lightningCount == 0)
         {
            lightH = thePlayer.dest.left + 24;
            lightV = thePlayer.dest.bottom - 24;
            lightningCount = 6;
         }
                           // Player is bones.
         thePlayer.mode = kFalling;
         if (thePlayer.facingRight)
            thePlayer.srcNum = 8;
         else
            thePlayer.srcNum = 9;
         thePlayer.dest.bottom = thePlayer.dest.top + 37;
         PlayExternalSound(kBoom2Sound, kBoom2Priority);
      }
      else if (diff > 2)    // Yes!  Enemy is killed!
      {                     // Well… we can't kill an enemy who is spawning.
         if ((theEnemies[i].mode == kSpawning) && (theEnemies[i].frame < 16))
            return;
                           // Resize enemy bounds (use an egg bounds).
         h = (theEnemies[i].dest.left + theEnemies[i].dest.right) >> 1;
         if (theEnemies[i].mode == kSpawning)
            v = theEnemies[i].dest.bottom - 2;
         else
            v = (theEnemies[i].dest.top + theEnemies[i].dest.bottom) >> 1;
         theEnemies[i].dest.left = h - 12;
         theEnemies[i].dest.right = h + 12;
         if (theEnemies[i].mode == kSpawning)
            theEnemies[i].dest.top = v - 24;
         else
            theEnemies[i].dest.top = v - 12;
         theEnemies[i].dest.bottom = theEnemies[i].dest.top + 24;
         theEnemies[i].h = theEnemies[i].dest.left << 4;
         theEnemies[i].v = theEnemies[i].dest.top << 4;
```

```
                        // Enemy is a falling egg!
        theEnemies[i].mode = kFalling;
        theEnemies[i].wasDest = theEnemies[i].dest;
        theEnemies[i].wasH = theEnemies[i].h;
        theEnemies[i].wasV = theEnemies[i].v;
                        // Give player points based on enemy kind.
        switch (theEnemies[i].kind)
        {
           case kOwl:
           theScore += 500L;
           break;

           case kWolf:
           theScore += 1000L;
           break;

           case kJackal:
           theScore += 1500L;
           break;
        }
        UpdateScoreNumbers();
        PlayExternalSound(kBoom2Sound, kBoom2Priority);
     }
     else        // Rare case -- neither the player nor the enemy gets killed.
     {           // They'll bounce off one another.
        if (theEnemies[i].hVel > 0)
           theEnemies[i].facingRight = TRUE;
        else
           theEnemies[i].facingRight = FALSE;
        PlayExternalSound(kScreechSound, kScreechPriority);
     }

     wasVel = thePlayer.hVel;
     thePlayer.hVel = theEnemies[i].hVel;
     theEnemies[i].hVel = wasVel;
     wasVel = thePlayer.vVel;
     thePlayer.vVel = theEnemies[i].vVel;
     theEnemies[i].vVel = wasVel;
  }
}

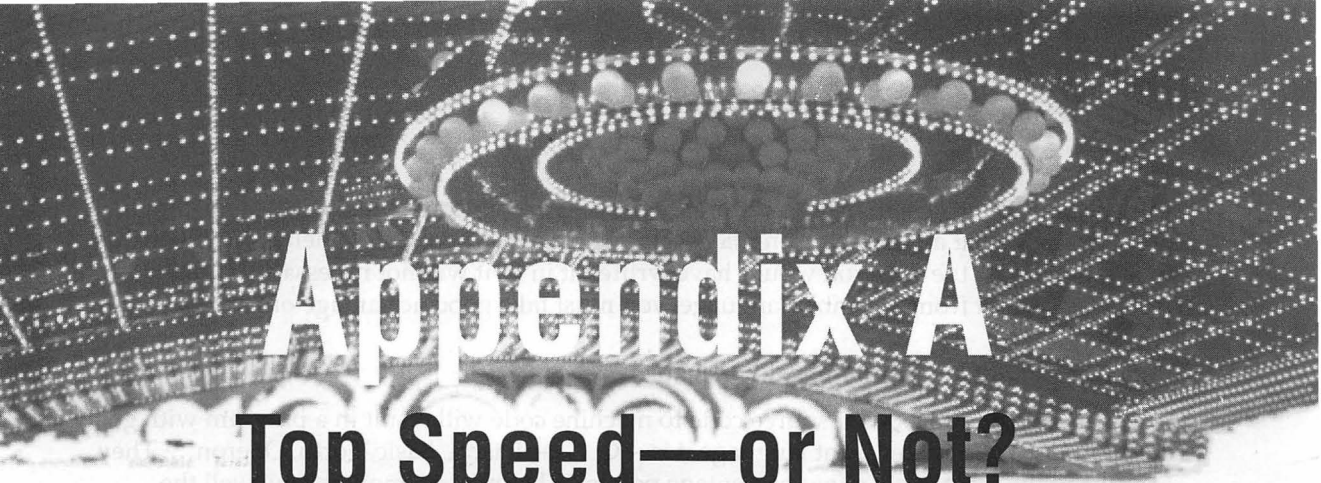
//-------------------------------------------------- CheckPlayerEnemyCollision()

// This is a simple "bounds test" for determining player/enemy collisions.

void CheckPlayerEnemyCollision (void)
{
  Rect   whoCares, playTest, wrapTest;
  short    i;

  playTest = thePlayer.dest;    // Make a copy of player's bounds.
```

```
  InsetRect(&playTest, 8, 8);    // Shrink it by 8 pixels all 'round.
  if (thePlayer.wrapping)        // Need to test 2 players if "wraparounding."
    wrapTest = thePlayer.wrap;
  InsetRect(&wrapTest, 8, 8);
                                 // Test all enemies.
  for (i = 0; i < numEnemies; i++)
  {                              // Ignore nonexistent enemies.
    if ((theEnemies[i].mode != kIdle) && (theEnemies[i].mode != kDeadAndGone))
    {                            // Simple bounds test.
      if (SectRect(&playTest, &theEnemies[i].dest, &whoCares))
      {                          // Call function to determine who wins (or tie).
        ResolveEnemyPlayerHit(i);
      }                          // If "wraparounding," test other rect.
      else if (thePlayer.wrapping)
      {
        if (SectRect(&wrapTest, &theEnemies[i].dest, &whoCares))
          ResolveEnemyPlayerHit(i);
      }
    }
  }
}
```



Appendix A
Top Speed—or Not?

A big problem for game programmers is how to make games fast enough. It seems as if we can never get enough processing power. The graphics chapters are very concerned about speed issues. In this appendix, we want to look into the question of speed optimization from a general point of view.

However, getting the maximum speed is not the only challenge. If you play some old game on a new computer, you will often find that it runs way too fast. This is another problem, one that is easily avoided—if you do it the right way.

Getting Top Speed

There are few types of programs for which execution speed is as important as games. In games you find the most extreme levels of optimization. Most top-notch games must have top-notch speed.

The big question is this: What slows your program down? That's what we'll discuss here.

Programming Language

You may sometimes hear that the choice of programming language is vital. Don't believe it. The quality of the compiler is much more important, and the quality of your code is even more important. Regarding the choice of language/compiler, there are really just three levels to bother with.

Assembly Language

Nothing can beat well-optimized assembly language. It is the lowest level and the highest detail you can ever get without taking the CPU apart. Not too long ago, many games were written in assembly language *only*. Nowadays, only the most critical parts tend to be written in assembly. In games, that usually means graphics routines, such as sprite blitters and 3-D rendering.

Even then, assembler will make a difference only if you can come up with an algorithm that is better than what a high-level language compiler can do. If you implement your algorithm in assembler just the way you would have written it in C, it will not necessarily run any faster. To benefit from assembly language, you must take good advantage of the low level.

Compiling Medium- to High-Level Languages

A system that compiles your source code to machine code will result in a program with good speed. It does not matter what language it is—C, C++, Pascal, Basic, Forth, Oberon.... They are all equally fast within a few percentage points. What matters more is how well the compiler optimizes the code—but even that is a minor issue nowadays. In most cases, all modern compilers will produce code that is almost as fast as assembly language.

Interpreters

An interpretive language, such as HyperTalk in HyperCard, is something different. It doesn't compile your source code to machine code, but either parses the text as it runs or compiles it to some intermediate form. This implies a major speed loss—expect tenfold or worse. However, many interpreters, HyperCard included, can call compiled code resources and thereby run at least those pieces at full speed.

Note
It's not your tools, it's what you do with them that matters.

Inner Loops

What counts is not just *what* you do, but *when* you do it. Concentrate on the things you do many times per second. Is there something that you do many times that could be done just once by moving it out of a loop?

All the following examples are of most interest when they're part of an inner loop, when used many times per frame. The pixel plotting example (see the section coming up, "Tables, Tables, Tables!") is perhaps the most obvious one.

Unrolling Loops

When something is done many times in a row, you can often win some speed by unrolling loops. That will save some of the time spent incrementing the loop variable and checking for completion. In the extreme case, you don't have any loop at all, but just do the task over and over, as many times it takes. However, it is usually enough to unroll a loop a few times, as in the following code.

The demo MoveData, which you can find on the CD-ROM in the App A - Top Speed—or Not? folder, demonstrates the importance of moving data in big chunks, a topic I will return to later. However, in this demo I use unrolled loops. Without unrolled loops, the code for moving a char at a time would have looked like this:

```
    memLoopLimit = kBlockSize / sizeof(char);
    timeBefore = TickCount();
    for (loop = 0; loop < kLoopNumber; loop++)
    {
      charSourcePtr = sourcePtr;
      charDestPtr = destinationPtr;
      for (memLoop = 0; memLoop < memLoopLimit; memLoop++)
      {
        *charDestPtr++ = *charSourcePtr++;
      }
    }
    timeAfter = TickCount();
    printf("Chars: %ld ticks for %ld copies.\n", timeAfter-timeBefore, ➥
(long)kLoopNumber);
```

However, when I unroll the loop 16 times, the loop looks like this:

```
    memLoopLimit = kBlockSize / sizeof(char) / 16;
    timeBefore = TickCount();
    for (loop = 0; loop < kLoopNumber; loop++)
    {
      charSourcePtr = sourcePtr;
      charDestPtr = destinationPtr;
      for (memLoop = 0; memLoop < memLoopLimit; memLoop++)
      {
        *charDestPtr++ = *charSourcePtr++;
        *charDestPtr++ = *charSourcePtr++;
        *charDestPtr++ = *charSourcePtr++;
        *charDestPtr++ = *charSourcePtr++;
        *charDestPtr++ = *charSourcePtr++;
        *charDestPtr++ = *charSourcePtr++;
        *charDestPtr++ = *charSourcePtr++;
        *charDestPtr++ = *charSourcePtr++;
        *charDestPtr++ = *charSourcePtr++;
        *charDestPtr++ = *charSourcePtr++;
        *charDestPtr++ = *charSourcePtr++;
        *charDestPtr++ = *charSourcePtr++;
        *charDestPtr++ = *charSourcePtr++;
        *charDestPtr++ = *charSourcePtr++;
        *charDestPtr++ = *charSourcePtr++;
        *charDestPtr++ = *charSourcePtr++;
      }
    }
    timeAfter = TickCount();
    printf("Chars: %ld ticks for %ld copies.\n", timeAfter-timeBefore, ➥
(long)kLoopNumber);
```

But, how much does the unrolled loop help to improve the speed? I compiled a new version of MoveData without unroll, as listed previously. Then I ran both that version and the unrolled one and compared the results for the character-by-character data moves. The times were as follows:

Power Mac 6100/60 (emulated):

- Without unrolled loop: 471 Ticks for 100 copies
- With unrolled loop: 182 Ticks for 100 copies

Power Mac 6100/60 (native):

- Without unrolled loop: 91 Ticks for 100 copies
- With unrolled loop: 73 Ticks for 100 copies

PowerBook 180c:

- Without unrolled loop: 310 Ticks for 100 copies
- With unrolled loop: 191 Ticks for 100 copies

As you can see, the unrolled loop makes a significant difference for 68K code, but not quite as much when running native on a Power Mac. Even though the difference may look small on the Power Mac (about 25 percent), the difference is definitely big enough to bother with if the unrolled loop is performed many times per second.

Fixed Point Arithmetics

In chapter 7, I described how to use fixed point numbers. On 68K Macs, this is of extreme importance. On Power Macs, you always have an FPU built-in (at least so far), so you wouldn't lose much time, if any. If you make a Power Mac-only game, you can consider floating point. That will usually (but not always) be even faster than fixed point, due to having calculations made in parallel and having more registers available. If you make a game for 68K, you should *never* use floating point—do all you can to go the fixed point route.

Tables, Tables, Tables!

And Noah said to the animals: "Go forth and multiply!"

But the adders came to Noah and said: "But we cannot multiply, we are adders."

Then Noah went out and chopped down some trees and built a table of logs. And he showed it to the adders and said:

"Look, I have built you a table of logs! Now you adders can multiply!"

When I made the first version of Ingemar's Skiing Game, I used trigonometric functions (`sin()` and `cos()`) to split the skier's movement into a horizontal and vertical component. I decided that those few floating point calculations per frame wouldn't hurt.

That first version was a lot slower than the final game, partially since it allowed background processing. This caused a low frame rate, so the trigonometric functions were indeed no major problem. However, when I refined the game and took out the background processing, I found that the trigonometric functions took a significant amount of time. On the slower Macs, it was the major bottleneck.

There is one simple way out of this problem. When initializing, build a table that holds all values that you may need. The table should preferably hold integer or fixed point values, so you need no floating point calculations at all during the game.

As an example, here is a snippet that generates tables in 8-bit fixed point for sine and cosine.

```
#include <Math.h>

#define PI      3.1415926535
#define kAngles 360

long gSineTable[kAngles];
long gCosineTable[kAngles];

void InitializeSineTables()
{
   long angle;

   for (angle=0; angle<kAngles; angle++)
   {
      gSineTable[angle]=sin(angle * 2 * PI / kAngles)*256;
      gCosineTable[angle]=cos(angle *  2 * PI/kAngles)*256;
   }
} /*InitializeSineTables()*/
```

Another example concerns graphics. When drawing directly to the screen or to a pixmap, you can find the beginning of a row by taking the `baseAddr` and adding the row number times the `rowBytes`. A considerable speed-up can be achieved by avoiding the multiplication and making a table look-up instead. Yes, even simple operations such as integer multiplication and division are slow enough to be worth avoiding!

In the PixelPlotTest demo, I use a table for the fastest case. The table, named `rowStartTable`, holds the addresses to the beginning of each row in a pixmap (see figure A.1).

```
   long rowStartTable[kSizeV+1];
```

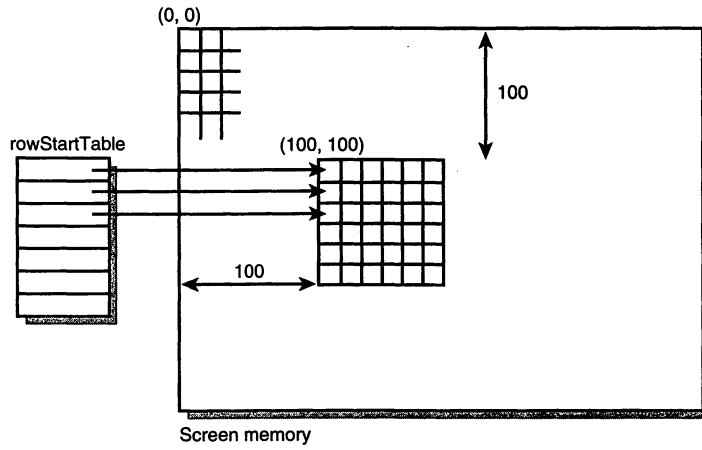


Figure A.1 *Each element in the array* `rowStartTable` *points to the first pixel in a row in the drawing area.*

This table is initialized as follows. The added 100s are there to reflect the window position used in the demo.

```
for ( v = 0 ; v <= kSizeV ; v++)
   rowStartTable[v] = baseAddr + rowBytes * (v + 100) + 100;
```

Without the table, I would find the address of a pixel as:

```
pixAddr = (Ptr) (baseAddr + rowBytes * (v + 100) + h + 100);
```

...while with the table, the address is

```
pixAddr = (Ptr)(rowStartTable[v] + h);
```

The latter case is a lot faster. The big difference is the multiplication—trust me!

Avoiding Function Call and Toolbox Trap Overhead

When you write the most time-critical parts of your program (i.e., inner loops that will be executed many times per frame), you can do considerable optimization by avoiding the overhead of Toolbox traps and even of the calling of functions.

There are often four ways to perform a task:

- Call a Toolbox routine to do it.
- Call the Toolbox routine using its trap address, avoiding the trap overhead.
- Write a function that does it.
- Write a macro that does it.

The last option is reasonable only for very simple tasks. I am rather reluctant to recommend macros, since it's easy to mess up a program completely if you use them too much. However, when used with some care, macros are useful for speed optimization.

Writing your own procedure is sometimes possible and sometimes not. Often you'll find that your replacements are slower than the Toolbox routines, despite your supposedly optimized code. The ROMs of the Mac are pretty good.

If the functionality you want is in the Toolbox, you should usually just call it by the usual trap mechanism, but be aware that those calls cost some time! You can get the trap address with `NGetTrapAddress()` and call that directly.

I have written a demo to show the difference in speed between these techniques. The chosen Toolbox trap is the `SetPt()` trap, which assigns two values to the fields of a point. This is the simplest Toolbox call I know, so it is easy to replace, and the impact of the overhead is big.

The program sends its output to the standard SIOUX console. The constant `kLoopNumber` tells how many times each loop should be executed. In each loop, a `SetPt()` is performed ten times.

```
#include <stdio.h>

#define kLoopNumber 30000
```

`SetPtMacro()` is a macro that is equivalent to `SetPt()`.

```
#define SetPtMacro(p,hh,vv) p.h=hh;p.v=vv
```

`MySetPt()` is a routine that is equivalent to `SetPt()`.

```
void MySetPt(Point *thePoint, short h, short v)
{
   thePoint->h = h;
   thePoint->v = v;
} /*MySetPt()*/
```

I declare the `SetPtPtr` type to be able to declare a pointer to the `SetPt()` routine, so I can call `SetPt()` without the trap overhead.

```
typedef pascal void (*SetPtPtr)(Point *, short, short);
```

Then we come to the main program...

```
void main(void)
{
```

The variables `timeBefore` and `timeAfter` are used with `TickCount()` for measuring the time for each loop. The variable `loop` is the loop variable, and `myPoint` is the point I keep setting over and over again.

```
long    timeBefore,timeAfter;
short   loop;
Point   myPoint;
```

The pointer `setPtAddress` is the pointer for the trap address.

```
SetPtPtr  setPtAddress;

InitToolbox();
```

The first test is calling `SetPt()` the standard way:

```
timeBefore = TickCount();
for (loop = 0; loop < kLoopNumber; loop++)
{
   SetPt(&myPoint, loop, loop);
```

It's repeated ten times...

```
}
timeAfter = TickCount();
printf("Trap call: %ld ticks for %ld calls.\n", timeAfter-timeBefore, ➥
(long)kLoopNumber*10);
```

The second test is calling `SetPt()` directly through the trap address:

```
setPtAddress = (SetPtPtr) NGetTrapAddress(_SetPt,ToolTrap);
//  _SetPt = $A880;
timeBefore = TickCount();
for (loop = 0; loop < kLoopNumber; loop++)
{
   (*setPtAddress)(&myPoint, loop, loop);
```

It's repeated ten times...

```
}
timeAfter = TickCount();
printf("Trap direct: %ld ticks for %ld calls.\n", timeAfter-timeBefore, ➥
(long)kLoopNumber*10);
```

The third test is calling my replacement function `MySetPt()`:

```
timeBefore = TickCount();
for (loop = 0; loop < kLoopNumber; loop++)
```

```
    {
      MySetPt(&myPoint, loop, loop);
```

It's repeated ten times...

```
    }
    timeAfter = TickCount();
    printf("Function call: %ld ticks for %ld calls.\n", timeAfter-timeBefore, ➥
(long)kLoopNumber*10);
```

Finally we come to the fourth test, a macro replacement:

```
    timeBefore = TickCount();
    for (loop = 0; loop < kLoopNumber; loop++)
    {
      SetPtMacro(myPoint, loop, loop);
```

It's repeated ten times...

```
    }
    timeAfter = TickCount();
    printf("Macro: %ld ticks for %ld calls.\n", timeAfter-timeBefore, ➥
(long)kLoopNumber*10);

} /*main()*/
```

The result depends on the computer on which you run the program. Here are the results on a selection of Macintosh models.

Power Mac 6100/60 (emulated):

- Trap call: 28 Ticks for 300,000 calls
- Trap direct: 26 Ticks for 300,000 calls
- Function call: 35 Ticks for 300,000 calls
- Macro: 7 Ticks for 300,000 calls

Power Mac 6100/60 (native):

- Trap call: 12 Ticks for 300,000 calls
- Trap direct: 259 Ticks for 300,000 calls*
- Function call: 2 Ticks for 300,000 calls
- Macro: 1 Tick for 300,000 calls

* Note: The code above for this part of the experiment does not work as native PPC code. You have to use `CallUniversalProc()`, which is what adds the overhead.

PowerBook 180c:

- Trap call: 200 Ticks for 300,000 calls
- Trap direct: 41 Ticks for 300,000 calls
- Function call: 52 Ticks for 300,000 calls
- Macro: 7 Ticks for 300,000 calls

Mac SE:

- Trap call: 926 Ticks for 300,000 calls
- Trap direct: 299 Ticks for 300,000 calls
- Function call: 475 Ticks for 300,000 calls
- Macro: 77 Ticks for 300,000 calls

However, when doing tests like this, you have to make sure that all the operations are really carried out! When making meaningless operations, the compiler's optimizer may notice and skip the instructions. Entire loops are sometimes skipped when the optimizer finds out that they do nothing meaningful.

Calling Nonnative Calls on Power Macs

The first program that I managed to compile on a Power Mac was my sprite demo OffscreenToys. When it ran, I was shocked—it was slow. Not unreasonably slow, but far from the speed I had expected. It ran at the same speed as my PowerBook 180c (33 MHz 68030), using native PPC code!

Ridiculous, right? I must have done something stupid, right? The reason was simple: I was using `PlotCIcon()` to draw icons, and that call is not native yet. Hence, my "native" program was spending most of its time in 68K code.

When I ran the variation of my little demo that uses `CopyBits()` instead of `PlotCIcon()`, I suddenly got the expected speed. `CopyBits()` is native so it gave me Power Mac speed.

Note
On Power Macs, expect many Toolbox calls to be emulated and try alternatives.

Moving Data in Big Chunks

When you move large amounts of data—which usually means blitting graphics between off-screen images or to the screen—you should use large chunks of data, at least as large as the width of the data bus. On a 68K, that means moving one longword at a time. On the Power Mac, you can move data as `doubles`, using the 64-bit floating point registers.

On the CD-ROM, you'll find the program MoveData, which allocates some memory, two buffers with 100,000 bytes in each, and then copies the contents of one buffer to the other. This is done by moving 1 byte at a time (char), a word at a time (short), a longword at a time (long), or by calling the Toolbox routine `BlockMove()`. The loops were unrolled for giving all cases (except `BlockMove()`) the same overhead. Here are the results.

Power Mac 6100/60 (emulated):

- Chars: 182 Ticks for 100 copies
- Shorts: 99 Ticks for 100 copies
- Longs: 65 Ticks for 100 copies
- `BlockMove()`: 21 Ticks for 100 copies

Power Mac 6100/60 (native):

- Chars: 73 Ticks for 100 copies
- Shorts: 42 Ticks for 100 copies
- Longs: 31 Ticks for 100 copies
- `BlockMove()`: 26 Ticks for 100 copies

PowerBook 180c:

- Chars: 191 Ticks for 100 copies
- Shorts: 107 Ticks for 100 copies
- Longs: 65 Ticks for 100 copies
- `BlockMove()`: 60 Ticks for 100 copies

The conclusion is that it makes a substantial difference to move data a longword at a time rather than a byte at a time—but you don't easily beat `BlockMove()`, at least not for moving blocks this big.

Putting in a Speed Limit

Once you get the maximum speed out of your game and it runs fine on your own Mac, don't leave it there just yet. What happens on a high-end Mac, or on the low-cost Macs being released next month? Perhaps your game will suddenly run so fast that it is unplayable.

Some old games still run on new Macs but are no longer playable because of the lack of a speed limit. In some games, the programmer limits speed by running a `for` loop some number of times. Even if you let the player choose how many times that loop should be running, you have not solved the problem. Someday, probably sooner than you think, computers will be so fast that even the maximum delay you allow is ridiculously fast.

There is a Toolbox call named `Delay()` that will stop execution a number of "Ticks" (1/60th of a second). It is a better solution, but might make your game unnecessarily slow on Macs slower than your own.

You can also call `WaitNextEvent()` with a sleep value that is appropriate for the frame rate you want. But this will lead to unknown delays and is not to be recommended for smooth animations.

Setting a Speed Limit with `TickCount()`

A simple but generally satisfactory solution is to delay the execution of the game so that each frame takes a constant time, or more if necessary. The Toolbox call `TickCount()` will deliver the number of Ticks since the Mac started. Even though this is not a 100 percent accurate measure, it is good enough for our purposes.

Let's say that we want each frame to take three Ticks (1/20th of a second, giving us 20 frames/second). First, we get the current time at the beginning of our game loop. Then we process one frame of animation. Once that is done, we can inspect the current time again. If the time is more than the starting time plus three Ticks, we need no delay. If it's less, we call `TickCount()` repeatedly until the time reaches the initial time plus three.

In SpriteEngine this is implemented as follows:

```
while (!Button())
{
   startTime = TickCount();
   RunSpriteEngine();
   while (TickCount() < startTime + kFrameTime);
}
```

Making games this way is simple. You move objects frame by frame, processing all objects with full control over how much they can move. The code will be simple and there will be few nasty surprises.

The major drawback is that if you cannot fulfill the desired frame rate, the game will slow down.

Scheduling Events after the Current Time

An alternative to the previous method is to take the time, and depending on the time determine what should happen. If an object is to move one pixel per Tick, you just move it as many pixels as there are Ticks that have passed since the last frame, the last time we had time to process it.

This can be implemented using `TickCount()` by just getting the current count. You can also use the Time Manager, even by using interrupts, but I'll leave that to you. The Time Manager will give you a much higher-resolution timer than `TickCount()`, but that hardly matters when making games.

This method has its weaknesses where the previous method has strengths, and vice versa.

The animation will never feel slow. If the Mac is slow, the frame rate goes down, but the action will be as fast, just a bit less smooth. When the game is run on a faster Mac than was available when the game was designed, it's ready to use the potential of a higher frame rate. You don't design it for 15 or 20 fps, you design it for the top frame rate that the computer can deliver.

This method is by far more complicated to code. Objects may move anywhere between zero and... how much? You just don't know. What if two objects are on a collision course? What if we get an unexpected delay, perhaps caused by network traffic or whatever? The objects might take a big leap and pass straight through each other. You need to take precautions, perhaps setting a maximum time difference so we get a certain slowdown if the Mac slows down.

The demos SE Timer and SE No Timer on the CD-ROM (in the SpriteEngine folder) demonstrate the difference. Both demos have 12 rectangular sprites, moved with fixed point arithmetics and with gravity.

In SE Timer each sprite has a `lastTime` field which tells when it was last moved. In a real game, I would integrate this in the sprite engine. In the demo, each sprite takes care of itself.

The difference between `TickCount()` and `lastTime` is the number of times the sprite should be moved. This is a simple way to make each sprite move to the position it should have for the current time, but the simplest solution is not necessarily the best solution. I could make border checks for every move and even collision detections, or I could move directly to the "right" position by adding `(TickCount()-lastTime) * speed` to the `fixedPointPosition`. I leave these options open to you.

The `MoveSprite()` procedure in SE Timer looks like this:

```
void MoveSprite(SpritePtr theSprite)
{
   long thisTime;

   thisTime = TickCount();

   while (theSprite->lastTime < thisTime)
   {
      theSprite->lastTime ++;
      theSprite->speed.v++; // Simple gravity
      theSprite->fixedPointPosition.h += theSprite->speed.h;
      theSprite->fixedPointPosition.v += theSprite->speed.v;
      theSprite->position.h = theSprite->fixedPointPosition.h >> 4;
      theSprite->position.v = theSprite->fixedPointPosition.v >> 4;
   };

   KeepOnScreenFixed(theSprite);
} /*MoveSprite()*/
```

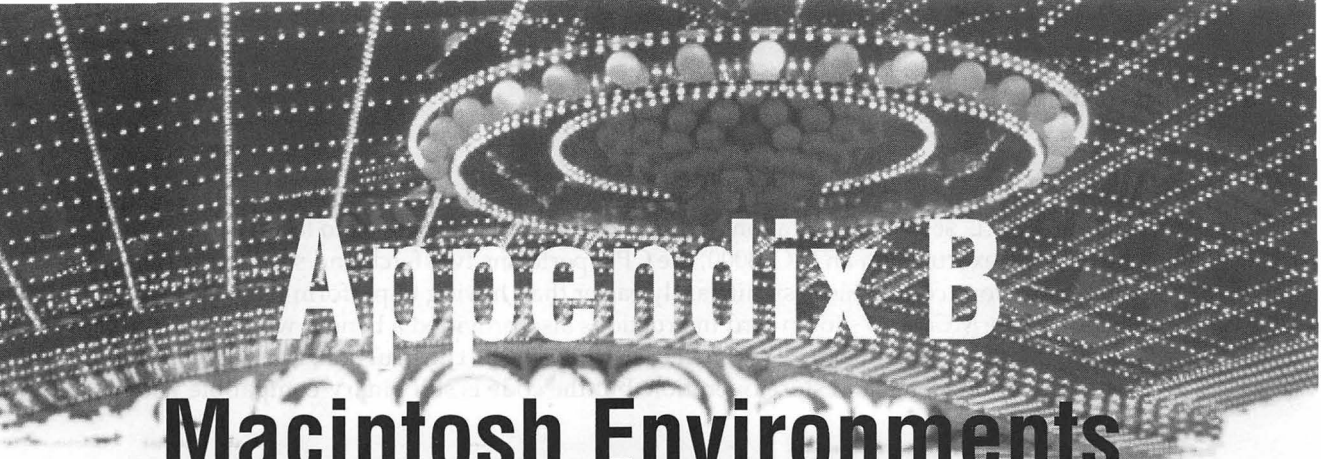
In SE No Timer there is no such timing. Each sprite is moved once per frame, and that's it. Its `MoveSprite()` routine is straightforward:

```
void MoveSprite(SpritePtr theSprite)
{
   theSprite->speed.v++; // Simple gravity
   theSprite->fixedPointPosition.h += theSprite->speed.h;
   theSprite->fixedPointPosition.v += theSprite->speed.v;
   theSprite->position.h = theSprite->fixedPointPosition.h >> 4;
   theSprite->position.v = theSprite->fixedPointPosition.v >> 4;
   KeepOnScreenFixed(theSprite);
} /*MoveSprite()*/
```

You've seen that one before in other demos.

If you run the demos on a fast Mac, you might not notice any difference, but on a slower Mac, SE No Timer will be slower—but you may also notice that SE Timer sometimes makes some errors. Sprites bounce off each other in ways that don't look nice, and they sometimes lock into each other or pass right through each other.

I generally prefer the simple solution. It gives no nasty surprises, just a little slowness on old Macs. However, for programs like texture-mapping 3-D games, even many fairly fast Macs can't produce the optimal frame rate, so the frame rate must adapt to the processing power available. In such cases, you should consider scheduling of some kind, similar to what I did in SE Timer or with Time Manager routines.



Appendix B
Macintosh Environments

The large variety of Macintosh computer configurations is daunting—Apple has released an average of eight Macintosh models per year since 1984. Fortunately, there are more similarities than differences between Macintosh computers, and the reference material in this appendix should help you out.

First, for those unfamiliar with the Mac's variety of microprocessors, we'll include a brief overview of the Motorola M68000 line of microprocessors, and then a brief overview of the PowerPC microprocessors used in Power Macintosh computers. Then we'll include a list of all Macintosh models, along with their essential characteristics and any special hardware benefits you may be able to make use of in your games. Finally, we'll provide a short laundry list of the changes you can expect if you want your game to be compatible with Macintosh computers running System 6.

The M68000 Microprocessors

Following is a brief overview of the Motorola M68000 line of processors. The processors' official names are the MC68000, MC68020, MC68030, MC68040, and MC68LC040 processors. Two floating point math coprocessors, called the MC68881 and MC68882, are featured in many Macs. The line as a whole is termed the M68000 line.

You can receive complete official documentation about these processors from Motorola at Motorola Literature Distribution, P.O. Box 20912, Phoenix, Arizona, 85036. The people at Literature Distribution are very friendly, and if you are nice to them, you may be able to request complete sets of manuals for free! There are *User's Manuals* available as well as *Programmer's Reference Manuals*.

MC68000

The MC68000 was the microprocessor in the original Macintosh. It is considered a 16-bit processor because it has a 16-bit data path, meaning it can fetch or store 16 bits at a time to or from memory. However, the MC68000 is more advanced and forward-looking than many other 16-bit processors. It has a 24-bit addressing space, meaning it can address up to 4 GB of memory. Its eight data and eight address registers are 32-bit registers rather than 16-bit

registers. Many of its instructions' operands are for 32-bit longwords, so the assembly instruction `MOVE.L $0, $4` moves a longword (4 bytes) from location `$0` to location `$4`. When this instruction executes on an MC68000, the CPU performs two fetch and store operations, transparent to your code. This is significantly easier than having to perform two `MOVE` instructions. The MC68000's longword instructions also provided a benefit when the MC68020 processor was introduced: When executing a `MOVE.L` instruction, the MC68020 has to perform only one fetch and store operation, but the code is still binary-compatible with the MC68000.

The MC68000 is featured in Macintosh computers at 8 MHz and 16 MHz.

The MC68000 is featured in the original Macintosh, the Mac 512KE, Macintosh Plus, Macintosh SE, Macintosh Classic, Macintosh Portable, and Macintosh PowerBook 100.

MC68020

The MC68020 is a generation ahead of the MC68000. It's very similar to the MC68000 with several additions:

- 32-bit addressing space (supported in System 7.0 and later)
- 32-bit data path
- 256 byte instruction cache
- Ability to add a math coprocessor for inline execution of floating point instructions
- Ability to add an MC68851 PMMU for demand-paged memory management to support virtual memory in the computer's OS
- Several additional instructions
- Higher clock speeds

The most significant performance improvements on the MC68020 are due to the 256 byte instruction cache, the faster clock speed of the processor, and the newly widened 32-bit data path.

The MC68020 is completely compatible with code that executes on an MC68000 processor, with the important exception of self-modifying code. The MC68020's 256 byte instruction cache is read-only and malfunctions if your code writes to memory that is currently present in the instruction cache. If you write to code that is present in the cache, the write will be performed to memory but not to the cache. The result is usually a crash. When the Macintosh II was introduced, many programs crashed immediately because they used self-modifying code for speed.

If you want to write self-modifying code, you should call the `MakeDataExecutable()` Toolbox routine to avoid cache problems.

The MC68851 PMMU allowed the OS to support virtual memory with the assistance of hardware, and the PMMU was indeed an option on the original Macintosh II computer.

Unfortunately, the addition of a PMMU to an MC68020 machine slows down the computer; one cycle is added to *every single instruction* executed by the CPU, as the PMMU performs its address lookups.

The MC68020 is used in Macintosh computers only at 16 MHz.

The MC68020 is used in the Macintosh II and the Macintosh LC computers.

MC68030

The MC68030 processor is an incremental improvement over the MC68020. It is about the same as an MC68020 with several additions:

- PMMU brought on-board the chip
- 256 byte data cache added
- Faster clock speeds

The addition of the PMMU ensures that the computer can run with virtual memory. The one-cycle-per-instruction penalty of the MC68851 PMMU is removed in the MC68030, since no external PMMU is necessary.

The 256 byte data cache on the MC68030 speeds up the processor about ten percent over an MC68020 at the same clock speed, according to an old Apple press release.

The MC68030 is used in various Macintosh computers at speeds of 16 MHz, 20 MHz, 25 MHz, 33 MHz, and 40 MHz. It can be found in third-party accelerator cards at 50 MHz.

The MC68030 is used in many Macintosh models. It's used in the Macintosh IIx, IIcx, IIci, IIfx, IIsi, IIvi, IIvx, LC II, LC III, LC 550, SE/30, Classic II, and Color Classic; in the PowerBook 140, 145, 145B, 150, 160, 165, 165c, 170, 180, and 180c; in the PowerBook Duo 210, 230, 250, 270c; and in the Performa 200, 400, 405, 410, 430, 450, 460, 466, 467, 550, 560, 600, and 600CD.

MC68040

The MC68040 processor was billed by Motorola as a generation beyond the MC68030. It is much faster than the MC68030 and has several additions:

- 4K instruction cache
- 4K data cache
- FPU brought onboard (mostly)
- New `MOVE16` instruction
- "Clock-doubled" internal bus speed

The larger instruction and data caches of the MC68040 can greatly help the speed of program execution, especially if your code is optimized with the 4K caches in mind. The instruction

cache is still read-only and still crashes your program if you make use of self-modifying code within the cache area.

When the Quadra 700 and 900 were introduced, programmers who used self-modifying code were burned *again*—some programmers who encountered the self-modifying code problem on the Macintosh II merely tweaked their code to be sure to self-modify code only if it were just outside the 256 byte boundary of the MC68020's cache. When the MC68040 increased the instruction cache to 4K, the self-modifiers' code crashed again as the modified code fell within the instruction cache again.

Use the `MakeDataExecutable()` Toolbox call to avoid cache problems with self-modifying code.

The FPU brought onboard is much faster than the MC68882 math coprocessor that's used in many MC68030 Macs. The MC68040's FPU is more limited than the MC68882; the MC68040's FPU contains a subset of the most commonly used FPU instructions and does not contain the MC68882's trigonometric or logarithmic functions in hardware. Instead, FPU instructions in these categories throw an interrupt, and a software library optimized for the MC68040 calculates and returns the results. Motorola claimed that trig functions on the MC68040 execute two to three times as fast as on the MC68882, even though they are calculated with software instead of on the chip.

The MC68040 includes a welcome new instruction called `MOVE16`, which moves 16 bytes with one instruction. The `BlockMove()` function makes use of `MOVE16` if appropriate. Sixteen bytes, incidentally, is the size of one of the "cache lines" in the MC68040's data cache. See the *MC68040 User's Manual* for more details.

The MC68040 is included in Macintosh computers at clock speeds of 25 MHz, 33 MHz, and 40 MHz. It can be found in certain accelerator boards at 50 MHz.

Since the MC68040's internal bus runs at twice the speed of its external bus, at one point Apple attempted to denote certain Macintosh models as featuring a "66/33 MHz" or "50/25 MHz" processor. The result of this unique nomenclature was probably only confusion.

The MC68040 is included in the Macintosh Centris 650; in the Quadra 610, 630, 650, 660AV, 700, 800, 840AV, 900, and 950; in the PowerBook 520, 520c, 540, and 540c; and in the LC 520.

MC68LC040

The MC68LC040 is an MC68040 processor with no FPU onboard. This chip is cheaper than an MC68040 chip, has a smaller die size, and consumes less power. Attempting to execute FPU instructions on an MC68LC040 will cause an exception and lead to a crash.

The MC68LC040 is used in Macintosh computers at speeds of 20 MHz, 25 MHz, and 33 MHz.

The MC68LC040 is used in the Macintosh Centris 610; in the LC 475, LC 575, and LC 630; in the Quadra 605; in the Performa 475, 476, 575, 577, 578, and 630; in the PowerBook 520, 520c, 540, and 540c; and in the PowerBook Duo 280c and 280.

MC68881 and MC68882 Floating Point Math Coprocessors

Many Macintosh models include either an MC68881 or MC68882 floating point math coprocessor. The only difference between the two is that the MC68882 is about 40 percent faster than the MC68881. Apple only used the MC68881 in the original Macintosh II computer.

The math coprocessor chip is not necessary—or available—if you are running on an MC68040 processor, which includes the math coprocessor onboard. Also, you can't have an MC68881 or MC68882 math coprocessor on a system with an MC68LC040 system, even though the MC68LC040 has no math coprocessor—the user's solution is to upgrade the processor to an MC68040.

The MC6888x processors have eight 80-bit floating point data registers. They can do work in single precision (32-bit representation), double precision (64-bit representation), or extended precision (80-bit representation).

In practice, games rarely make use of floating point math. Historically, this has been because floating point math has always been much slower than the less precise integer math. This may change as the PowerPC, with its excellent floating point performance, becomes more dominant.

PowerPC Microprocessors

In 1992, Apple, IBM, and Motorola announced the creation of a new alliance which was aimed at taking away market share from the mighty PC-compatible market dominated by Intel. They were to jointly design and create a new RISC processor called the PowerPC, based on IBM's RISC architecture called POWER. This new processor was to be the foundation of a new generation of computing for many operating systems, and Apple's PowerPC-based Macintosh computer was to be called Power Macintosh.

Power Macintosh computers are able to execute M68000-based software (i.e., the entire installed base up to that point) by means of a software *emulator*. The emulator simply reads each instruction in an M68000 program and executes several PowerPC RISC instructions that perform the same function. Performance is pretty good, but not astonishing: A 60 MHz Power Macintosh executes emulated code at the speed of a fast MC68030 Macintosh. Apple is working on a second-generation emulator that makes use of dynamic recompilation to get much faster speeds out of M68000 code.

Performance of the emulator is especially important because a large percentage of Macintosh System software code still exists as 68K object code. Less than half of the Macintosh System software code is native under System 7.5.1.

For more information about Power Macintosh, see the volumes of *New Inside Macintosh* that deal with the PowerPC chip.

You can receive complete official documentation about these processors from Motorola or from IBM. Motorola can be contacted at Motorola Literature Distribution, P.O. Box 20912, Phoenix, Arizona, 85036. IBM can be contacted at Mail Stop A25/862-1, PowerPC Marketing, 1000 River Street, Essex Junction, Vermont 05452, or at 1-800-POWERPC.

PowerPC 601

The PowerPC 601 is a RISC microprocessor with 32 32-bit general-purpose registers and 32 64-bit floating point registers. It has a 32-bit address bus but a 64-bit data bus, meaning you can move 64 bits to or from memory per instruction. It has a 32K unified instruction and data cache which is critical to the 601's speed. The "pipelining" of instructions is also critical to the 601's performance. The 601 contains three execution units which operate in parallel: one integer unit, one branch processing unit, and one floating point unit. It is thus possible for the 601 to execute three instructions *per cycle* if the instructions are pipelined properly such that all the execution units may have something to do on a given cycle.

One interesting feature of the 601 is its inclusion of several registers and instructions that are part of the original POWER specification. These registers and instructions are *not* part of the PowerPC specification, but the 601 includes them in an attempt to be a "bridge" between POWER code and PowerPC code. (Not that this "bridge" affects any of us in the Macintosh world.) There are actually a few compilers out there for the PowerPC 601 chip (such as the current version of gcc) that can generate POWER instructions. You must be sure to use a "POWER-clean" compiler to ensure future compatibility with Power Macintosh computers. (Metrowerks CodeWarrior is POWER-clean.)

The PowerPC 601 is used in the Power Macintosh 6100/60, 6100/66, 7100/66, 7100/80, 8100/80, 8100/100, and 8100/110 computers—and several of the new Macintosh clones. Apple's numbering scheme for the Power Macintosh computers is as follows: The number before the slash refers to the case design, while the number after the slash is the processor's speed in megahertz.

Note

It is possible to buy a $100 kit from many mail-order outfits that replaces a Power Mac's clock chip with a faster one—for instance, you can boost a 6100/60 from 60 MHz to 80 MHz.

PowerPC 603

The PowerPC 603 was intended to be a lower-power version of the 601 with a smaller die size and lower performance, mostly due to a smaller amount of cache memory on board. The 603 has a 32-bit address bus and either a 64-bit or a 32-bit data bus.

The 603 has an 8K instruction cache and an 8K data cache. It includes five execution units: an integer unit, a floating point unit, a branch processing unit, a load/store unit, and a system register unit.

The first Macintosh to ship with a 603 processor was the Power Macintosh 5200/75, which ran a 75 MHz 603. It is expected that Macintosh computers that were originally conceived based on the 603 are to run with a 603e chip instead.

Apple engineers were reportedly dissatisfied with the speed of the PowerPC 603 chip when running Apple's MC68000 emulator. Apple asked for a new chip design based on the 603 and got the 603e.

PowerPC 603e

The PowerPC 603e is very similar to the PowerPC 603, but features a 16K instruction cache and a 16K data cache.

Initial tests indicate that the 603e comes very close to the speeds attained by the higher-cost PowerPC 601 chip. It is expected that the 603e will be used across Apple's line of PowerBook and Performa computers.

PowerPC 604

The PowerPC 604 is a step beyond the 601. It has six execution units which can operate in parallel: three integer units, one floating point unit, one branch processing unit, and one load/store unit. The chip includes a 16K instruction cache and a 16K data cache. It has a 32-bit address bus and a 64-bit data bus.

The PowerPC 604 processor debuted on the Power Macintosh 9500 in June 1995.

Daystar stirred up some excitement early in 1995 when it announced it was going to make a Macintosh clone based on four PowerPC 604 processors, running a new multiprocessing API that replaces the Thread Manager. It is expected that software written to this new API will just blaze.

PowerPC 620

As of the date of publication, the PowerPC 620 has not yet shipped, so it's difficult to predict its speed or eventual uses. It is expected to feature a 64-bit address bus and lots of execution units, and it will probably be priced pretty high and exist as a CPU for servers for a while.

Macintosh Computers

Note

If you need detailed information about every Macintosh, including the processor, the amount of DRAM and VRAM included and expandable to, and the slots, you should get the latest copy of the "Apple Compatibility Chart." You can request copies of the chart at `S.LINE.ORDER` *on AppleLink. From the Internet, this address would be* `s.line.order@applelink.apple.com`.

Compact Macintosh Models

The original Macintosh computer won design awards for its compact, all-in-one "friendly" case which was such a contrast to the oblong metal boxes typical of IBM computers. Some loved the all-in-one case, and some thought it made the Macintosh look too much like a toy. In any case, Apple introduced many models based on variants of the original case from 1984 to 1992.

All compact Macintosh models include a built-in monitor. Only the Color Classic features a color monitor; the other models are black-and-white only. (Note that several models in the compact case include Color QuickDraw, but you have to attach an external color monitor to really make use of it.)

Macintosh (128K)

Processor:

- 8 MHz 68000

Advantages:

- None

Disadvantages:

- Black-and-white only

Notes:

- Outdated—most software doesn't run on the original Macintosh
- Upgrade to 512K RAM was available, resulting in the "Fat Mac"
- 64K ROMs
- 400K floppy drive

- No SCSI or HFS
- Built-in, 512 x 342 black-and-white display

Macintosh 512KE

Processor:

- 8 MHz 68000

Advantages:

- None

Disadvantages:

- Black-and-white only
- Outdated

Notes:

- Introduced along with Mac Plus; the 512KE, unlike the Mac Plus, does not include a SCSI port nor SIMMs for upgradeable RAM
- 128K of ROM
- First Mac with HFS file system
- Macintosh 512KE should actually be able to run System 7.5.1 if provided with a 4 MB RAM upgrade
- Built-in, 512 x 342 black-and-white display

Macintosh Plus

Processor:

- 8 MHz 68000

Advantages:

- None

Disadvantages:

- Outdated

Notes:

- First Macintosh with a SCSI port
- Built-in, 512 x 342 black-and-white display

Macintosh SE

Processor:

- 8 MHz 68000

Advantages:

- Slot for external monitor or Ethernet

Disadvantages:

- Outdated

Notes:

- Introduced simultaneously with Macintosh II
- First compact Macintosh with a slot
- 256K ROM
- Still no Color QuickDraw
- Built-in, 512 x 342 black-and-white display

Macintosh Classic

Processor:

- 8 MHz 68000

Advantages:

- None

Disadvantages:

- Outdated

Notes:

- Introduced simultaneously with Macintosh LC and IIsi as one of the new "Low Cost" Macintoshes
- Almost identical to Macintosh SE, but without a slot
- Interesting "ROM Disk" enabling you to boot from a volume in ROM with System 6.0.6 (a System version never otherwise released)
- Built-in, 512 x 342 black-and-white display

Macintosh SE/30

Processor:

- 16 MHz 68030
- 16 MHz 68882

Firsts:

- First compact Mac with 68030 processor

Advantages:

- Slot included for external video

Disadvantages:

- Slow by today's standards
- ROMs not 32-bit clean

Notes:

- First early machine that had a pretty high probability of having multiple monitors
- Same as the Mac IIx but with only the one SE/30 slot
- Built-in, 512 x 342 black-and-white display

Macintosh Classic II

Processor:

- 16 MHz 68030

Advantages:

- None

Disadvantages:

- Outdated

Notes:

- Built-in, 512 x 342 black-and-white display

Macintosh Color Classic

Processor:

- 16 MHz 68030

Advantages:

- None

Disadvantages:

- Slow by today's standards

Notes:

- Built-in, 512 x 384 256-color display

Macintosh II Series

The Macintosh II line of computers was made from 1987 to 1992. They range from the huge technological step forward represented by the Macintosh II to the less-than-impressive-in-technology-but-created-to-fill-a-market-niche Macintosh IIvx.

Macintosh II computers all ran 68020 or 68030 microprocessors. They had no built-in monitors.

Macintosh II

Processor:

- 16 MHz 68020
- 16 MHz 68881
- 68851 PMMU (optional)

Firsts:

- First color Macintosh
- First Mac with slots
- First Mac with 68020
- First Mac with math coprocessor (68881)
- First Mac with PMMU (68851, optional)

Advantages:

- Six NuBus slots—could therefore have up to six monitors at once

Disadvantages:

- Nearly the lowest end of color Macs
- Installing a 68851 slows down the Mac II even more
- ROMs not 32-bit clean

Macintosh IIx

Processor:

- 16 MHz 68030
- 16 MHz 68882

Firsts:

- First Mac with 68030
- First Mac with 68882 math coprocessor

Advantages:

- First Mac to ship with 1.44 MB floppy drive

Disadvantages:

- Slow by today's standards
- ROMs not 32-bit clean

Notes:

- Uncommon

Macintosh IIcx

Processor:

- 16 MHz 68030
- 16 MHz 68882

Firsts:

- First color Mac in a more compact desktop case

Advantages:

- Easy to add RAM

Disadvantages:

- Slow by today's standards
- ROMs not 32-bit clean

Notes:

- Same as the Mac IIx but with only three NuBus slots

Macintosh IIsi

Processor:

- 20 MHz 68030

Advantages:

- Built-in video

Disadvantages:

- Slow by today's standards

Notes:

- Supported either an SE/30 slot or a NuBus slot

Macintosh IIci

Processor:

- 25 MHz 68030
- 25 MHz 68882

Firsts:

- First Mac whose processor ran at a speed above the minimum available
- First color Mac with built-in video
- First Mac with L2 cache (via plug-in card)
- First Mac with 32-bit clean ROMs
- First Mac with built-in video output; it supported up to 256 colors on a 640 x 480 display

Advantages:

- Built-in video probably faster than NuBus video

Disadvantages:

- Running in 256 colors slowed down main processor

Macintosh IIvi

Processor:

- 25 MHz 68030 (but only a 16 MHz data bus)

Advantages:

- None

Disadvantages:

- Data bus slowed down processor

Notes:

- Sold only in European market

Macintosh IIvx

Processor:

- 33 MHz 68030 (but only a 16 MHz data bus)

Advantages:

- None

Disadvantages:

- Data bus slows down system

Macintosh IIfx

Processor:

- 40 MHz 68030
- 40 MHz 68882

Firsts:

- First Mac with IOPs

Advantages:

- IOPs present for faster serial operations
- Multiplexed RAM for faster memory accesses

Disadvantages:

- Serial ports incompatible with some applications—software fix available
- Special SCSI terminator required when using external SCSI devices

Notes:

- The Macintosh IIfx contained two 6502 processors on the motherboard functioning as "IOPs" (i/o processors). They handled I/O to and from the serial ports, the ADB ports, and the floppy drive. This freed up the CPU and sped it up a bit, but led to incompatibilities with some software, and the result was a Serial Switch control panel that let the user choose between Faster and More Compatible.
- The Macintosh IIfx also supported SCSI DMA, but only the A/UX operating system ever took advantage of this hardware feature. SCSI Manager 4.3, the first SCSI Manager to support SCSI DMA, does not support the IIfx's SCSI DMA!

Macintosh Quadra and Centris Computers

Macintosh Quadra computers were introduced in 1991 with the Quadra 700 and 900, and the last Quadra was introduced in 1995 with the consumer-end Quadra 630. The name "Quadra" comes from the MC68040 processor used in the machines.

Three Macintosh Centris computers were introduced. They were intended to be "middle-range" computers priced below the Quadra line, but it was decided this was confusing to the consumer and the Centris line was soon merged into the Quadra line.

All Quadras and Centrises feature a Motorola 68040 or 68LC040 processor. None of them have a built-in monitor. Initially, the Quadras were high-end computers, but toward the end of the life cycle of the Quadra, some low-cost computers were introduced, including the bargain Quadra 605.

Macintosh Quadra 700, Quadra 900

Processor:

- 25 MHz 68040

Firsts (700 and 900):

- First Macs with 68040 processor
- First Macs with built-in video supporting "Millions" of colors and monitors larger than a 640 x 480 resolution
- First Macs to support multisync monitor display

Advantages:

- Faster and more versatile built-in video than previous Macs
- 68040 contains built-in FPU

Disadvantages:

- None

Macintosh Centris 650, Quadra 610

Processor:

- 25 MHz 68040

Advantages:

- Faster and more versatile built-in video than previous Macs
- 68040 contains built-in FPU

Disadvantages:

- None

Macintosh Quadra 605

Processor:

- 25 MHz 68040

Advantages:

- Faster and more versatile built-in video than previous Macs
- 68040 contains built-in FPU

Disadvantages:

- No NuBus slots

Macintosh Quadra 650, Quadra 800

Processor:

- 33 MHz 68040

Advantages:

- 68040 contains built-in FPU

Disadvantages:

- None

Macintosh Quadra 950

Processor:

- 33 MHz 68040

Advantages:

- Contains IOPs for serial ports
- 68040 contains built-in FPU

Disadvantages:

- See "Advantages"

Notes:

- See "Quadra 900"—only difference is that CPU runs at 33 MHz
- See "Macintosh IIfx" for notes about IOPs

Macintosh Centris 610

Processor:

- 20 MHz 68LC040

Advantages:

- None

Disadvantages:

- Slowest 68040 Macintosh
- No math coprocessor

Macintosh Centris 660AV

Processors:

- 25 MHz 68040
- AT&T 3210 DSP

Firsts:

- First Macintosh with a DSP on the motherboard
- First Macintosh with composite and SVHS video in and out

Advantages:

- AT&T 56001 DSP on board
- 68040 contains built-in FPU

Disadvantages:

- 660AV and 840AV were only Macs to include this DSP

Macintosh Quadra 840AV

Processors:

- 40 MHz 68040
- AT&T 3210 DSP

Advantages:

- Fastest 68040 Macintosh (at 40 MHz)
- AT&T 56001 DSP on board
- 68040 contains built-in FPU

Disadvantages:

- 660AV and 840AV were only Macs to include this DSP

Macintosh Quadra 630

Processor:

- 33 MHz 68040

Advantages:

- Contains "Valkyrie" video chip
- 68040 contains built-in FPU

Disadvantages:

- Relatively slow video bus

Macintosh LC Computers

The original "Macintosh LC" computer was introduced in 1991 as the first low-cost color Macintosh. Its performance was lower than computers in the Macintosh II line with similar characteristics, but hey, it was cheap!

The term "LC" has been retained; at first it was used as a term for consumer-level computers, but when the Macintosh Performa line was introduced (see the following section), the LC became a term for computers aimed at the education market.

All Macintosh LC computers do support color. Most require an external monitor, while some include a 13" Triniton monitor and are all-in-one units.

Note

One note about the LC and LC II computers: Apple introduced its 512 x 384 color monitor along with the Macintosh LC as a lower-cost option for consumers to get color. This monitor is frequently cited as the bane of game developers. At the time, the LC reached more consumers in the home—meaning game-playing *consumers—than any other color Macintosh, and an unknowable high percentage of these consumers had purchased the less expensive 12" monitor. So we had to write games based around the lame 512 x 384 dimensions, which is why many older games run in small-looking windows on your 640 x 480 color monitor.*

Macintosh LC

Processor:

- 16 MHz 68020 (with 16-bit data bus)

Advantages:

- None

Disadvantages:

- Slowest color Macintosh available; this is the minimum baseline

Macintosh LC II

Processor:

- 16 MHz 68030 (with 16-bit data bus)

Advantages:

- None

Disadvantages:

- Just about as slow as the Macintosh LC

Notes:

- The LC II was fitted with a 68030 not to speed up the machine, but to let it run virtual memory

Macintosh LC III

Processor:

- 25 MHz 68030

Advantages:

- None

Disadvantages:

- None

Notes:

- A little faster than a Mac IIci without a cache card

Macintosh LC 475

Processor:

- 25 MHz 68LC040

Advantages:

- None

Disadvantages:

- None

Macintosh LC 520

Processor:

- 25 MHz 68LC040

Advantages:

- None

Disadvantages:

- None

Macintosh LC 550

Processor:

- 33 MHz 68030

Advantages:

- None

Disadvantages:

- None

Macintosh LC 575

Processor:

- 33 MHz 68LC040

Advantages:

- None

Disadvantages:

- None

Macintosh LC 630

Processor:

- 33 MHz 68LC040

Advantages:

- Contains "Valkyrie" video chip

Disadvantages:

- Relatively slow video bus

Notes:

- Identical to Quadra 630 but no FPU onboard the chip

Macintosh PowerBook (and Portable) Series

Apple's first portable computer was rather late and slow in coming. The Macintosh Portable was introduced in 1989. It was a well-designed computer, and several decisions that had seemingly been made in the user's interest led to the machine's ridicule. The Portable was extremely heavy, and initially didn't even include a backlit display. This, added to the machine's price tag, led to its popular ridicule.

Apple's second attempt to crack the portable market was a smash hit. The PowerBook has been one of the top-selling portable computers every year since its introduction, and it's the *only* computer to have if you work in Hollywood, dude.

PowerBook Duo computers are lighter than other PowerBooks, lack an internal floppy drive, and have a docking mechanism that enables them to connect their internal bus to an expansion bus inside a docking station. This expansion bus can include a math coprocessor, NuBus slots, and SCSI devices.

Macintosh Portable

Processor:

- 16 MHz 68000

Advantages:

- None

Disadvantages:

- Outdated

Macintosh PowerBook 100

Processor:

- 16 MHz 68000

Advantages:

- None

Disadvantages:

- No internal floppy drive
- Outdated

Notes:

- Built-in 640 x 400 black-and-white display

Macintosh PowerBook 140

Processor:

- 16 MHz 68030

Advantages:

- None

Disadvantages:

- Color output only with a kludgey SCSI monitor connector

Notes:

- Built-in 640 x 400 black-and-white display

Macintosh PowerBook 145, 145B

Processor:

- 25 MHz 68030

Advantages:

- None

Disadvantages:

- None

Notes:

- Built-in 640 x 400 black-and-white display

Macintosh PowerBook 150

Processor:

- 33 MHz 68030

Advantages:

- None

Disadvantages:

- None

Notes:

- Built-in 640 x 400 black-and-white display

Macintosh PowerBook 170

Processor:

- 25 MHz 68030
- 25 MHz 68882

Advantages:

- None

Disadvantages:

- Color output only with a kludgey SCSI monitor connector

Notes:

- Built-in 640 x 400 black-and-white display

Macintosh PowerBook 160

Processor:

- 25 MHz 68030

Advantages:

- None

Disadvantages:

- None

Notes:

- Built-in 640 x 400 16-gray display

Macintosh PowerBook 165

Processor:

- 33 MHz 68030

Advantages:

- None

Disadvantages:

- None

Notes:

- Built-in 640 x 400 16-gray display

Macintosh PowerBook 165c

Processor:

- 33 MHz 68030
- 33 MHz 68882

Advantages:

- None

Disadvantages:

- None

Notes:

- Built-in 640 x 400 256-color display

Macintosh PowerBook 180

Processor:

- 33 MHz 68030
- 33 MHz 68882

Advantages:

- None

Disadvantages:

- None

Notes:

- Built-in 640 x 400 16-gray display

Macintosh PowerBook 180c

Processor:

- 33 MHz 68030
- 33 MHz 68882

Advantages:

- None

Disadvantages:

- None

Notes:

- Built-in 640 x 480 256-color display

Macintosh PowerBook 520

Processor:

- 25 MHz 68LC040

Advantages:

- None

Disadvantages:

- None

Notes:

- Built-in 640 x 480 16-gray display

Macintosh PowerBook 520c

Processor:

- 25 MHz 68LC040

Advantages:

- None

Disadvantages:

- None

Notes:

- Built-in 640 x 480 256-color display

Macintosh PowerBook 540

Processor:

- 33 MHz 68LC040

Advantages:

- None

Disadvantages:

- None

Notes:

- Built-in 640 x 480 64-gray display

Macintosh PowerBook 540c

Processor:

- 33 MHz 68LC040

Advantages:

- None

Disadvantages:

- None

Notes:

- Built-in 640 x 480 256-color display

Macintosh PowerBook Duo 210

Processor:

- 25 MHz 68030

Advantages:

- None

Disadvantages:

- None

Notes:

- Built-in 640 x 400 16-gray display

Macintosh PowerBook Duo 230

Processor:

- 33 MHz 68030

Advantages:

- None

Disadvantages:

- None

Notes:

- Built-in 640 x 400 16-gray display

Macintosh PowerBook Duo 250

Processor:

- 33 MHz 68030

Advantages:

- None

Disadvantages:

- None

Notes:

- Built-in 640 x 400 16-gray display

Macintosh PowerBook Duo 270c

Processor:

- 33 MHz 68030
- 33 MHz 68882

Advantages:

- None

Disadvantages:

- None

Notes:

- Built-in 640 x 480 256-color display

Macintosh PowerBook Duo 280

Processor:

- 33 MHz 68LC040

Advantages:

- None

Disadvantages:

- None

Notes:

- Built-in 640 x 400 16-gray display

Macintosh PowerBook Duo 280c

Processor:

- 33 MHz 68LC040

Advantages:

- None

Disadvantages:

- None

Notes:

- Built-in 640 x 480 256-color display

Macintosh Performa Series

Apple created the low-cost Macintosh Performa computers in a bid to dramatically increase sales on machines with low profit margins. Performas are sold in mass market and retail stores such as Sears and even Price Club/Costco. Performas are sold as bundles with a monitor and some software (including games!), all included in the deal at one price.

Macintosh Performa models are actually other Macintosh computers with a different name stickered on the front of the computer. (For instance, the Macintosh Performa 200 is exactly the same as a Macintosh Classic II.) Certain Performa models actually differ from each other only in the size of the internal hard disk. For this reason, we won't list the Performas here. See the "Apple Compatibility Chart" for more information.

Power Macintosh Computers

Apple's Power Macintosh computers all use a PowerPC RISC chip for the CPU. They all include an M68000 emulator in ROM for support of "old" Macintosh programs. The physical case design of these machines started with cases based on designs that were made for Macintosh Quadra computers... and who knows where Apple will lead us next.

Power Macintosh 6100/60, 6100/60AV, 6100/66, 6100/66AV

Advantages:

- PowerPC 601 processor running at 60 or 66 MHz
- 64-bit wide data bus
- Cheap clock chip upgrade lets many users run at 80 or 84 MHz
- Many users make use of SIMM socket available for cache RAM

Disadvantages:

- None

Power Macintosh 7100/66, 7100/66AV, 7100/80, 7100/80AV

Advantages:

- PowerPC 601 processor running at 66 or 80 MHz
- 64-bit data bus
- Many users make use of SIMM socket available for cache RAM

Disadvantages:

- None

Notes:

- 7100 series includes both a DRAM-based video connector that supports up to "Thousands" of colors and a VRAM-based video connector. The AV models have 2 MB VRAM on board, and strangely, AV card's data bus is only 32 bits wide. Non-AV models have a VRAM card instead with 1 MB VRAM, expandable to 2, and a 64-bit-wide data bus.

Power Macintosh 8100/80, 8100/80AV, 8100/100, 8100/100AV, 8100/110

Advantages:

- PowerPC 601 processor running at 80, 100, or 110 MHz
- 64-bit data bus
- SIMM socket filled with cache RAM at factory

Disadvantages:

- None

Notes:

- 8100/80 and 8100/100 include both a DRAM-based video connector that supports up to "Thousands" of colors and a VRAM-based video connector. The AV models have 2 MB VRAM on board, and strangely, the AV card's data bus is only 32 bits wide. Non-AV models have a VRAM card instead with 1 MB VRAM, expandable to 2, and a 64-bit-wide data bus.
- 8100/110 did not officially support AV card.

Power Macintosh 5200/75

Advantages:

- PowerPC 603 processor running at 75 MHz

Disadvantages:

- PowerPC 603 processor is slow compared to 601; Apple attempted to make performance comparable to 6100/60 by using 75 MHz clock speed and L2 cache

Notes:

- Includes "Valkyrie" video chip

System 6 Compatibility

Macintosh System software has experienced many, many revisions since System 1.0. The current version of System software as of this writing is System 7.5.1.

Throughout this book, we have assumed you're running System 7.0 or later. Most users do, too. However, many, many old Macs out there still run some variety of System 6. For ease of programming, most game authors today assume the user has System 7.0. But you may wish to appeal to a broader audience—although most people with System 6 don't use their Macs very much, it doesn't *hurt* your product to support System 6. It may indeed take a little more development time to support System 6. It's up to you whether to support System 6 users or not.

These days, the tendency is to require System 7.0 or later. There are a few real reasons to do so from the development side, but most of them are minor. The biggest reason to require System 7 is really just compatibility testing. It would be a drag to have to go through a round of System 6 testing on all machines on which you intend to run.

Plus, after you have lived in System 7 for a few months, going back to System 6 is like going back to a command prompt. Doh!

Making Your Program Not Crash under System 6

System 7.0 added many features to the Macintosh that we now take for granted. The most obvious was a greatly updated version of the Finder, but there were many subtle additions. If your users run System 6 (or, the horror, previous System versions), their Macs will most likely crash if you attempt to use Toolbox routines that make use of features listed in the bullet points below. To prevent a crash, you need to call `Gestalt()` and determine whether your user is running System 7.0 or later.

The following features are some of the additions to the Toolbox that were made in System 7 that you may need to care about. There are others that no game has ever used and never will use (for instance, Publish and Subscribe). Apple's *Inside Macintosh*, Volume VI, published in 1991 and still available in many places but no longer in print, is the comprehensive guide to System 7 changes. Here, however, are the more important items:

- 32-bit addressing
- Virtual memory
- Personal File Sharing
- Sound Manager 3.0 (an extension in System 7.0 through 7.1.1, integrated in System 7.5 and later)

- Alias files
- New FSSpec file manager calls and data structures
- Automatic folder detection (i.e., Preferences folder within System folder)
- Balloon Help
- PPC Toolbox
- TrueType fonts
- Apple Events
- Communications Toolbox integrated (INIT not needed anymore)

If you intend to make use of any of these items, your application needs to require System 7.0 or later. To provide crash protection, one of the very first things you should do after initializing the Toolbox is to determine the System version and exit if the poor user is still running System 4.2.

Be sure to do this in all your programs! You never know what weird environments are out there, and it's best to gracefully exit rather than face the wrath of Weird User.



Index

Symbols

2-D elastic collisions, 479-489
2-D space games, 65-66
3-D action games, 71-72
3-D games (collision detection), 470
68000 (Mac), 63

A

abstract action games, 75-76
acceleration (game entities), 421-422
action games (game entities)
  combining behaviors, 443-445
  evading, 436-438
  hunting, 434-436
  programmed behaviors, 441-443
  random movement, 438-441
  sleeping, 441
ActivatePalette( ) function, 103
adding backgrounds, 195-198
additive color, 88
AddPoints( ) function, 286
addRectToUpdate( ) function, 181-182, 189
addresses (AppleTalk), 553-617
AddSoundToQ ( ) function, 38
adjustRect( ) function, 182
adventure games, 72-74
aesthetics
  artwork, 712-713
  audience tastes, 728-730
  interface, 714-725
  Macintosh, 717
  menus, 715-717
  packaging, 727-728
  sound, 725-726
AIFF (file format), 356-364
alerts (resource files), 253
alignment, blitter routines (PowerPC), 741-742
allocating memory (porting functions), 634
AnimateEntry( ) function, 106
AnimatePalette( ) function, 107
animation, 2-6
  backgrounds, 121-144
    adding, 195-198
    restoring, 136
  frames, 4
  HelloWorld code, 13
  monitors, refreshing, 4-5
  palettes, 106-108
  refresh rate, 5-6
  speed, 2-4
  Sprite Animation Toolkit (SAT), 467
  sprites, 121
    Demo code, 140-143
    drawing, 138
  Ticks, 4
APDA (Apple Programmer's and Developer's Association), 641
Apple Events, 279
  Dungeon 3, 238-241
AppleTalk, 548-585
  addresses, 553
  asynch parameters, 555
  ATP (AppleTalk Transaction Protocol), 566-608
  DDP (Datagram Delivery Protocol), 578-585

extended networks, 551
messages, 565-585
MPP driver, 552
Name Binding Protocol, 555-565
nodes, 549
self-sending, 554
socket numbers, 550
appletalkingStartup( ) function, 552
application interface, 291-292
arrays (rectangles), 189
artificial intelligence, *see* game entities
artwork
aesthetics, 712-713
graphics, 683-686
ray-tracing, 712
ASoundIsPlaying( ) function, 35
aspect ratio (graphics), 690-691
assembly language
CPUs, 670-672
speed, 797-798
asynch parameters (AppleTalk), 555-617
asynchronous sound, 32
ATP (AppleTalk Transaction Protocol), 566-608
ATPReceiveCompletionC( ) function, 576
ATPReceiveCompletionRoutine ASM( ) function, 573
attributes
box object (QuickDraw 3), 514
QuickDraw 3, 494
audience tastes, 728-730
audio, *see* sound

B

backfacing style objects (QuickDraw 3), 510-511
backgrounds, 121-144
adding, 195-198
restoring, 136
Balloon Help (resource files), 248-250
Barris, Robert, 703
baud rates (serial communications), 588
behaviors (game entities)
combining, 443-445
programming, 441-443
bitmaps, 3
color, 17
black-and-white, 6
blits (block transfers), 110
blitter, optimizing routines (PowerPC), 732-742
blitToScreen( ) function, 138, 179
BlockMove( ) function, 345
BNDL (resource files), 250
board games, 69-70
game entities, 445-466
deterministic, 446-463
Tic-Tac-Toe, 448-463
bouncing (collision handling), 471-474
2-D elastic, 479-489
elastic, 477-479
large and small objects, 475-489
plastic, 476
realism, 474-489
box object (QuickDraw 3), 512-513
attributes, 514
defining, 514-515
referencing, 515-516
brightness model (color), 92-96
buffers
graphics, 108-109
off-screen graphics, 684
Button( ) function, 269
bytes
CPUs, 672-676
porting code, 656-657

C

C language (porting), 637
callbacks
Hollywood Sound API, 362-363
MGWSound.c, 38-39
Sound Manager, 344-346
camera object (QuickDraw 3), 501-504
card games, 70-71
Castles: Siege and Conquest (game), 703
CD-ROMs, 664
compressing data, 665-666
reading, 664-665
CenterAlert( ) function, 20
centering windows, 297-298
channels (Sound Manager), 331
CheckKey( ) function, 268
cicn (graphics format), 256, 260
Classic (Mac), 63
clConnAlloc( ) function, 603
clConnAllow( ) function, 605
clConnClose( ) function, 606
clConnDispose( ) function, 606
clConnIdle( ) function, 606
clConnOpen( ) function, 605
clConnSetProp( ) function, 604
clear color pixels, 150-158
clearUpdate( ) function, 181
clIdle( ) function, 606
client-server communications (Outland), 607-608
clipped sprites, 155, 169
CloseDownSound( ) function, 16, 33
CloseMovieFile( ) function, 373
closing
Outland connections, 606
QuickTime movies, 373-377
CLUT (Color Look-Up Tables), 96-99
clut (graphics formats), 256
code
Color Picker, 93-95
Dungeon, 58-59

Dungeon 3, 241-242
Dungeon E, 411-416
Glypha III, 757-795
HelloWorld code, 6-30
animating, 13
contants, 8-10
PICT resources, 9-10
routines, 11-16
variables, 10-11
HelloWorld2 code
color, 16-23
error checking, 21-23
MGWGraphics.c, 21
MGWUtilities.c, 18-21
pixmaps, 21
HelloWorld3.c code, 23-25
routines, 23-25
HelloWorld4, 25-26
HelloWorld5, 26-28
HelloWorld6, 28-30
HelloWorld6.1, 30-39
matching to game needs, 632
Maze, 405-410
porting, 643-655
data, 656-657
development platform, 650-651
graphics, 647, 650
MIDI, 649
original language, 650
original platform, 646-650
sound, 648
speed, 651-653
System Software, 653-655
Tic-Tac-Toe, 450
collision detection, 466-470
3-D games, 470
grids, 469-470
image space collisions, 470
sprites, 466-470
collision handling, 470-489
bouncing, 471-474
2-D elastic, 479-489
elastic, 477-479
large and small objects, 475-476
plastic, 476
realism, 474-489
color
additive, 88
bitmaps, 17
black-and-white, 6
brightness model, 92-96
clear pixels, 150-158
CLUT (Color Look-Up Tables), 96-99
color wheel, 90-91
dialog boxes, 306
gamma, 92
graphics, 87-95, 695-697
HelloWorld2 code, 16-23
direct, 17-18
indexed, 17-18
HSV model, 92-96
hue, 92
Macs, 64-66
palettes, 99-108, 127
animation, 106-108
drawing, 105-106
Picker, 93-95
RGB, 89
RYB, 89
saturation, 92
selecting, 91
subtractive, 88
tables, 127
values, 93
Color Classic (Mac), 63
color maps (graphics), 682-686
combining behaviors (game entities), 443-445
compiler
porting, 636, 639-641
warning messages (porting), 635
compression
CD-ROMs, 665-666
Hollywood Sound API, 365-366
Sound Manager, 343
configuring serial communications ports, 587-591
connection library
Outland, 600-601
closing, 607
reading/writing, 605-606
connection records (Outland), 602-606
connectionMade( ) function, 604
constants
Dungeon, 46-49
HelloWorld code, 8-10
controllers, 265-277
joysticks, 702-707
keyboard, 266-267, 700-707, 715
GetKeys( ), 267-268
key-down events, 267-268
user configuration, 266-267
mouse, 269-277, 701-702
Button( ), 269-278
GetMouse( ), 269-278
VelocityDemo, 272-277
controls
Demo code, 181
dialog boxes, 303-305
ConvertFileToMovieFile( ) function, 374
converting, 627-632, 637-638
code, 643-655
copyright laws, 638-639
porting, 632-639
see also porting
coordinates
Glypha III, 752-753
texture maps (QuickDraw 3), 531-539
CopyBits( ) function, 9, 118-120, 139, 144
graphics, 108-109, 680-682
CopyMask( ) function, 9, 144
CopyResource( ) function, 223
copyright laws, 638-639
CPUs, 670

assembly language, 670-671
byte order, 672-676
interrupt cycles, 676-677
memory, 671-672
macros, 675-676
CreateMACESndResourceFrom Ptr( ) function, 365
CreateMyCamera( ) function, 498
CreateMyMesh( ) function, 529
CreateMyTexturePixmapData( ) function, 534
CreateMyView( ) function, 497
CreateOffscreenBitMap( ) function, 7
CreateOffscreenBitMapLite( ) function, 7
CreateSndResourceFromPtr( ) function, 367
crsr (graphics formats), 256
CTab2Palette( ) function, 98, 103
CTabChanged( ) function, 98
ctSeed graphics (troubleshooting), 681-686
Culbertson bidding system (games), 463
CurResFile( ) function, 223
CURS (graphics format), 256

D

data (porting code), 656-657
data size parameters (PowerPC), 746-747
DDP (Datagram Delivery Protocol), 578-585
debugging
porting, 640-641, 666-667
sound, 385-386
declarations (Dungeon), 46-49
defining box object (QuickDraw 3), 514-515
Demo code, 131
controls, 181
Demo.res, 124
global.h, 123
rectangles, 183-188
shutdownDemo( ) function, 139
sprites, 140-143
Demo.res, 124
demoKey( ) function, 133
desk accessory interface, 292-293
desktop (hiding), 298-299
deterministic board games (game entities), 446-447
Tic-Tac-Toe, 448-463
devices (graphics)
CLUT (Color Look-Up Tables), 96-99
direct, 96
fixed, 96
indexed, 96-99
dialog boxes, 722-725
color, 306
controls, 303-305
DLOG resources, 303-305
editable texts, 304-305
filter functions, 307-309
filters, 306-307
high scores, 305
icons, 304-305
interfaces, 301-318
KeyConfig Demo, 311-318
modal, 305-306
pictures, 304-305
preferences, 305
resource files, 253
ScoresDemo, 310-311
static texts, 303-305
values, 309
direct color, HelloWorld2 code, 17-18
direct devices (graphics), 96
dirty rectangle, 179
disk events, 278
displaying menu bar, 129
DisposeCTable( ) function, 98
DisposeGWorld( ) function, 112
DisposeMovie( ) function, 374
DisposePalette( ) function, 103
disposeSprite( ) function, 141, 148
distances
Maze, 405-410
world representations, 398-411
Euclidean mathematics, 400-411
grids, 404-411
hexagonal, 402-411
DistanceTransformBackward( ) function, 408
DistanceTransformForward( ) function, 407
DITL (graphic format), 262
DLOG resources (dialog boxes), 303-305
DoAbout( ) function, 288
DoBackground( ) function, 286
DoDelay( ) function, 16
doDemoFrame( ) function, 131, 134, 186
doEncode( ) function, 172
doEventLoop( ) function, 129-130
DoFileMenu( ) function, 288
DoKey( ) function, 287
DoMenuSelection( ) function, 289
DoMouse( ) function, 286
DoUpdate( ) function, 288
draw context (QuickDraw 3), 498-500, 504-508
draw environment (QuickDraw 3), 495-512
draw loop (QuickDraw 3), 516-518
drawing
graphics, 257-259
full-screen, 687-690
palettes, 105-106
sprites, 138, 141
clear color, 150-158
clipped, 155
operations, 160

DrawMyObjects( ) function, 516
DrawPicture( ) function, 122, 142
drawSprite( ) function, 138, 149
Dungeon, 41-45
  code, 58-59
  constants, 46-49
  declarations, 46-49
  developing, 45
  graphics, 42-43
  moving, 44
  random numbers, 45
  sound, 43-44
  subroutines, 49-57
  variables, 46-49
Dungeon 1 Plus, 80
  exiting, 85
  maps, 80
  source code, 80-85
Dungeon 2, 284-290
Dungeon 3
  Apple Events, 238-241
  code, 241-242
  save files, 233-238
Dungeon E (game entities), 411-416
dungeon-digging games, 74-75

E

Edit menu, 722
editable texts (dialog boxes), 304-305
elastic collisions, 477-479
  2-D, 479-489
encoded sprites, 160-179
encoder (pixels), 178
encodeRect( ) function, 172-174
EndRendering( ) function, 518
endSpriteDraw( ) function, 141-142
EnterMovies( ) function, 372
EraseRect( ) function, 122
errors
  HelloWorld2 code, 21-23
  porting
    memory, 634-670
    toolbox, 634-670
Euclidean mathematics (distances), 400-402
evading (game entities), 436-438
events
  Apple, 279
  disk, 278
  Dungeon 2, 284-290
  interfaces, 278-290
  keyDown, 278
  loops, 279-283
  mouseDown, 278
  suspend/resume, 279
  update, 278
exiting Dungeon 1 Plus, 85
expansion (Sound Manager), 343
extended networks (AppleTalk), 551-617

F

faces (sprite sets), 4
fading graphics, 117-118
feedback from players, 668-670
file formats (Sound Manager), 354-358
  AIFF, 356-364
  snd, 354-356
File Manager, 209-215
File menu, 722
files
  File Manger, 209-215
  forks, 210
  handles, 211-213
    locking, 214-215
  high score lists (saving), 226-232
  passwords, 233
  porting, 662-663
  preferences, 223
  pregenerated data (storing), 215-220
  resource
    STR, 247
    STR#, 246
  Resource Manger, 209-215
  resources, 211-214
    alerts, 253
    Balloon Help, 248-250
    dialogs, 253
    Finder, 250-253
    graphics, 255-257
    menus, 253
    OS, 250-253
    purgeable, 215-250
    ResEdit, 243-244
    sound, 254
    switching between, 223-226
  save, 223
    Dungeon 3, 233-238
    saving, 232-242
fill styles (QuickDraw 3), 511-512
filter functions (dialog boxes), 307-309
filters (dialog boxes), 306-307
FilterStereoWindows( ) function, 520
FindComputerMove( ) function, 449
Finder (resource files), 250-253
FindPlayerMove( ) function, 449
fixed devices (graphics), 96
fixed point numbers, 132
  game entities, 423-426
  speed, 800
floating point numbers (PowerPC), 742-744
forks (files), 210
frames, 4
  doDemo( ) function, 136
  fading, 117-118
  game entities, 416
full-screen drawing, 687-690
functions
  ActivatePalette( ), 103
  AddPoints( ), 286
  addRectToUpdate( ), 181-182, 189

AddSoundToQ ( ), 38
adjustRect( ), 182
allocating memory (porting), 634
AnimateEntry( ), 106
AnimatePalette( ), 107
appletalkingStartup( ), 552
ASoundIsPlaying( ), 35
ATPReceiveCompletion C( ), 576
ATPReceiveCompletionRoutine ASM( ), 573
blitToScreen( ), 138, 179
BlockMove( ), 345
Button( ), 269
CenterAlert( ), 20
CheckKey( ), 268
clConnAlloc( ), 603
clConnAllow( ), 605
clConnClose( ), 606
clConnDispose( ), 606
clConnIdle( ), 606
clConnOpen( ), 605
clConnSetProp( ), 604
clearUpdate( ), 181
clIdle( ), 606
CloseDownSound( ), 16, 33
CloseMovieFile( ), 373
connectionMade( ), 604
ConvertFileToMovieFile( ), 374
CopyBits( ), 9, 118-120, 139, 144, 680
  graphics, 108-109
CopyMask( ), 9, 144
CopyResource( ), 223
CreateMACESndResourceFrom Ptr( ), 365
CreateMyCamera( ), 498
CreateMyMesh( ), 529
CreateMyTexturePixmap Data( ), 534
CreateMyView( ), 497
CreateOffscreenBitMap( ), 7
CreateOffscreenBitMapLite( ), 7
CreateSndResourceFromPtr( ), 367
CTab2Palette( ), 98, 103
CTabChanged( ), 98
CurResFile( ), 223
demoKey( ), 133
DisposeCTable( ), 98
DisposeGWorld( ), 112
DisposeMovie( ), 374
DisposePalette( ), 103
disposeSprite( ), 141, 148
DistanceTransform Backward( ), 408
DistanceTransform Forward( ), 407
DoAbout( ), 288
DoBackground( ), 286
DoDelay( ), 16
doDemoFrame( ), 131, 134, 186
doEncode( ), 172
doEventLoop( ), 129-130
DoFileMenu( ), 288
DoKey( ), 287
DoMenuSelection( ), 289
DoMouse( ), 286
DoUpdate( ), 288
DrawMyObjects( ), 516
DrawPicture( ), 122, 142
drawSprite( ), 138, 149
encodeRect( ), 172, 174
EndRendering( ), 518
endSpriteDraw( ), 141-142
EnterMovies( ), 372
EraseRect( ), 122
filter (dialog boxes), 307
FilterStereoWindows( ), 520
FindComputerMove( ), 449
FindPlayerMove( ), 449
Gestalt( ), 553
GetCTable( ), 98
GetDefaultOutput Volume( ), 346
GetDeviceList( ), 114
GetDialogItem( ), 309
GetGDevice( ), 114
GetGWorldPixMap( ), 112
GetKeys( ), 267-268
GetMainDevice( ), 114
GetMaxDevice( ), 114
GetMouse( ), 269
GetNewPalette( ), 103
GetNextDevice( ), 114
GetNextEvent( ), 727
GetNodeAddress( ), 553
GetPalette( ), 103
GetRate( ), 364
GetResource( ), 213
GetSoundResource( ), 359
GetSoundResource Information( ), 366
GetSysBeepVolume( ), 347
GetThisMacsDepth( ), 19
getUpdateRect( ), 181-183
globalsInit( ), 553
HasDepth( ), 115
HideMenuBar( ), 300, 719
hideMenuBar( ), 127
HitSprite( ), 468
InitializeForSound( ), 33
initMacToolboxes( ), 126, 170
InstallCallBackRoutine( ), 39
Is16BitAvailable( ), 363
IsMPPOpen( ), 552
IsSoundOn( ), 35
IsStereoAvailable( ), 363
IsVoiceEmpty( ), 364
KillOffscreenBitMap( ), 7
LipSynch( ), 13
LoadAllSounds( ), 34
LoadASound ( ), 33
loadSprite( ), 140, 146-148
LockPixels( ), 136
MacCanSwitchDepths( ), 19
MainLoop( ), 290

malloc( ), 634
ModalDialog( ), 308
moveObjects( ), 136
MoveSprite( ), 421-422, 426
MoviesTask( ), 375
MPPOpen( ), 552
NBPSetNTE( ), 556
NewATPCompletion Proc( ), 574
NewGWorld( ), 110
NewMovieFromFile( ), 373
NewPalette( ), 103
NSetPalette( ), 104
OffsetRect( ), 468
OpenDriver( ), 552, 586
OpenMovieFile( ), 373
OTLook( ), 615
Palette2CTab( ), 98
parameters (PowerPC), 745
PauseHardware( ), 365
PBRead( ), 594
PBWrite( ), 592
PGetRequest( ), 573
PlayASound ( ), 36
PlayFile( ), 385
PlayLoopSound ( ), 37
PlayQuickTimeNotes( ), 381, 385
PlaySoundEffect1( ), 385
PlaySoundHandle( ), 360
PlaySynchSound ( ), 36
PLookupName( ), 564
PmBackColor( ), 106
PmForeColor( ), 105-106
porting, 635
PostCallbackFunction( ), 345
PRemoveName( ), 562
PrerollMovie( ), 375
ProfileDump( ), 129
PSetSelfSend( ), 554
Q3RotateTransform( ), 529
Q3TextureShader_New( ), 534
QuickTime movies, 374-377
Random( ), 45
RectSeparate( ), 472
RedAlert( ), 20
RegionHit( ), 469
RegisterThisSound( ), 359
ReleaeAllSounds ( ), 34
ReleaseASound ( ), 34
renderShapeClipped( ), 156
renderSpriteClipped( ), 155
restoreBackground( ), 136, 197
ResumeHardware( ), 365
RGBForeColor( ), 105
RunSpriteEngine( ), 417, 468
SaveAIFFFile( ), 385
SaveGame( ), 234
SectRect( ), 468
SeparateBalls( ), 485
SerGetBuf( ), 593
SerHShake( ), 589
serialSendData( ), 592, 595
SerReset( ), 588
SerSetBuf( ), 591
ServiceTasks( ), 365
SetDefaultOutput Volume( ), 347
SetDoubleBufferBypass( ), 498
SetGDevice( ), 114
SetGWorld( ), 112
SetMouse( ), 271
SetMovieActive( ), 376
SetPalette( ), 104, 127
SetRate( ), 364
SetSoundVBCallBack( ), 365
SetStereoPlacement( ), 347
SetStereoPosition( ), 364
SetSysBeepVolume( ), 347
SetTheRects( ), 11
SetupAIFFHeader( ), 356
SetupMenus( ), 290
SetupSndHeader( ), 354
SetVolume( ), 364
ShowClam( ), 16
showMenuBar( ), 129
shutdownBackground( ), 196
shutdownDemo( ), 128, 139
shutdownEncoder( ), 171
SndChannelStatus( ), 338
SndDisposeChannel( ), 334
SndDoCommand( ), 335
SndDoImmediate( ), 335
SndNewChannel( ), 331
SndPauseFilePlay( ), 339
SndPlay( ), 337
SndStartFilePlay( ), 339
SndStopFilePlay( ), 339
SoundActive( ), 363
SoundCallBack( ), 38
SoundPriorityPlaying( ), 35
speed, 802-806
StartMovie( ), 375
StartQuickTimeMovie( ), 376
StartRendering( ), 517
startSpiteDraw( ), 138
startSpriteDraw( ), 141, 148
startupBackground( ), 195
startupDemo( ), 128, 132
startupEncoder( ), 170
StdFilter( ), 307
StopMovie( ), 375
SwitchingOutSound( ), 33
TestDeviceAttribute( ), 115
ThisMacCanPlaySounds( ), 35
ThisMacHasSoundChip( ), 35
TickCount( ), 808
UnionRect( ), 15
UnlockPixels( ), 136
UnsignedFixed ( ), 348
UseResFile( ), 223
WaitNextEvent( ), 280

G

game entities, 410-466
acceleration, 421-422
action games, 434-445
combining behaviors, 443-445
evading, 436-438

hunting, 434-436
programmed behaviors, 441-443
random movement, 438-441
sleeping, 441
board games, 445-466
deterministic, 446-463
Tic-Tac-Toe, 448-463
collision detection, 466-470
3-D games, 470
grids, 469-470
image space collisions, 470
sprites, 466-469
collision handling (bouncing), 470-489
Dungeon E, 411-416
fixed point arithmetic, 423-426
frames, 416
Hunter & Evader demos, 436-438
indeterministic, 463-466
positioning, 419-420
SE FixedPoint, 426
SE Gravity, 422
SE Gravity FixedPoint, 426-431
speed, 420-421
SpriteEngine, 417-419
sprites, 416-417
SpriteTools.c, 418
variables, 422
game mechanics (world representation), 389-390
distances, 398-411
network world, 396-397
polygon, 397-399
structured, 391-398
unstructured, 390-391
game play (porting), 628
games
2-D space games, 65-66
3-D action, 71-72
abstract action, 75-76
action game entities, 434-445
adventure, 72-74
board, 69-70
board games (game entities), 445-466
cards, 70-71
Castles: Siege and Conquest, 703
Culbertson bidding system, 463
debugging, 666-667
dialog boxes, 722-725
Dungeon, 41-45
code, 58-59
constants, 46-49
declarations, 46-49
developing, 45
graphics, 42-43
moving, 44
random numbers, 45
sound, 43-44
subroutines, 49-57
variables, 46-49
Dungeon 1 Plus, 80
exiting, 85
maps, 80
source code, 80-85
dungeon-digging, 74-75
existing games, 60-61
Glider, 711
Glypha III, 749-750
high score lists, 226-232
indeterministic, 463-466
Joust, 710
Mahjong, 464
Maze (code), 405-410
metaphors, 709-712
Myst, 712
Pac Man, 710
passwords, 233
platform, 66
player feedback, 668-670
programmer, 79-80
puzzle, 69-70
reviewers, 669
save files, 232-242
shipping, 669
side-scrollers, 67-68
simulations, 77
Space Invaders, 710
strategy, 77-78
Tic-Tac-Toe, 450
gamma (color), 92
GDevices, 113-116
gender, 728-730
Gestalt( ) function, 553
GetCTable( ) function, 98
GetDefaultOutputVolume( ) function, 346
GetDeviceList( ) function, 114
GetDialogItem( ) function, 309
GetGDevice( ) function, 114
GetGWorldPixMap( ) function, 112
GetKeys( ) function (keyboard layout), 267-268
GetMainDevice( ) function, 114
GetMaxDevice( ) function, 114
GetMouse( ) function, 269
GetNewPalette( ) function, 103
GetNextDevice( ) function, 114
GetNextEvent( ) function, 727
GetNodeAddress( ) function, 553
GetPalette( ) function, 103
GetRate( ) function, 364
GetResource( ) function, 213
GetSoundResource( ) function, 359
GetSoundResourceInformation( ) function, 366
GetSysBeepVolume( ) function, 347
GetThisMacsDepth( ) function, 19
getUpdateRect( ) function, 181-183
Glider, 711
global.h (Demo code), 123
globalsInit( ) function, 553
Glypha III, 749-750
coordinates, 752-753
idle mode, 751
initialization, 750-751, 757-795

play mode, 751-752
player modes, 753-754
rendering, 754-756
speed, 756-757
Gouraud shading (QuickDraw 3), 526-531
graphics
artwork (aesthetics), 712-713
buffers, 108-109
color, 87-95
additive, 88
brightness model, 92-96
color-wheel, 90-91
gamma, 92
HSV model, 92-96
hue, 92
Picker, 93-95
RGB, 89
RYB, 89
saturation, 92
selecting, 91
subtractive, 88
values, 93
CopyBits( ) function, 108-109, 118-120
devices
CLUT (Color Look-Up Tables), 96-99
direct, 96
fixed, 96
indexed, 96-99
drawing, 257-259
Dungeon, 42-43
fading, 117-118
formats
cicn, 259-263
DITL, 262
PICT, 259-263
GDevices, 113-116
GWorlds, 110-112
menu bar (hiding), 118
multiple monitors, 116-117
pixmaps, 108-109
porting, 628, 655
troubleshooting, 663
porting code, 647
QuickDraw 3D, 491
ray-tracing, 712
resource files, 255-257
troubleshooting, 677-698
artwork, 683-687
aspect ratio, 690-691
color, 695-697
color maps, 682-686
CopyBits( ) function, 680-682
ctSeed, 681-686
full-screen drawing, 687-690
monitors, 685
off-screen buffers, 684-687
pixels, 691-694
player options, 697-698
resolution, 682-687
screen address variables, 679-682
grids
collision detection, 469-470
distances, 404-416
GWorlds, 110-112

H

handles
files, 211-213
locking, 214-215
porting, 634
handshaking (serial communications), 589
HasDepth( ) function, 115
HelloWorld code, 6-30
animating, 13
contants, 8-10
PICT resources, 9-10
routines, 11-16
variables, 10-11
HelloWorld2 code
color, 16-23
direct, 17-18
indexed, 17-18
error checking, 21-23
MGWGraphics.c, 21
MGWUtilities.c, 18-21
pixmaps, 21
HelloWorld3.c code, 23-25
HelloWorld4 code, 25-26
HelloWorld5 code, 26-28
HelloWorld6 code, 28-30
HelloWorld6.1 code, 30
help (Balloon Help), 248-250
hexagonal distances, 402-404
HideMenuBar( ) function, 300, 719
hideMenuBar( ) function, 127
hiding
desktop, 298-299
menu bar, 118, 127, 299-301, 718-722
high scores
saving lists, 226-232
dialog boxes, 305
HitSprite( ) function, 468
Hollywood Sound API, 358-367
callbacks, 362-363
compression, 365-366
determining features, 363
memory, 359-360
playing sound, 360-362
sampling, 364
snd file formats, 366-367
stereo, 364-365
VBL, 365
volume, 364
HSV model color, 92-96
hue (color), 92
Human Interface Guidelines (Apple Computer), 630
Hunter & Evader demos, 436-438
hunting (game entities), 434-436

I

icl4 (graphics format), 256
ICN (graphics format), 256
ICON (graphics format), 256
icons (dialog boxes), 304-305

ics (graphics format), 256
ID numbers (masks), 145
idle mode (Glypha III), 751
idle routines (Outland), 606
image space collisions, 470
importing QuickTime, 370-372
indeterministic games, 463-466
indexed color, HelloWorld2 code, 17-18
indexed devices (graphics), 96-99
indexes, blitter routines (PowerPC), 737-741
initialization (Glypha III), 750-751
InitializeForSound( ) function, 33
initializing QuickDraw 3, 495
initMacToolboxes( ) function, 126, 170
input buffers (serial communications), 591
InstallCallBackRoutine( ) function, 39
instruments (QuickTime), 377-387
interfaces, 277-318
  aesthetics, 714-725
  application, 291-292
  centering window, 297-298
  desk accessory, 292-293
  desktop (hiding), 298-299
  dialog boxes, 301-318
    color, 306
    controls, 303-305
    DLOG resources, 302-305
    editable texts, 304-305
    filter functions, 307-309
    filters, 306-307
    high scores, 305
    icons, 304-305
    KeyConfig Demo, 311-318
    modal, 305-306
    pictures, 304-305
    preferences, 305
    ScoresDemo, 310-311
    static texts, 303-305
    values, 309
  events, 278-290
  MacOS, 293-294
  menu bar (hiding), 299-301
  menus, 294-297
  porting, 629-631
  screen-filler, 291
  take-over, 290-291
Internet
  Outland, 600-608
    client-server communications, 607-608
    connection library, 600-607
    connection records, 602-605
    closing connections, 606
  porting resources, 642
interpolation (QuickDraw 3), 509-510
interpretive languages, 798
interrupt cycles
  CPUs, 676-677
  Time Manager, 677
IP (Internet Protocol) Multicast, 616-618
Is16BitAvailable( ) function, 363
IsMPPOpen( ) function, 552
IsSoundOn( ) function, 35
IsStereoAvailable( ) function, 363
IsVoiceEmpty( ) function, 364

J

Johnston, Eric, 487
Joust, 710
joysticks, 702-707

K

key-down events, 267-268
keyboard controllers, 266-267, 700-707, 715
  GetKeys( ), 267-268
  key-down events, 267-268
  user-configuration, 266-267
KeyConfig Demo (dialog boxes), 311-318
keyDown events, 278
keystrokes, 133
KillOffscreenBitMap( ) function, 7

L

languages
  assembly (CPUs), 670-672
  C (porting), 637
  interpretive, 798
  porting code, 650
  speed, 797-798
LC (Mac), 63
LC 475, 62
LC III, 63
library (Outland), 600-601
  closing, 607
  reading/writing, 605-606
LipSynch( ) function, 13
listings, *see* code
lists (rectangles), 188-195
LoadAllSounds( ) function, 34
LoadASound( ) function, 33
loading
  sound (MGWSound.c), 33-34
  sprites, 140
loadSprite( ) function, 140, 146-148
locking handles (files), 214-215
LockPixels( ) function, 136
loops
  doEventLoop( ), 130
  events, 279-283
  QuickDraw 3 (draw), 516-518
  speed, 798-800
lossless compression (Sound Manager), 343
lossy compression (Sound Manager), 343

M

MacCanSwitchDepths( ) function, 19
MACE (Macintosh Audio Compression and Expansion), 343, 365

Macintosh (128K) processor, 818
Macintosh 512KE processor, 819
Macintosh Audio Compression and Expansion (MACE), 343
Macintosh Centris 610 processor, 827
Macintosh Centris 650 processor, 826
Macintosh Centris 660AV processor, 828
Macintosh Classic II processor, 821
Macintosh Classic processor, 820
Macintosh Color Classic processor, 821
Macintosh II processor, 822
Macintosh IIci processor, 824
Macintosh IIcx processor, 823
Macintosh IIfx processor, 825
Macintosh IIsi processor, 823
Macintosh IIvi processor, 824
Macintosh IIvx processor, 824
Macintosh IIx processor, 822
Macintosh LC 475 processor, 830
Macintosh LC 520 processor, 830
Macintosh LC 550 processor, 831
Macintosh LC 575 processor, 831
Macintosh LC 630 processor, 831
Macintosh LC II processor, 829
Macintosh LC III processor, 830
Macintosh LC processor, 829
Macintosh Performa processor, 839
Macintosh Plus processor, 819
Macintosh Portable processor, 832
Macintosh PowerBook 100 processor, 832
Macintosh PowerBook 140 processor, 832
Macintosh PowerBook 145, 145B processor, 833
Macintosh PowerBook 150 processor, 833
Macintosh PowerBook 160 processor, 834
Macintosh PowerBook 165 processor, 834
Macintosh PowerBook 165c processor, 834
Macintosh PowerBook 170 processor, 833
Macintosh PowerBook 180 processor, 835
Macintosh PowerBook 180c processor, 835
Macintosh PowerBook 520 processor, 835
Macintosh PowerBook 520c processor, 836
Macintosh PowerBook 540 processor, 836
Macintosh PowerBook 540c processor, 836
Macintosh PowerBook Duo 210 processor, 837
Macintosh PowerBook Duo 230 processor, 837
Macintosh PowerBook Duo 250 processor, 837
Macintosh PowerBook Duo 270c processor, 838
Macintosh PowerBook Duo 280 processor, 838
Macintosh PowerBook Duo 280c processor, 838
Macintosh Quadra 605 processor, 826
Macintosh Quadra 610 processor, 826
Macintosh Quadra 630 processor, 828
Macintosh Quadra 650 processor, 827
Macintosh Quadra 700 processor, 826
Macintosh Quadra 800 processor, 826, 827
Macintosh Quadra 840AV processor, 828
Macintosh Quadra 950 processor, 827
Macintosh SE processor, 820
Macintosh SE/30 processor, 820
MacOS (interfaces), 293-294
  resource files, 250-253
macros (porting), 636
  memory, 675
Macs, 62-63
  color, 64-66
  PowerPC
    601, 731-732
    603, 731-732
    604, 732
    alignment, 741-742
    data sizes, 746-747
    floating point math, 742-744
    function parameters, 745
    indexes, 737-741
    optimizing blitter routines, 732-742
    profiler, 733-737
    static declarations, 744
  screen sizes, 63-64
  System 6, 841-842
  see also processors
Mahjong (game), 464
MainLoop( ) function, 290
malloc( ) function, 634
maps (Dungeon 1 Plus), 80
markers (sound), 31-32
masks, 144
  CopyBits( ) function, 144
  ID numbers, 145
Maze (code), 405-410
MC68000 processor, 811
MC68020 processor, 812
MC68030 processor, 813
MC68040 processor, 813
MC68881 processor, 815
MC68882 processor, 815
MC68LC040 processor, 814
memory
  CPUs, 671-672
    macros, 675-676

game entities, 433-466
Hollywood Sound API, 359-360
porting, 661-662
allocation functions, 634
errors, 634-670
menu bar
displaying, 129
game aesthetics, 717
hiding, 118, 127, 299-301, 718-722
menus
aesthetics, 715-717
Edit, 722
File, 722
interfaces, 294-297
resource files, 253
meshes (QuickDraw 3), 522-527
Gouraud shading, 526-531
polygon faces, 526
rotating, 527-529
messages (AppleTalk), 565-585
metaphors (games), 709-712
MGWGraphics.c, 21
MGWSound.c, 32-39
callbacks, 38-39
sound
loading, 33-34
playing, 35-37
releasing, 34
MGWUtilities.c, 18-21
MIDI (porting code), 649
modal dialog boxes, 305
ModalDialog( ) function, 308
monitors
graphics, 685
multiple, 116-117
refreshing (animation), 4-5
sizes, 63-64
mono (Sound Manager), 330
mouse
controllers, 269-277, 701-702
Button( ), 269-278
GetMouse( ), 269-278
VelocityDemo, 272-277
doDemoFrame( ), 131
mouseDown event, 278
moveObjects( ) function, 136
MoveSprite( ) function, 421-422, 426
movies (QuickTime), 368
closing, 373-377
functions, 374-377
opening, 373-377
MoviesTask( ) function, 375
moving, 136
Dungeon, 44
random (game entities), 438-441
speed, 806-807
MPP driver (AppleTalk), 552-617
MPPOpen( ) function, 552
multichannels (Sound Manager), 331
multimedia (QuickTime), 367-384
multiple monitors (graphics), 116-117
music
porting, 657-659
QuickTime, 368-370
instruments, 377-387
notes, 379-383
tracks, 372-377
see also sound
Myst (game), 712

N

Name Binding Protocol (AppleTalk), 555-565
native calls (PowerPC), 806-807
navigating ResEdit, 244-246
NBPSetNTE( ) function, 556
network world (world representations), 396-397
networks, 547-548
AppleTalk, 548-585
addresses, 553-617
asynch parameters, 555-617
ATP (AppleTalk Transaction Protocol), 566-577
DDP (Datagram Delivery Protocol), 578-585
extended networks, 551-617
messages, 565-585
MPP driver, 552-617
Name Binding Protocol, 555-565
nodes, 549-617
self-sending, 554-617
socket numbers, 550-617
IP (Internet Protocol) Multicast, 616-618
Open Transport, 608-625
Outland, 600-608
client-server communications, 607-608
connection library, 600-607
connection records, 602-605
closing connections, 606
serial communications, 585-599
baud rates, 588
configuring ports, 587-591
handshaking, 589
input buffers, 591
receiving data, 592
sending data, 592
synchronizing streams, 595
NewATPCompletionProc( ) function, 574
NewGWorld( ) function, 110
NewMovieFromFile( ) function, 373
NewPalette( ) function, 103
nodes (AppleTalk), 549-617
notes (QuickTime), 379-383
NSetPalette( ) function, 104
numbers (fixed point), 132, 423

O

objects
QuickDraw 3, 494
sprites, 140
off-screen buffers, 684-686
OffsetRect( ) function, 468

Open Application (Apple Event), 238
Open Document (Apple Event), 238
Open Transport, 608-619
  XTI (X/Open Transport Interface), 609-610
OpenDriver( ) function, 552, 586
opening QuickTime movies, 373-377
OpenMovieFile( ) function, 373
optimizing PowerPC
  blitter routines, 732-742
  data sizes, 746-747
  floating point math, 742-744
  function parameters, 745
  static declarations, 744
OS resource files, 250-253
  see also MacOS
OTLook( ) function, 614
Outland, 600-608
  client-server communications, 607-608
  connection library, 600-601
    closing, 607
    reading/writing, 605-606
  connection records, 602-605
  connections (closing), 606
overlapping
  collision detection, 466
  rectangles, 192

P

Pac Man, 710
packaging aesthetics, 727-728
Palette Manager, 99-108
Palette2CTab( ) function, 98
palettes, 99-108
  animation, 106-108
  color, 127
  drawing, 105-106
panning (Sound Manager), 347-348
parameters (QuickDraw 3), 518-519
passwords, 233
PAT (graphics format), 257
PauseHardware( ) function, 365
PBRead( ) function, 594
PBWrite( ) function, 592
PGetRequest( ) function, 573
Picker (color), 93-95
PICT (graphics format), 257-259
  HelloWorld code, 9-10
pictures (dialog boxes), 304-305
pixels
  clear color, 150-158
  encoder, 178
  graphics, 691-694
  locking, 136
  masks, 144
pixmaps
  graphics, 108-109
  HelloWorld2 code, 21
plastic collisions, 476
platforms
  games, 66
  porting code, 646-651
play mode (Glypha III), 751-752
PlayASound ( ) function, 36
playback
  Hollywood Sound API, 360-362
  QuickTime music, 372-377
  rate (Sound Manager), 348-349
player modes (Glypha III), 753-754
player options (graphics), 697-698
players
  audience tastes, 728-730
  configuring keyboard layout, 266-267
  feedback, 668-670
  interfaces, 277-318
  see also users
PlayFile( ) function, 385
playing sound (MGWSound.c), 35-37
PlayLoopSound ( ) function, 37
PlayQuickTimeNotes( ) function, 381, 385
PlaySoundEffect1( ) function, 385
PlaySoundHandle( ) function, 360
PlaySynchSound ( ) function, 36
PLookupName( ) function, 564
pltt (graphics format), 257
PmBackColor( ) function, 106
PmForeColor( ) function, 105-106
pointers (porting), 634
polygons
  meshes (QuickDraw 3), 526
  texture maps (QuickDraw 3), 537-539
  world representations, 397-399
porting, 627-637
  APDA (Apple Programmer's and Developer's Association), 641
  books, 641-643
  bytes, 656-657
  C language, 637
  code, 643-655
    converting, 643-655
    data, 656-657
    development platform, 650-651
    graphics, 647
    MIDI, 649
    original language, 650
    original platform, 646-650
    sound, 648
    speed, 651-653
    System Software, 653-655
  compiler, 636
  converting, 632-639
  copyright laws, 638-639
  debuggers, 640-641
  files, 662-663
  functions, 635
  game play, 628

graphics, 628, 655
troubleshooting, 663
handles, 634
Human Interface Guidelines (Apple Computer), 630
interface, 629-631
Internet resources, 642
macros, 636
matching code to game needs, 632
memory, 661-662
allocation functions, 634
errors, 634-670
music, 657-659
pointers, 634
QuickTime, 659-661
RAM, 661
ResEdit, 640
resources, 641-643
sound, 629, 657-659
Think Reference, 634
Toolbox Assistant, 634
toolbox errors, 634-670
tools, 639-641
users, 631
variables, 635
video, 659-661
warning messages, 635
see also converting
ports (serial communications), 587-591
positioning game entities, 419-420
PostCallbackFunction( ) function, 345
Power Mac 6100/60, 62
Power Macintosh 5200/75 processor, 840
Power Macintosh 6100/60 processor, 839
Power Macintosh 7100/66 processor, 839
Power Macintosh 8100/80 processor, 840
PowerBook 160, 63
PowerPC
601, 731-732
603, 731-732
604, 732
alignment, 741-742
data sizes, 746-747
floating point math, 742-744
function parameters, 745
indexes, 737-741
optimizing blitter routines, 732-742
processors, 815-817
profiler, 733-737
speed (nonnative calls), 806-807
static declarations, 744
PowerPC 601 processor, 816
PowerPC 603 processor, 816
PowerPC 603e processor, 817
PowerPC 604 processor, 817
PowerPC 620 processor, 817
ppat (graphics format), 257
ppt# (graphics format), 257
preference files, 223
preferences (dialog boxes), 305
Preferences folder (settings), 220-223
pregenerated data (storing), 215-220
PRemoveName( ) function, 562
PrerollMovie( ) function, 375
Print Document (Apple Event), 238
processors, 811-815
Macintosh (128K), 818
Macintosh 512KE, 819
Macintosh Centris 610, 827
Macintosh Centris 650, 826
Macintosh Centris 660AV, 828
Macintosh Classic, 820
Macintosh Classic II, 821
Macintosh Color Classic, 821
Macintosh II, 822
Macintosh IIci, 824
Macintosh IIcx, 823
Macintosh IIfx, 825
Macintosh IIsi, 823
Macintosh IIvi, 824
Macintosh IIvx, 824
Macintosh IIx, 822
Macintosh LC, 829
Macintosh LC 475, 830
Macintosh LC 520, 830
Macintosh LC 550, 831
Macintosh LC 575, 831
Macintosh LC 630, 831
Macintosh LC II, 829
Macintosh LC III, 830
Macintosh Performa, 839
Macintosh Plus, 819
Macintosh Portable, 832
Macintosh PowerBook 100, 832
Macintosh PowerBook 140, 832
Macintosh PowerBook 145, 145B, 833
Macintosh PowerBook 150, 833
Macintosh PowerBook 160, 834
Macintosh PowerBook 165, 834
Macintosh PowerBook 165c, 834
Macintosh PowerBook 170, 833
Macintosh PowerBook 180, 835
Macintosh PowerBook 180c, 835
Macintosh PowerBook 520, 835
Macintosh PowerBook 520c, 836
Macintosh PowerBook 540, 836
Macintosh PowerBook 540c, 836
Macintosh PowerBook Duo 210, 837
Macintosh PowerBook Duo 230, 837
Macintosh PowerBook Duo 250, 837
Macintosh PowerBook Duo 270c, 838
Macintosh PowerBook Duo 280, 838

Macintosh PowerBook Duo 280c, 838
Macintosh Quadra 605, 826
Macintosh Quadra 610, 826
Macintosh Quadra 630, 828
Macintosh Quadra 650, 827
Macintosh Quadra 700, 826
Macintosh Quadra 800, 826, 827
Macintosh Quadra 840AV, 828
Macintosh Quadra 950, 827
Macintosh SE, 820
Macintosh SE/30, 820
MC68000, 811
MC68020, 812
MC68030, 813
MC68040, 813
MC68881, 815
MC68882, 815
MC68LC040, 814
Power Macintosh 5200/75, 840
Power Macintosh 6100/60, 839
Power Macintosh 7100/66, 839
Power Macintosh 8100/80, 840
PowerPC, 815-817
PowerPC 601, 816
PowerPC 603, 816
PowerPC 603e, 817
PowerPC 604, 817
PowerPC 620, 817
profilers, 126, 143
ProfileDump( ) function, 129
profilers, 126, 143
blitter routines (PowerPC), 733-737
programmed behaviors (game entities), 441-443
programmer games, 79-80
programmers
Johnston, Eric, 487
Simon, Phil, 539
Wright, Will, 427
PSetSelfSend( ) function, 554
purgeable resources (files), 215-250
puzzle games, 69-70

Q

Q3RotateTransform( ) function, 529
Q3TextureShader_New( ) function, 534
QuickDraw 3D, 491
attributes, 494
box object, 512-514
attributes, 514
defining, 514-515
referencing, 515-516
draw environment, 495-512
draw loop, 516-518
initializing, 495
meshes, 522-527
Gouraud shading, 525-526
polygon faces, 526
rotating, 527-529
objects, 494
parameters, 518-519
shader object, 508-512
stereoscopic, 519-521
style object, 508-512
style objects
backfacing, 510-511
fill styles, 511-512
interpolation, 509-510
texture maps, 531-539
polygons, 537-539
shader objects, 533-537
transformations, 530-531
view object, 496-498
camera object, 501-504
draw context, 498-500, 504-508
QuickTime, 367-384
importing, 370-372
instruments, 377-387
movies, 368
closing, 373-377
functions, 374-377
opening, 373-377
music, 368-370
playing tracks, 372-377
porting, 659-661
Quit Application (Apple Event), 238

R

racism, 728-730
RAM (porting), 661
random movement (game entities), 438-441
random numbers (Dungeon), 45
Random( ) function, 45
ray-tracing, 712
reading
CD-ROMs, 664-665
Outland library, 605-606
receiving data (serial communications), 592
rect variables (HelloWorld code), 11
rectangles
addRectToUpdate( ) function, 182
adjustRect( ) function, 182
arrays, 189
clearUpdate( ) function, 181
Demo code, 183-188
dirty, 179
getUpdateRect( ) function, 183
list, 188-195
overlaps, 192
RectSeparate( ) function, 472
RedAlert( ) function, 20
referencing box object (QuickDraw 3), 515-516
refresh rate (animation), 4-6
RegionHit( ) function, 469
RegisterThisSound( ) function, 359
ReleaeAllSounds ( ) function, 34
ReleaseASound ( ) function, 34
releasing sound (MGWSound.c), 34

rendering, 754-756
renderShapeClipped( ) function, 156
renderSpriteClipped( ) function, 155
ResEdit, 27
  DITL, 262
  navigating, 244-246
  porting, 640
  Resorcerer, 248
  resource files, 243-244
resolution (doubling), 682-686
Resorcerer, 248
Resource Manger, 209-215
resources
  files, 211-214
    alerts, 253
    Balloon Help, 248-250
    dialogs, 253
    Finder, 250-253
    graphics, 255-257
    menus, 253
    OS, 250-253
    purgeable, 215-250
    ResEdit, 243-244
    sound, 254
    STR, 247
    STR#, 246
    switching between, 223-226
  porting, 641-643
restoreBackground( ) function, 136, 197
ResumeHardware( ) function, 365
reviewers, 669
RGB color, 89
RGBForeColor( ) function, 105
RISC processors, 744
rotating meshes (QuickDraw 3), 527-529
routines, *see* functions
routines (Outland), 606
RunSpriteEngine( ) function, 417, 468
RYB color, 89

S

sampling
  Hollywood Sound API, 364
  Sound Manager, 342-343
SAT (Sprite Animation Toolkit), 467
saturation (color), 92
save files, 223
  Dungeon 3, 233-238
  saving, 232-242
SaveAIFFFile( ) function, 385
SaveGame( ) function, 234
saving
  high score lists, 226-232
  passwords, 233
  save files, 232-242
  settings (Preferences folder), 220-223
ScoresDemo (dialog boxes), 310-311
screen address variables (graphics), 679-682
screen shots, 727
screen-filler interface, 291
screen sizes, 63-64
scrolling games, 67-68
SE FixedPoint (game entities), 426
SE Gravity (game entities), 422
SE Gravity FixedPoint (game entities), 426-431
SE Simple (SpriteEngine Simple), 421
SectRect( ) function, 468
selecting color, 91
self-sending (AppleTalk), 554-617
sending data (serial communications), 592
SeparateBalls( ) function, 485
SerGetBuf( ) function, 593
SerHShake( ) function, 589
serial communications, 587-599
  baud rates, 588
  configuring ports, 587-591
  handshaking, 589
  input buffers, 591
  receiving data, 592
  sending data, 592
  synchronizing streams, 595
serialSendData( ) function, 592, 595
SerReset( ) function, 588
SerSetBuf( ) function, 591
servers (Outland), 607-608
ServiceTasks( ) function, 365
SetDefaultOutputVolume( ) function, 347
SetDoubleBufferBypass( ) function, 498
SetGDevice( ) function, 114
SetGWorld( ) function, 112
SetMouse( ) function, 271
SetMovieActive( ) function, 376
SetPalette( ) function, 104, 127
SetRate( ) function, 364
SetSoundVBCallBack( ) function, 365
SetStereoPlacement( ) function, 347
SetStereoPosition( ) function, 364
SetSysBeepVolume( ) function, 347
SetTheRects( ) function, 11
settings (Preferences folder), 220-223
SetupAIFFHeader( ) function, 356
SetupMenus( ) function, 290
SetupSndHeader( ) function, 354
SetVolume( ) function, 364
shader object (QuickDraw 3), 508-512
  texture maps, 533-537
shading, Gouraud (QuickDraw 3), 526-531
shipping games, 669
ShowClam( ) function, 16
showMenuBar( ) function, 129
shutdownBackground( ) function, 196
shutdownDemo( ) function, 128, 139
shutdownEncoder( ) function, 171
SICN (graphics format), 257

side-scrolling games, 67-68
silence (Sound Manager), 337
Simon, Phil, 539
simulation games, 77
SIZE (resource files), 250
skip pixels, 165
sleeping (game entities), 441
snd (file format), 354-356
  Hollywood Sound API, 366-367
SndChannelStatus( ) function, 338
SndDisposeChannel( )
  function, 334
SndDoCommand( ) function, 335
SndDoImmediate( ) function, 335
SndNewChannel( ) function, 331
SndPauseFilePlay( ) function, 339
SndPlay( ) function, 337
SndStartFilePlay( ) function, 339
SndStopFilePlay( ) function, 339
socket numbers (AppleTalk),
  550-617
sound, 30-39
  aesthetics, 725-726
  asynchronous, 32
  debugging, 385-386
  Dungeon, 43-44
  markers, 31-32
  MGWSound.c, 32
    callbacks, 38-39
    loading, 33-34
    playing, 35-37
    releasing, 34
  porting, 629, 657-659
  porting code, 648
  resource files, 254
  synchronous, 32
  troubleshooting, 698-700
    music, 700
    streaming audio, 699
Sound Manager, 328-342
  callbacks, 344-346
  channels, 331
  compression, 343
  data types, 330-342
  expansion, 343
  file formats, 354-358
    AIFF, 356-364
    snd, 354-356
  Hollywood Sound API, 358-367
    callbacks, 362-363
    compression, 365-366
    determining features, 363
    memory, 359-360
    playing sound, 360-362
    sampling, 364
    snd file format, 366-367
    stereo, 364-365
    VBL, 365
    volume, 364
  lossless, 343
  lossy, 343
  mono, 330
  panning, 347-348
  playback rate, 348-349
  sampling, 342-343
  silence, 337
  SndDisposeChannel( )
    function, 334
  SndDoCommand( )
    function, 335
  SndDoImmediate( )
    function, 335
  SndNewChannel( )
    function, 331
  SndPlay( ) function, 337
  status, 338
  stereo, 330
  VBL (Vertical Blanking task),
    349-353
  volume, 346-347
SoundActive( ) function, 363
SoundCallBack( ) function, 38
SoundPriorityPlaying( )
  function, 35
source code (Dungeon 1 Plus),
  80-85
Space Invaders, 710
speed
  animation, 2-4
  fixed point arithmetic, 800
  functions, 802-806
  game entities, 420-421
  Glypha III, 756-757
  languages, 797-798
  limits, 807-810
    current time, 808
    TickCount( ) functions, 808
  loops, 798-800
  moving data, 806-807
  nonnative calls (PowerPC),
    806-807
  porting code, 651-653
  tables, 800-802
  Toolbox, 802-806
Sprite Animation Toolkit
  (SAT), 467
sprite sets, 4
SpriteEngine
  game entities, 417-419
  SE Simple, 421
SpriteHandlers.h, 419
sprites, 4, 121
  clear color, 150-158
  clipped, 155, 169
  collision detection, 466-469
  Demo code, 140-143
  drawing, 138-141
    operations, 160
  encoded, 159-179
    tool, 170
  game entities, 416-417
  Hunter & Evader demos,
    436-438
  loading, 140
  objects, 140
  structures, 145
SpriteTools.c, 418
StartMovie( ) function, 375
StartQuickTimeMovie( )
  function, 376
StartRendering( ) function, 517

startSpiteDraw( ) function, 138
startSpriteDraw( ) function, 141, 148
startupBackground( ) function, 195
startupDemo( ) function, 128, 132
startupEncoder( ) function, 170
static declarations (PowerPC), 744
static texts (dialog boxes), 303-305
status (Sound Manager), 338
StdFilter( ) function, 307
stereo
  Hollywood Sound API, 364-365
  Sound Manager, 330
stereoscopic (QuickDraw 3), 519-521
StopMovie( ) function, 375
storing pregenerated data, 215-220
STR (resource files), 247
STR# (resource files), 246
strategy games, 77-78
streaming audio (sound), 699
streams (serial communications), 595
strings (resource files), 246-247
structure definition, 131
structured world representations, 391-398
structures (sprites), 145
style objects (QuickDraw 3), 508-512
  backfacing, 510-511
  fill styles, 511-512
  interpolation, 509-510
submitting draw loop (QuickDraw 3), 518
subroutines (Dungeon), 49-57
subtractive color, 88
suspend/resume events, 279
switching between resource files, 223-226
SwitchingOutSound( ) function, 33
synchronizing streams (serial communications), 595
synchronous sound, 32
System 6, 841-842
System Software (porting code), 653-655

T

tables
  color, 127
  speed, 800-802
take-over interface, 290-291
TestDeviceAttribute( ) function, 115
texture maps (QuickDraw 3), 531-539
  polygons, 537-539
  shader objects, 533-537
Think Reference (porting), 634
ThisMacCanPlaySounds( ) function, 35
ThisMacHasSoundChip( ) function, 35
Tic-Tac-Toe, 448-463
TickCount( ) function, 808
Ticks (time), 4
Time Manager (interrupt cycles), 677
Toolbox, 802-806
Toolbox Assistant (porting), 634
toolbox errors (porting), 634-670
tools
  encoded sprites, 170
  porting, 639-641
transformations (QuickDraw 3), 530-531
  rotating meshes, 527-529
  scope, 530
troubleshooting
  CPUs, 670-677
    assembly language, 670-672
    byte order, 672-676
    interrupt cycles, 676-677
    memory, 671-672, 675-676
  graphics, 677-698
    artwork, 683-687
    aspect ratio, 690-691
    color, 695-697
    color maps, 682-686
    CopyBits( ) function, 680-682
    ctSeed, 681-686
    full-screen drawing, 687-690
    monitors, 685
    off-screen buffers, 684-687
    pixels, 691-694
    player options, 697-698
    resolution, 682-687
    screen address variables, 679-682
  porting graphics, 663
  sound, 698
    music, 700
    streaming audio, 699

U

UnionRect( ) function, 15
UnlockPixels( ) function, 136
UnsignedFixed ( ) function, 348
unstructured world representations, 390-391
update events, 278
UseResFile( ) function, 223
users
  configuring keyboard layout, 266-267
  interfaces, 277-318
  porting, 631
  see also players

V

values
  color, 93
  dialog boxes, 309
vaporware, 644
variables
  Dungeon, 46-49
  game entities, 422

HelloWorld code, 10-11
porting, 635
VBL (Vertical Blanking task)
Hollywood Sound API, 365
Sound Manager, 349-353
VelocityDemo (mouse controller), 272-277
vers (resource files), 250
vertices, polygons (QuickDraw 3), 526
video (porting), 659-661
video cards, 96
view objects (QuickDraw 3), 496-498
camera object, 501-504
draw context, 498-500, 504-508
violence, 728-730
volume
Hollywood Sound API, 364
Sound Manager, 346-347

W-X-Y-Z

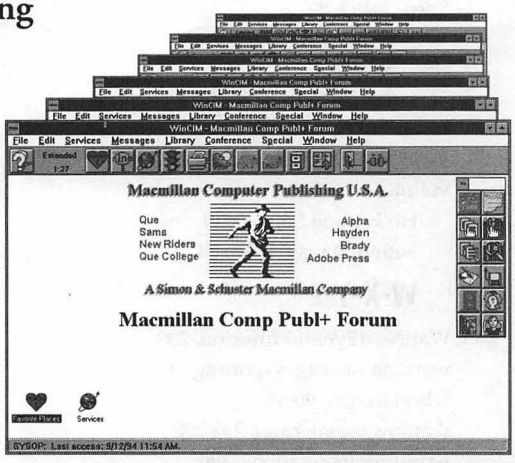
WaitNextEvent( ) function, 280
warning messages (porting), 635
wheel (color), 90-91
windows (centering), 297-298
world representations, 390
distances, 398-411
Euclidean mathematics, 400-402
grids, 404-416
hexagonal, 402-404
Maze, 405-410
game entities, 410-431
acceleration, 421-422
Dungeon E, 411-416
fixed point arithmetic, 423-426
positioning, 419-420
SE FixedPoint, 426
SE Gravity, 422
SE Gravity FixedPoint, 426-431
speed, 420-421
SpriteEngine, 417-419
sprites, 416
variables, 422
network world, 396-397
polygons, 397-399
structured, 391-398
unstructured, 390-391
Wright, Will, 427
writing (Outland library), 605-606

XTI (X/Open Transport Interface), 609-610

GET CONNECTED
to the ultimate source of computer information!

The MCP Forum on CompuServe

Go online with the world's leading computer book publisher! Macmillan Computer Publishing offers everything you need for computer success!



Find the books that are right for you!
A complete online catalog, plus sample chapters and tables of contents give you an in-depth look at all our books. The best way to shop or browse!

- Get fast answers and technical support for MCP books and software
- Join discussion groups on major computer subjects
- Interact with our expert authors via e-mail and conferences
- Download software from our immense library:
  - Source code from books
  - Demos of hot software
  - The best shareware and freeware
  - Graphics files

Join now and get a free CompuServe Starter Kit!

To receive your free CompuServe Introductory Membership, call **1-800-848-8199** and ask for representative #597.

The Starter Kit includes:

- Personal ID number and password
- $15 credit on the system
- Subscription to *CompuServe Magazine*

Once on the CompuServe System, type:

GO MACMILLAN

for the most computer information anywhere!

MACMILLAN COMPUTER PUBLISHING

CompuServe

Get a deal on MacTech Magazine just for being a eader of *Tricks of the Mac Game Programming Gurus!*

Information! The Name of The Game

Nothing changes faster than Macintosh computer technology. *MacTech Magazine* (formerly MacTutor) is dedicated to bringing quality tools and information to the Macintosh developer community.

Each month, we provide you with the information *you* need to know – reviews of hot products, news and events, in-depth feature articles and much, much more – directly from the experts in the field. Regular columnists include industry leaders such as noted author Dave Mark, Apple's employee #8 Chris Espinosa, Apple software engineers, and an entire team from Symantec! Articles are accompanied by full working source code examples.

Always on the cutting edge, *MacTech Magazine* is an invaluable source to overcome programming problems and the day-to-day issues that plague Macintosh programmers. *MacTech Magazine* brings you articles such as:

- **A regular Getting Started column** by noted author Dave Mark
- **PowerPC information** by those "in the know" at Apple and from the developer community
- **Tips and Tidbits** from programmers around the globe
- **Programmer's Challenge** a monthly contest by efficiency expert Bob Boonstra
- **Inside Information column** by Apple's Chris Espinosa – one of Apple's first employees
- **NewsBits** for up-to-date developer relevant news
- **From the Field Reports** give you information about conferences, announcements and other developer relevant breaking news
- **A monthly Symantec Top 10 column** by Symantec's Technical Support group
- **Legal Eagles** covers legal issues specific to developers – from the release of new software to copyrights
- **The latest in shortcuts and insider secrets** from massive speed enhancements to faster coding methods to cool tricks
- **Full source code listings** available on CompuServe, America Online, AppleLink, the Internet floppy disk, and CD
- **Coverage of a broad range of languages** and environments including C, C++, Pascal, Assembly, MacApp®, TCL, OOP, HyperCard®, Forth, FORTRAN, Lisp, Prograph™, database languages and others
- **The latest developer tools at discount prices** available through *MacTech Magazine's* Mail Order Store
- **Hot product reviews, and *more!***

In short, *MacTech Magazine* will help you to be a better informed, more productive Macintosh developer. Maximize your programming efforts.

"Articles in MacTech Magazine are here to give developers the additional support that they need."

– Gene Wang, Executive Vice President of Applications and Development Tools, Symantec Corporation

"Every programmer should subscribe to MacTech Magazine. It is an indispensable source of programming tips and information on development tools."

– Gary Little, Developer Tools Product Manager, Apple Computer, Inc.

Invest in a subscription today!

Try MacTech Magazine. *Risk free!*

❑ **YES!** I would like to receive a one year subscription (12 issues) to MacTech Magazine for $39.95 — a 43% savings! If I don't find MacTech Magazine helpful, I can cancel my subscription for a prompt refund.

Name _____

Company _____

Address _____

City, State, Zip _____

❑ Payment enclosed. (Please use separate envelope.) ❑ Bill me later

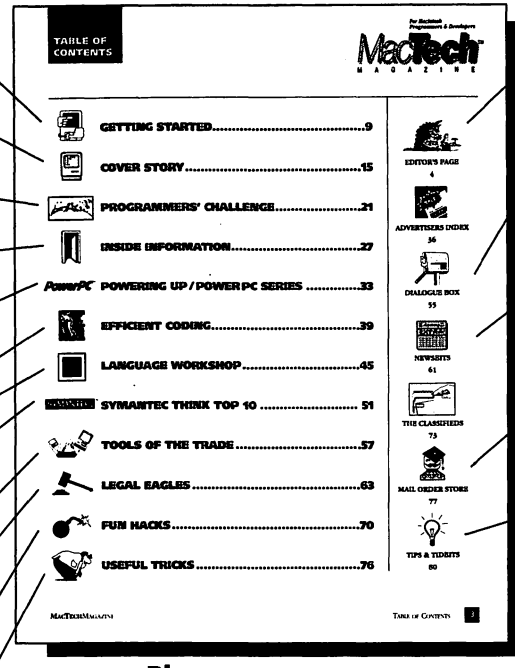
For Macintosh Programmers & Developers

MacTech™ MAGAZINE

Voice: 805/494-9797 • Fax: 805/494-9798

Offer good in U.S. and possessions. Canadian subscriptions discounted to US$51.95. International US$89.95. Please allow 4-6 weeks for shipment of first issue.

1. Dave Mark, noted author, demystifies programming for those new to the Mac.
2. In-depth coverage of System Software Technologies, Object Oriented Programming, Visual Programming, Databasing ...
3. Bob Boonstra's monthly contest will challenge even the most experienced programmer!
4. Chris Espinosa talks about Apple – from the inside. Where it has been and where it is going.
5. Get information from the experts and Apple on how to make the transition to Power Macintosh.
6. Articles from industry leaders that will improve your code writing.
7. Workshop articles including languages from Assembly and C to Pascal, FORTRAN, and others.
8. Technical support for Symantec's products.
9. Impartial insider reviews of the latest development tools.
10. Legal issues specific to programming – from the release of new software to copyrights.
11. Interesting, not so conventional programming ideas.
12. Ideas to help you clean up, use shortcuts on, and organize your work.



13. Comments on the industry and pertinent notable events.
14. An open forum for developers to share their thoughts, dilemmas, solutions and opinions.
15. Recent releases, product updates, company mergers, shipping dates ...
16. Valuable developer tools and reference materials at discounted prices.
17. Your opportunity to share information or tips useful to other programmers – see it in print & get paid for it!

Plus:

18. Full, clearly written explanations.
19. Illuminating tables, diagrams & illustrations
20. Source code within each article that demonstrates technique.

TO MAIL, FOLD ALONG DOTTED LINE AND SEAL WITH TAPE. PLEASE DO NOT STAPLE.

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 71745 LOS ANGELES, CA 90025

POSTAGE WILL BE PAID BY ADDRESSEE

XPLAIN CORPORATION
P O BOX 250055
LOS ANGELES CA 90099-3873

Metrowerks CodeWarrior™ Order Form

Please Print Clearly

Name

Company or Educational Institution

Address

Address

City State/Province Zip/Postal Code

Telephone Number

Fax Number

E-mail Address

	Qty	Cost Each	Total
CodeWarrior Gold		$399.00	
CodeWarrior Bronze		$99.00	
CodeWarrior Magic		$299.00	
CodeWarrior Academic		$99.00	
Academic Lab Pack 10		$650.00	
Academic Lab Pack 25		$1450.00	
Inside CodeWarrior Doc.		$34.95	
		Total	
		Sales Tax (As May Apply)	
		Shipping & Handling	
		Total Payment	

☐ Visa ☐ Master Card ☐ American Express

Credit Card Number

Expiry Date (MM/YY)

Cardholder's Signature

To Order:

Mail: Metrowerks Corp.
The MCC Building
3925 West Braker Lane
Suite 310
Austin, TX 78759

Voice: (512) 305-0400

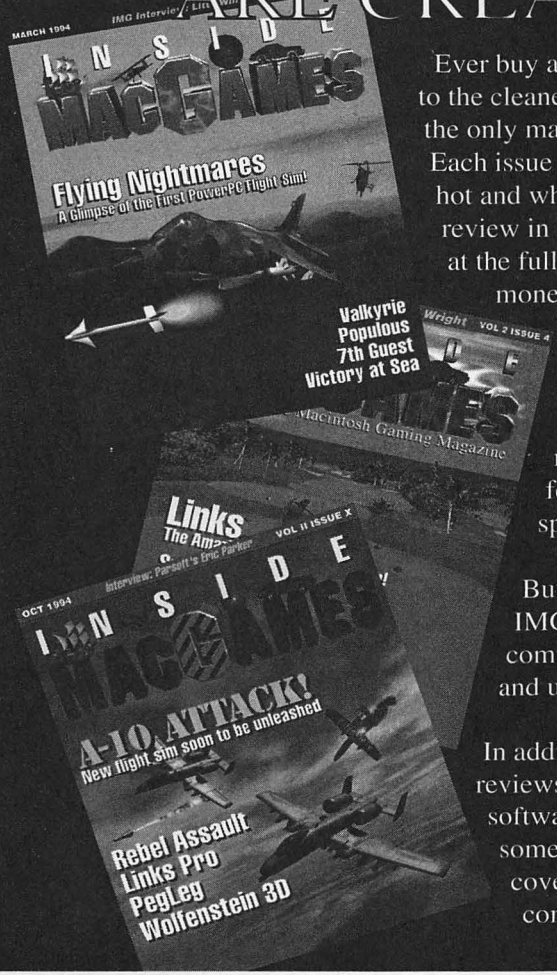
Fax: (512) 305-0440



TM

For site license and general sales information:
Please call Metrowerks at (512) 305-0400 or send email to sales@metrowerks.com

NOT ALL GAMES ARE CREATED EQUAL



Ever buy a Macintosh game and feel like you've been taken to the cleaners? What you need is Inside Mac Games (IMG), the only magazine solely devoted to Macintosh entertainment. Each issue is packed with in-depth game reviews of what's hot and what's not. Before you buy that next game, read the review in Inside Mac Games, try the demo, and take a look at the full-size screen shots before putting your hard-earned money down.

Inside Mac Games also gives you exclusive previews of the hottest soon-to-be released games, stunning color screen shots, QuickTime™ movies, insightful interviews, industry news, feature columns, hints, tips, and tricks, rumors, and special reports.

But hang on, there's more! Each CD-ROM issue of IMG is filled with the latest shareware games, commercial games demos, walkthroughs, bug fixers and updaters, and cheater programs. You get it all!

In addition to games, Inside Mac Games also provides reviews of the latest educational and edutainment software. So no matter what your interests are, IMG has something for the entire family. And best of all, IMG covers ONLY Macintosh entertainment. Finally, a computer entertainment magazine "for the rest of us!"

YES! I WANT TO SUBSCRIBE TO INSIDE MAC GAMES!

Get 10 issues on CD-ROM for just $59 ($79 outside the U.S.)

FULL NAME (please print clearly) _____ ADDRESS _____

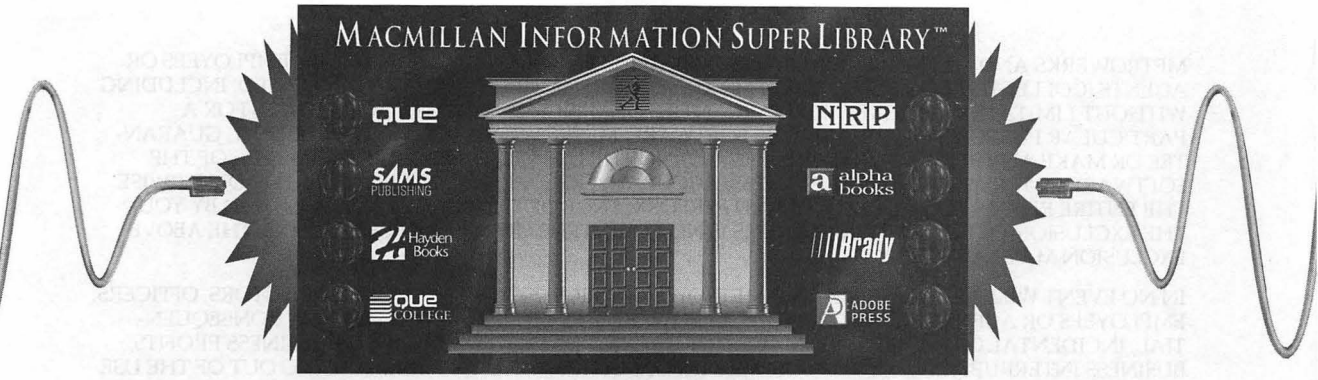
CITY _____ STATE _____ ZIP _____

☐ PAYMENT ENCLOSED ☐ VISA/MC CARD# _____ EXP DATE _____

For faster service call toll free: 1-800-339-0636

Make all checks payable to: Inside Mac Games, 3862 Grace Lane, Glenview, IL 60025

PLUG YOURSELF INTO...



THE MACMILLAN INFORMATION SUPERLIBRARY™

Free information and vast computer resources from the world's leading computer book publisher—online!

FIND THE BOOKS THAT ARE RIGHT FOR YOU!

A complete online catalog, plus sample chapters and tables of contents give you an in-depth look at ***all*** of our books, including hard-to-find titles. It's the best way to find the books you need!

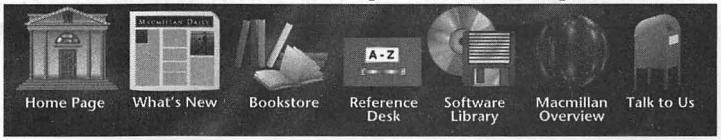
- STAY INFORMED with the latest computer industry news through our online newsletter, press releases, and customized Information SuperLibrary Reports.
- GET FAST ANSWERS to your questions about MCP books and software.
- VISIT our online bookstore for the latest information and editions!
- COMMUNICATE with our expert authors through e-mail and conferences.
- DOWNLOAD SOFTWARE from the immense MCP library:
  - Source code and files from MCP books
  - The best shareware, freeware, and demos
- DISCOVER HOT SPOTS on other parts of the Internet.
- WIN BOOKS in ongoing contests and giveaways!

TO PLUG INTO MCP:

GOPHER: gopher.mcp.com

FTP: ftp.mcp.com

WORLD WIDE WEB: **http://www.mcp.com**



METROWERKS AND METROWERKS' LICENSOR(S), AND THEIR DIRECTORS, OFFICERS, EMPLOYEES OR AGENTS (COLLECTIVELY METROWERKS) MAKE NO WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE SOFTWARE. METROWERKS DOES NOT WARRANT, GUARANTEE OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS OR OTHERWISE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE IS ASSUMED BY YOU. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME JURISDICTIONS. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

IN NO EVENT WILL METROWERKS AND METROWERKS' LICENSOR(S), AND THEIR DIRECTORS, OFFICERS, EMPLOYEES OR AGENTS (COLLECTIVELY METROWERKS) BE LIABLE TO YOU FOR ANY CONSEQUENTIAL, INCIDENTAL OR INDIRECT DAMAGES (INCLUDING DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, AND THE LIKE) ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE EVEN IF METROWERKS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. BECAUSE SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU. Metrowerks liability to you for actual damages from any cause whatsoever, and regardless of the form of the action (whether in contract, tort (including negligence), product liability or otherwise), will be limited to exceed the cost of the replacement of the media on which the software is distributed.

PLEASE READ THIS LICENSE CAREFULLY BEFORE USING THE SOFTWARE. BY USING THE SOFTWARE, YOU ARE AGREEING TO BE BOUND BY THE TERMS OF THIS LICENSE. IF YOU DO NOT AGREE TO THE TERMS OF THIS LICENSE, PROMPTLY RETURN THE UNUSED SOFTWARE TO THE PLACE WHERE YOU OBTAINED IT AND YOUR MONEY WILL BE REFUNDED.

1. License. The application, demonstration, system and other software accompanying this License, whether on disk, in read-only memory, or on any other media (the "Software") the related documentation and fonts are licensed to you by Metrowerks. You own the disk on which the Software and fonts are recorded but Metrowerks and/or Metrowerks' Licensor retain title to the Software, related documentation and fonts. This License allows you to use the Software and fonts on a single Apple computer and make one copy of the Software and fonts in machine-readable form for backup purposes only. You must reproduce on such copy the Metrowerks copyright notice and any other proprietary legends that were on the original copy of the Software and fonts. You may also transfer all your license rights in the Software and fonts, the backup copy of the Software and fonts, the related documentation and a copy of this License to another party, provided the other party reads and agrees to accept the terms and conditions of this License.

2. Restrictions. The Software contains copyrighted material, trade secrets and other proprietary material. In order to protect them, and except as permitted by applicable legislation, you may not decompile, reverse engineer, disassemble or otherwise reduce the Software to a human-perceivable form. You may not modify, network, rent, lease, loan, distribute or create derivative works based upon the Software in whole or in part. You may not electronically transmit the Software from one computer to another or over a network.

3. Termination. This License is effective until terminated. You may terminate this License at any time by destroying the Software, related documentation and fonts and all copies thereof. This License will terminate immediately without notice from Metrowerks if you fail to comply with any provision of this License. Upon termination you must destroy the Software, related documentation and fonts and all copies thereof.

4. Export Law Assurances. You agree and certify that neither the Software nor any other technical data received from Metrowerks, nor the direct product thereof, will be exported outside the United States except as authorized and as permitted by the laws and regulations of the United States. If the Software has been rightfully obtained by you outside of the United States, you agree that you will not re-export the Software nor any other technical data received from Metrowerks, nor the direct product thereof, except as permitted by the laws and regulations of the United States and the laws and regulations of the jurisdiction in which you obtained the Software.

5. Government End Users. If you are acquiring the Software and fonts on behalf of any unit or agency of the United States Government, the following provisions apply. The Government agrees: (i) If the Software and fonts are supplied to the Department of Defense (DoD), the Software and fonts are classified as "Commercial Computer Software" and the Government is acquiring only "restricted rights" in the Software, its documentation and fonts as

that term is defined in Clause 252.227-7013(c)(1) of the DFARS; and (ii) If the Software and fonts are supplied to any unit or agency of the United States Government other than the DoD, the Government's rights in the Software, its documentation and fonts will be as defined in Clause 52.227-19(c)(2) of the FAR or, in the case of NASA, in Clause 18-52.227-86(d) of the NASA Supplement to the FAR.

6. Limited Warranty on Media. Metrowerks warrants the diskettes and/or compact disc on which the Software and fonts are recorded to be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of purchase as evidenced by a copy of the receipt. Metrowerks' entire liability and your exclusive remedy will be replacement of the diskettes and/or compact disc not meeting Metrowerks' limited warranty and which is returned to Metrowerks or a Metrowerks authorized representative with a copy of the receipt. Metrowerks will have no responsibility to replace a disk/disc damaged by accident, abuse or misapplication. ANY IMPLIED WARRANTIES ON THE DISKETTES AND/OR COMPACT DISC, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF DELIVERY. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS, AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY BY JURISDICTION.

7. Disclaimer of Warranty on Apple Software. You expressly acknowledge and agree that use of the Software and fonts is at your sole risk. Except as is stated above, the Software, related documentation and fonts are provided "AS IS" and without warranty of any kind and Metrowerks and Metrowerks' Licensor(s) (for the purposes of provisions 7 and 8, Metrowerks and Metrowerks' Licensor(s) shall be collectively referred to as "Metrowerks") EXPRESSLY DISCLAIM ALL OTHER WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. HB DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE SOFTWARE WILL MEET YOUR REQUIREMENTS, OR THAT THE OPERATION OF THE SOFTWARE WILL BE UNINTERRUPTED OR ERROR-FREE, OR THAT DEFECTS IN THE SOFTWARE AND THE FONTS WILL BE CORRECTED. FURTHERMORE, HB DOES NOT WARRANT OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE AND FONTS OR RELATED DOCUMENTATION IN TERMS OF THEIR CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE. NO ORAL OR WRITTEN INFORMATION OR ADVICE GIVEN BY HB OR AN HB AUTHORIZED REPRESENTATIVE SHALL CREATE A WARRANTY OR IN ANY WAY INCREASE THE SCOPE OF THIS WARRANTY. SHOULD THE SOFTWARE PROVE DEFECTIVE, YOU (AND NOT HB OR AN HB AUTHORIZED REPRESENTATIVE) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

8. Limitation of Liability. UNDER NO CIRCUMSTANCES INCLUDING NEGLIGENCE, SHALL HB BE LIABLE FOR ANY INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES THAT RESULT FROM THE USE OR INABILITY TO USE THE SOFTWARE OR RELATED DOCUMENTATION, EVEN IF HB OR AN HB AUTHORIZED REPRESENTATIVE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME JURISDICTIONS DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.

In no event shall Metrowerks' total liability to you for all damages, losses, and causes of action (whether in contract, tort (including negligence) or otherwise) exceed that portion of the amount paid by you which is fairly attributable to the Software and fonts.

9. Controlling Law and Severability. This License shall be governed by and construed in accordance with the laws of the United States and the State of California, as applied to agreements entered into and to be performed entirely within California between California residents. If for any reason a court of competent jurisdiction finds any provision of this License, or portion thereof, to be unenforceable, that provision of the License shall be enforced to the maximum extent permissible so as to affect the intent of the parties, and the remainder of this License shall continue in full force and effect.

10. Complete Agreement. This License constitutes the entire agreement between the parties with respect to the use of the Software, the related documentation and fonts, and supersedes all prior or contemporaneous understandings or agreements, written or oral, regarding such subject matter. No amendment to or modification of this License will be binding unless in writing and signed by a duly authorized representative of Metrowerks.

The Tricks CD-ROM

The *Tricks of the Mac Game Programming Gurus* CD-ROM includes:

- All the source code from all the chapters in the book. Due to space limitations, the demo and sample code has not always been printed in full in the book, but it is all here on CD-ROM!

  Note
  Metrowerks C/C++ projects will not run directly from the CD-ROM. To compile and update them, projects must be on read/write memory, and the CD-ROM is strictly read-only. Drag each individual project folder (including project, source code, and resources) to your hard drive before opening its project.

- Metrowerks C/C++ Lite, an instructional version of the Metrowerks CodeWarrior programming environment. Other than a few disabled commands, such as New Project, Metrowerks C/C++ Lite runs just like the full version. It will run from the CD-ROM, though somewhat more slowly than it runs from a hard drive. For maximum performance, run the CodeWarrior installer. This will copy all the required files to your hard drive. About 30 MB of disk space is required.

  Don't Call Metrowerks
  Metrowerks cannot provide technical support for this limited version. There is a special offer for the commercial version of Metrowerks CodeWarrior near the end of this book. Please do not call Metrowerks for technical support until you have a full version of CodeWarrior Bronze, Silver, or Gold.

- Tools, information, special offers, demos, lots more source code, and lots of games. Stick the disc in your drive and click on what interests you. Start with the Read Me First! file to find the latest data on what's included.
- A custom index to the entire CD-ROM. Just double-click on the Index of Contents icon to get a description of everything that's on the CD-ROM.

One final note: Hayden will be offering other books of interest to Macintosh game developers. If this book helps you develop a new game, we would like to include your game (or a demo of it) on the CD-ROM of the next book in the series. Contact the Project Editor at `MacGameDev` on AOL (`macgamedev@aol.com` for other service providers). We look forward to hearing from you!

Check out the
Mac Game Programming Gurus
Web site at:
http://www.mcp.com/hayden/mac_game-gurus

A WORLD-CLASS GATHERING OF GURUS!

Jamie McCornack, Project Editor, Author and Coordinator. Jamie has written numerous game development articles, and is editor/publisher of the online game development magazine, The Macintosh Game Developer.

Ingemar Ragnemalm, Author. Ingemar works at the Swedish public service television company designing and programming software used in entertainment shows. He has released numerous titles for the Mac, including the Sprite Animation Toolkit programming library.

Paul Celestin, CD Development. Paul is president of Celestin Company, Inc. which has produced the CD-ROMs Apprentice, a collection of high-quality source code and utilities for Mac programmers, and Destination: Internet, a collection of Mac Internet resources.

Steve Hales and Jim Nitchals, Audio Chapter. Steve has been a contractor for most major Mac game publishers. He licenses SoundMusicSys, a sound driver he and Jim developed, which is used in over 30 commercial games. Jim works for Apple computer, currently on QuickTime sound and music software.

Jon Benton, Graphics. Jon is a senior developer working with Presage Software. Jon has worked on various projects including a kids' gamepack developed for MacPlay, called Mario's Game Gallery.

Mick Foley, Advanced Graphics. Mick works at Presage Software and has worked on such games as SimCity, SimAnt, SimCity 2000.

Brian Greenstone, QuickDraw 3D™ and Power Mac. Brian currently works for Mindscape and has worked on 6 Super Nintendo games, 3 Mac games, and 3 Apple II games.

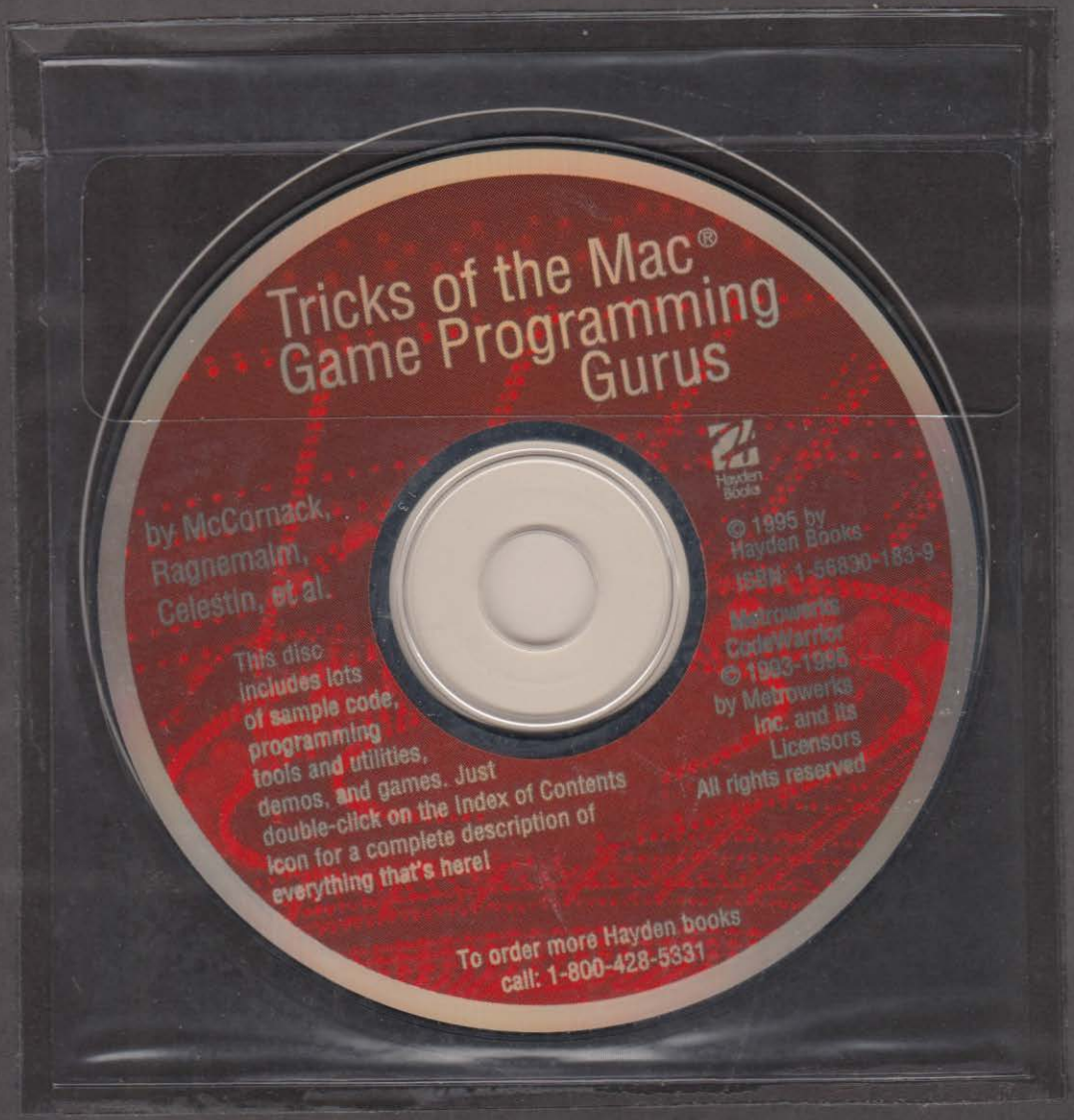
Roy Wood & Stuart Cheshire, Networking. Roy is working on MacConquest, a full-color multiplayer network game for use on the OutLand online gaming network. Stuart Cheshire is well known for his excellent network game, Bolo.

Doug Grounds, Porting. Doug founded and is the President of Lion Entertainment, Inc. He is the producer and technical advisor on Super Wing Commander and Doom II for the Mac. He has worked at Binary One, New World Computing, and Strategic Simulations where he programmed and produced many games for Sega Genesis, Macintosh, and 3DO environments.

Eric Johnston, Porting. Eric Johnston is a programmer at LucasArts Entertainment. He has worked on all of Lucas Arts' Mac games to date, as well as games on some other platforms. He developed Rebel Assault, 3 Indiana Jones titles, 2 Monkey Island titles, Loom, Pipe Dream, and Putt-Putt Joins the Parade.

john calhoun, Game Aesthetics and Glypha. john has commercially published Glider 4.0 with Casady & Greene, Inc., Pararena 2.0, More Glider, Spaceway 2000, Silicon Casino, and Glider PRO. He assisted on the graphics on Mission: Thunderbolt; and the graphics, sound and some code in ZOA and ZOA II.

The Tricks CD-ROM



TRICKS OF THE MAC GAME PROGRAMMING GURUS

Master the Techniques You Need to Create Your Own Games

This is *the* book for game programmers. With comprehensive coverage of the most important topics, this book provides the information and the sample code you need to develop game programs! You'll get the rare opportunity to learn from the experts, including special interviews with well-known Mac game programmers!

"As Apple's Game Evangelist, I highly recommend this book to anyone who is developing a Macintosh game or wants to know more about game development. With some of the best Macintosh game developers contributing to this book, it's material I keep right next to my Macintosh!"
-Eric Klein, Jr.
Game Evangelist—Apple Computer, Inc.

"This chapter alone is worth the price of the book."
(Advanced Graphics)

"If I'd had this kind of information at my fingertips when I wrote Bolo, it would have been finished in half the time."
—Stuart Cheshire

"Excellent advice that's easy to implement and yields great performance gains." *(Power Macintosh)*

"The best info available on QuickDraw 3D."
(QuickDraw 3D)

Technical Reviewers— *Brian Greenstone, President —Pangea Software; Jake Hoelter, Executive Producer—Presage Software; Dennis Cohen, Sr. Software Engineer— Claris Corp; Stuart Cheshire, PhD Student/Research Assistant, Stanford University; Mark Adams—Macintosh Game Developer, LookingGlass Technologies*

Learn about QuickDraw™ 3D, the m significant new technology for gam programmers!

Port your games to and from other platforms

Understand how to incorporate sophisticated graphics into your ga

Incorporate networking techniques into your own games

Learn how to optimize games for Power Macs



CD-ROM includes ...

CodeWarrior™ Lite— enables you edit, compile, and run the included code

Lots of great programming tools an utilities

Glypha III—with fully commented source code to help you understan how this popular game was programmed

All the source code explained in th book

And more!

$50.00 USA / $68.95 CAN / £46.95Net UK (inc of V



Hayden Books

The Don Crabb Macintosh Library

Category: Macintosh Programming
All User Levels



New Casual Accomplished Expert

Requirements: System 7.0 or greater, CD-ROM drive, 4 MB of RAM

ISBN 1-56830-183-9

