SERIES EDITOR SCOTT KNASTER

Macintosh Inside Out

Programming QuickDraw™

Includes Color QuickDraw™ and 32-bit QuickDraw™

DAVID A. SUROVELL
FREDERICK M. HALL
KONSTANTIN OTHMER

Programming QuickDraw™

Programming QuickDraw™

Includes Color QuickDraw™ and 32-bit QuickDraw™

David A. Surovell, Frederick M. Hall, and Konstantin Othmer



Addison-Wesley Publishing Company, Inc.
Reading, Massachusetts Menlo Park, California New York
Don Mills, Ontario Wokingham, England Amsterdam Bonn
Sydney Singapore Tokyo Madrid San Juan
Paris Seoul Milan Mexico City Taipei

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial capital letters.

The authors and publishers have taken care in preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Library of Congress Cataloging-in-Publication Data
Surovell, David A.
Programming QuickDraw : includes color QuickDraw and 32-bit QuickDraw / David A. Surovell, Frederick M. Hall, and Konstantin Othmer.
p. cm. -- (Macintosh inside out)
Includes bibliographical references and index.
ISBN 0-201-57019-X (p) : $26.95
1. Macintosh (Computer)--Programming. 2. Computer graphics. 3. QuickDraw. I. Hall, Frederick M. II. Othmer, Konstantin. III. Title. IV. Series.
QA76.8.M3S86 1992
006.6' 76--dc20 91-40339
CIP

Copyright © 1992 by David A. Surovell, Frederick M. Hall, and Konstantin Othmer

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America. Published simultaneously in Canada.

Sponsoring Editor: Carole McClendon
Technical Reviewer: Tim Cotter
Project Editor: Joanne Clapp Fullagar
Cover Design: Ronn Campisi
Set in 10.5-point Palatino by CIP

1 2 3 4 5 6 7 8 9 -MW- 9594939291
First printing, December 1991

Dedication

This book is dedicated to our parents:

Edward and Barbara Surovell
Robert and Ann Hall
Ekkehard and Sieglinde Othmer

Contents

Foreword by Scott Knaster xiii
Acknowledgments xv
Introduction xvii

1. **Graphics Programming Overview 1**
Introduction to Graphics Environments 2
Personal Computer Graphics: A Brief Look Backward 2
Why Is Graphics Programming Difficult? 4
Introduction to Macintosh Graphics Environments 4
What's in the Toolbox? 7
QuickDraw 9
Doing It Yourself: The Path Not Taken 10
Defining a Practical Programming Approach 12
Naming Conventions 14
A Nickel's Worth of Objects 15
Use of Atomic Data Types 20
Important QuickDraw Clients 20
Underneath QuickDraw 21
Summary 22

2. **Basic Drawing Technique and Theory 23**
Raster Graphics 24
Digital Images 26
Points 26
BitMaps 28
Basic Drawing Operations 29

Using Shapes 29
GrafVerbs 30
Pens and PenStates 33
Lines 38
Rectangles 40
Rectangle-Based Shapes 41
Polygons 53
Arcs and Wedges 55
Basic Text 57
Pictures 65
The GrafPort 69
Boundaries and Coordinate Systems 73
Visible Area 75
Clipping 76
Bottleneck Procedures 77
Summary 78

3. Drawing in Color 81
Software Components of Color 82
Graphics Toolbox Organization 82
Multi-Bit Pixels 88
Color Models 89
Indexed Color 89
Direct Color 90
Fixed Color 92
Classic Color 92
Basic Color Theory 93
RGB Color 93
Chromaticity and Primary Colors 94
RGB Color Values 97
Limits of Color Display 99
Color Picker 100
Color Picker Dialog 100
HSV Colorspace 103
CMY Colorspace 104
ColorTables 104

Graphic Devices 106
Using the Graphics Device Manager 107
Color Mapping 113
Color GrafPorts 122
Pen Colors 124
Transfer Modes 127
Digital Images 135
PixMaps 136
PixMap Data Structure Description 137
PixMap Management Traps 140
PixMap Management Strategies 141
Creating Offscreen PixMaps 142
Color Patterns 146
Using PixPats 147
CGrafPort PixPats 147
Creating Custom PixPats 148
Moving Pixels Around 148
Introducing CopyBits() 149
Horizontal and Vertical Scaling 151
Examples of Using CopyBits() 152
Performance Considerations 157
PixMap Resolution and Patching the StdBits() Bottleneck 160
Summary 163

4. Display Control and Offscreen Drawing 165
The Need for Drawing Offscreen 165
Video Display Overview 166
Causes of Flicker 166
Basic Offscreen Drawing Procedure 168
GWorlds 170
Using GWorlds 170
More GWorld Techniques 173
GWorld Usage Summary 178
Application Display Control 178
Rules about Changing GDevice Settings 178
Using HasDepth() and SetDepth() 179

The Palette Manager 180
Basic Palette Concepts 181
Important Palette Manager Traps 187
The Dark Past of the Palette Manager 190
Using Palettes 191
Palette Review 201
Specialized ColorTables 201
GDevice ColorTables 203
User Interface ColorTables 203
Summary 204

5. Applied Graphics Techniques 205
Building a Paint Application — ArtToy 205
Objectives and Goals 206
Software Tools Needed 206
Window Fundamentals 207
Graphic Foundations of Windows 208
Window Regions 209
Window Types and Presentation Considerations 213
Views 214
Focusing 218
Basic Window Event-Handling Techniques 221
Desktop Interactions 224
Introduction to Document Organization 227
Rasterizing 228
Document Rendering Techniques 229
Printing 230
Overview of the Printing Process 231
Print Data Structures 232
Important Print Traps 233
A Basic Window Object 234
Objectives 234
Class Definition 235
Public Methods 238

A GWorld Window Object 243
Objectives 243
Class Definition 243
Public Methods 244
Summary 247

6. Mouse-Based Graphics 249
The Two Basic Types of Mouse-Based Tools 250
Sketching 250
What All Mouse-Based Objects Have in Common 253
Drawing Tools and Abstract Differentiation 256
Regular Shapes 256
Irregular Shapes: Collected Points 270
Selection Tools 284
Summary 310

7. Image Processing with QuickDraw 311
Pixel-Processing Traps 312
Basic CopyDeepMask() and CopyMask() Functionality 314
Explicit Arguments 314
Global Arguments 317
Altering Images with CopyBits() 320
Aliasing 326
Rasterizing 328
Aliased Regions 331
Antialiased Lines 332
Antialiased Text 334
Review of Key Points 338
Depth Conversion and Dithering 338
Depth Conversion Cases 338
What Are Dithering and Error Diffusion? 341
How CopyBits() Dithers 342
Image Resizing 343
Dithered Patterns 348
Using Deep PixMaps on Classic QuickDraw Machines 349

The BitMapToRgn() Trap 350
Device Loops 352
The DeviceLoop() Trap 353
Using DeviceLoop(): A Source Code Example 354
Summary 355

8. All About Pictures 357
Picture Analysis with the Picture Utilities 358
Taking Pictures, in Review 359
Using the Picture Utilities 361
Count of Basic QuickDraw Objects 364
Color Analysis 366
Using Deep PixMaps on Classic QuickDraw Machines 376
Summary 384

9. Compatibility and Configuration Issues 385
Hardware Configuration 386
The Well-Behaved Application 387
Software Compatibility 393
Detecting and Handling Limited System Software 394
Imposing Configuration Requirements 396
32-Bit Addressing Issues 397
A Short History 398
Addressing Video Display Memory 399
Handling 32-Bit Addressing Mode 400
Other Issues 403
Summary 404

Bibliography 405

Index 411

Foreword by Scott Knaster

Macintosh programmers are continually assaulted by all the graphics stuff that they get to work with. QuickDraw, in its Classic, Color, and 32-bit varieties, Color Manager, Palette Manager, graphics devices, color picking, and on and on, all have their part in making life interesting. On the Macintosh, graphics isn't just something you do for fun — it's at the heart of every application, and if you don't treat it that way, your users will know.

Early Macintosh programmers knew how important graphics was to the system. Of the original Macintosh's 64K bytes of ROM (which was an incredibly generous amount in 1984), a full third of it (or so) was devoted to QuickDraw. As Apple's rivals figured out that the Macintosh's graphical interface was probably a good idea after all, it was QuickDraw that helped the Macintosh hold its technology lead for years.

In 1987, when Apple changed the Macintosh from a product to a product line by introducing the Macintosh II with its slots and color monitors, QuickDraw was extended to do nifty things like draw in millions of colors and onto several monitors together at one time. (As some of you may know, the original QuickDraw could draw in eight colors, but a lack of output devices made that a fairly well-kept secret.) In its transmuted form, QuickDraw remained at the heart of the Macintosh's magic. The Macintosh graphics world was, however, getting significantly more complex.

The next couple of years brought the release of the precocious and awkwardly named 32-bit QuickDraw (although it had a very hip code name, Jackson Pollock, taken from the color-splashing artist). Now

you truly couldn't tell the QuickDraws without a road map. The Macintosh's graphics capabilities had expanded vastly, and those who hadn't kept up were surprised and amazed by what was available.

In an effort to help you make sense of all the graphics machinery that Apple has poured into the Macintosh, Dave, Fred, and Konstantin have assembled this handbook, which can act as your map into the Macintosh's graphics highways. I don't want to stretch a metaphor until it breaks, but they've definitely been down these roads, spending years making pixels glow and dance before they decided to write it all down. Keep this book with you as you explore the Macintosh's graphics and you'll have a deeper understanding of what's happening and where you might wind up.

Scott Knaster
Macintosh Inside Out Series Editor

Acknowledgments

In the course of writing this book, we received assistance from numerous people without which this book would have been impossible.

The people at Addison-Wesley were all very pleasant and helpful. In particular, we'd like to thank Joanne Clapp Fullagar, our tireless and patient editor, who kept the project moving and helped us fight through all the little details. Thanks to Carole McClendon for cheering us on and keeping us moving forward. A kind tip of the hat to Rachel Guichard for handling the day-to-day things. Finally, thanks to Vicki Hochstedler for guiding this book through the production process swiftly and gracefully.

We wish to thank Bob Royce, our illustrator, who was both patient and tireless in creating most of the artwork in this book.

Thanks are in order for Tim Cotter, whose technical review helped set us straight on a number of technical points in the manuscript.

We would like to thank Steve Mockensturm for the use of his rendering of the Les Paul guitar and other odds and ends used in the color plates.

The companies Acuson, MedImage, and Apple Computer, Inc., where we have worked at one time or another during the preparation of this book, deserve thanks for their support of our efforts.

David Surovell thanks his close friend Cynthia Wenzel, who gave tremendous support and encouragement throughout the entire course of the writing effort.

Fred thanks Maggie Bayless for her love and support.

David and Fred thank Patrick Leonard, who has been a fine friend and mentor to the both of us, sharing with us much wit and wisdom over the years.

Konstantin thanks Bruce Leak for 24-hour road service and expertise; Chris Derossi for generating the picture of colored balls and for 24-hour billiards losses; Peter Hoddie for assisting with the sample code in Chapter 8; and David Van Brink for the picture of the squishy, slimy buttons.

Thanks to Adam Lavine of Specular International for Infini-D which was used to create portions of the color plates, Joe Williams and Tim Cotter of Delta Tao for Color MacCheese which was used extensively for creating the color plates, and Georgiann Delaney of Apple Computer for writing BitBlitz which was used to generate some of the images on the color plates.

The manuscript for this book was created using the following software:

FullWrite Professional	Borland/Ashton-Tate/A2 SoftWorks
MacWrite II	Claris
FreeHand	Aldus

Introduction

When we began working on this book, we heard "I'll buy it, I'm looking forward to reading it" from a number of other good Macintosh software engineers. We found it rather puzzling: Many (perhaps most) of these people we consider to be exceptionally capable; some of them are so talented it scares the daylights out of us. It's hard for us to imagine that we could inform them about *any* aspect of the Macintosh, much less the user interface engine itself!

The plain truth is that there are only a few programmers out there who know how to hack the Macintosh's user interface in its entirety. Many people who know a few of the Toolbox "managers" inside out are totally naive about many of the others. This also holds true for QuickDraw and the subset of graphics-related Toolbox managers.

Computer graphics programming is a vast discipline. To even attempt to write a complete book on the subject would be ridiculous. The comparatively small sub-domain of Macintosh graphics programming is still large enough. When the Macintosh was introduced in 1983, QuickDraw was a nice, compact set of graphic primitives tuned for speed in a black-and-white world. Over the years, it has become a high-powered, but sprawling graphics environment, implemented across an entire range of microprocessors and memory configurations, while maintaining a flexible and consistent software interface over a 6-year period, facilitating simultaneous display across multiple monitors, and more. System 7 brings to the Macintosh another batch of new graphics programming tools, along with significant changes to some of the existing ones in the preceding versions of QuickDraw.

Becoming comfortably conversant with such a large body of software is difficult. Not to say that it's as bad as learning to play the bagpipes, which I've heard takes about 2 ½ years before you can play the Scottish equivalent of "Mary Had A Little Lamb." But big it is, and bigness is intimidating. Many of the difficulties that we've experienced are the result of an incorrect choice of methods amongst the many available in the Macintosh graphic programming environment.

One major problem that many developers encounter when undertaking a Macintosh programming venture is a lack of access to needed materials. Those people who work for or with firms who are "official" Apple software developers have easy access to current programming materials and reference information that other programmers can't easily obtain or don't even know about. This book won't replace such services as the Tech Notes, AppleLink, Macintosh Developer Technical Services (Mac DTS), Apple Programmer's and Developer's Association (APDA), *develop* magazine, and so forth. But we have tried to distill the more important information that we've gathered from these sources and combine it with the reference presentation of *Inside Macintosh* into a concise and accurate approach to graphics programming on the Macintosh. Treat it as a companion to, rather than a replacement for, those "official" materials to which you may already have access.

▶ Scope and Focus

It's a struggle to keep the discussion and materials focused narrowly enough to learn any facet of Macintosh graphic programming, and at the same time demonstrate and encourage sensible programming tactics. Due to their computation-intensive nature, interactive graphics systems push the underlying hardware and software systems to the limits (and occasionally beyond); this necessitates some discussions of the form "Well, you COULD do it that way, but...." Another major theme is that the presentation of Macintosh graphics programming covers many areas of the Mac that don't have a "Quick" in them.

▶ Of Theory and Practice

We believe that the success of this book depends on how well we keep the average reader (that is, a sensible programmer with not much QuickDraw experience) in mind. Except there is no "average reader." What we *do* expect is that most of you have work to do, whether the work is job-oriented or playful curiosity. Learning to drive Macintoshes

graphically from the inside is hopefully a means to an end, rather than an end unto itself. We also believe that a sensible balance of theory talk and practical workshops are the cleanest, fastest way to understanding the Macintosh's graphic world. We know that people don't learn to program by reading all of *Inside Macintosh* before writing a single line of code, therefore we've tried to take a "show and tell" style of presentation, augmenting technical prose with a number of illustrations and source code examples. We hope that you find this approach to be practical, while giving you a solid understanding of Macintosh graphics programming. We'll try to minimize the rote memorization common to solving programming problems.

▶ Modularity

You can treat this book as a tutorial and read it front to back. However, you may prefer to skip around and read those sections first that you find most intriguing. We expect that you will be able to do so without a significant loss of continuity or technical comprehension. Some chapters are a bit code-heavy, others may seem like too much talk, and so we urge you to go back and forth between them to synthesize the intent and discipline of QuickDraw programming. Feel encouraged to skip around while reading this book. The technical foundations of Macintosh graphics programming are covered in the first four or five chapters; the remaining chapters deal with advanced and applied topics. Tackling QuickDraw concepts individually is crucial to a thorough conceptual understanding and solid practical style. Much of the object-oriented programming debate is concerned with creating small yet functional modules that become the building blocks upon which large, complex applications can be built.

▶ Source Code Disk

In the process of writing this book, we found that space limitations would prevent us from including a lot of useful routines and other source code examples. We have therefore compiled a source code disk which includes the following materials:

- all source code in the book
- the complete ArtToy paintbox application source and resources, including many familiar sketching and selection tools
- GrafPort and GDevice management routines

- "useful" region generation routines
- regular polygon generation routines
- color pattern ('ppat') resizing and resource I/O routines
- color palette animation examples

The disk can be obtained by sending the coupon at the end of the book and a check to the authors.

▶ What's Missing

With such a large body of material to cover, it's virtually inevitable that some elements of Macintosh graphics programming have been omitted. There are nuances to even the most basic elements that may have been left out; for example, we could have written a couple of chapters on color pattern handling alone. Still other important topics have yet to be addressed fully by the Macintosh graphics programming environment. The use of alpha channel imaging techniques and matching color specifications to hardcopy output are examples of such topics. Finally, there are a number of printing topics that are related to the QuickDraw-to-PostScript translation that occurs when generating hardcopy output from LaserWriters and other devices that don't use QuickDraw for graphics generation and display.

▶ The End of the Beginning

We hope that you will find this book to be useful and enjoyable, and that it spends the rest of its days as a dog-eared companion next to the Macintosh where your development is undertaken!

1 ▶ Graphics Programming Overview

Only a programmer who is used to a world of bitmaps and single nine-inch screens could look at a programming environment of 32-bit colors and multiple thirteen-inch displays and think of it as wide open and liberating. Compared to the way human beings interact with each other, our medium is still profoundly limited. But if we understand the medium and the ways we can exploit it, we can enhance the quality of the information exchanged between Macintosh computers and Macintosh users.

You'll find this book to be a source of both basic and advanced information about Macintosh graphics programming. In it we'll try never to stray far from practical applications of the material being covered. But before diving right into coding and the specifics of Macintosh graphics programming, you should be familiar with a few fundamentals of graphics programming. It will help your understanding of QuickDraw if you have a sense of the motivations behind graphics programming choices and graphical user interfaces. We'll be discussing the levels of support that operating systems give applications. By *support* we mean the callable tools that are embodied in the operating system for rendering drawn objects on the screen. If an application can ask its operating system to "rotate the surface 30 degrees in X and 110 degrees in Y," then it's getting a lot of support. If the most the application can do is indicate which pixels are turned on and which are turned off, it should sue the operating system for negligence or lack of support.

In this chapter we'll describe graphics programming in general and relate it to the Macintosh graphics environment, QuickDraw. In doing so, we'll identify the important graphics features available to you in the Macintosh environment.

▶ Introduction to Graphics Environments

The range of personal computers, from the low end of the clones up through the high end of the design workstations, have this in common: They participate in a one-on-one dialog with the user, who is manipulating input devices. And, with very few exceptions, the output side of the dialog is presented to the user on a CRT display. How the computer does its job of presenting what it has to say is what gives it its personality and instills in the user a sense of either ease or frustration.

When the job is done well, the user is brought closer to his or her data; the operating system gets out of the way of the essential interaction, which is between the user and the information presented by the software. It is like the ideal sought by baseball umpires who, while being essential to the proper functioning of a baseball game, have done their job best if the players leave the field remembering the competition and not the umpires. Our goal as programmers and interface designers is to give users a sense of having interacted not with a computer but with the information it contained and manipulated for them. When the job of presenting information on a screen is done poorly, the computer is perceived as being less of a tool and more of an obstacle between the user and the data.

▶ Personal Computer Graphics: A Brief Look Backward

Every personal computer operating system has to express itself on the screen — computers with no output are boring at best. A computer's visible output tells the user what role the computer is playing: I'm a word processor; I'm a video game; I'm a VCR. The problem is that early personal computers didn't do that very well. Steve Jobs illuminated the problem when he compared computers to toasters. The role of a toaster is singular and well defined in terms of input, output, and control. The purposes of a personal computer, however, are numerous and generalized — the evolution of their respective user interfaces demonstrates this all too clearly.

Most of the familiar personal computer systems came bundled with a family of programs that comprised their operating systems. Most such operating systems lacked substantial visual presentation support.

Why? Perhaps because the early personal computer operating systems reflected the tasks computers had always been used for. The emphasis was on problem-solving with numerical results; presenting

the results was less important than producing the results. Word processing, for example, dealt in rows and columns of text characters. One could have argued that you didn't need graphical controls to manipulate and present essentially alphanumeric input and output. Graphics programming had always been the province of specialized software used by people whose needs were essentially graphical: model designers and image processors.

If you take these views of the nature of computing, then it might not occur to you to build graphics support into the very operating system itself.

Of course, price was another factor. Cheap, slow microprocessors couldn't draw and maintain fancy graphics and still have time to do the "important" work of "data processing."

With little graphics support built into the operating system, each application was responsible for its own graphics support. When such systems came out of the basement, they became anything but toys and problems showed up quickly. When applications did their own graphics, it was tough to enforce consistency, and consistency is important in the little things like buttons as well as in the big things like colors.

Unfortunately, these operating systems were also quite restrictive. The user's perception of the typed, command line interface was that the thread of control was limited to a tight call-and-response pattern in which things happened one at a time. Commands, once initiated, simply ran to completion or error, whichever killed them first.

To someone who grew up using Turing machines or punch cards, a command line interface is a big step up. The success of personal computers that used command line–oriented interfaces argues that they were not sufficiently bad to keep people from using them. But the move to graphical user interfaces argues that they were an evolutionary step toward the ideal user interface (whatever that turns out to be). Users perceive a graphical interface as letting them broaden the scope of their focus to include not just a typing prompt but the whole screen. The graphical interface can offer more options while presenting more information than is possible with a command line interface.

From the first, the Macintosh made graphics support a big part of the operating system. Placing QuickDraw in the Toolbox allowed the Macintosh to take its celebrated leap in personal computing. It is a design philosophy that has withstood the test of time.

The popularity of mice and other graphically oriented input devices are important additions to the world of personal computers. They also present challenges to software developers, demanding a new level of application responsiveness and user interface creativity.

A graphically oriented user interface is a natural consequence of the desire to make computers function in "human" ways, processing short bursts of regular tasks. Many consider the Macintosh to be the first commercially successful microcomputer to solve the dilemma of delivering complex functionality while providing ease of operation.

The bad news is that supporting such an environment places greater demands on the applications that run on the Macintosh. The good news is that QuickDraw is there to help.

▶ Why Is Graphics Programming Difficult?

Maybe the reason that early personal computer operating systems lacked substantial graphics support is that graphics programming is fairly demanding, both from a hardware and from a software standpoint. What seems simple to the uninformed can be very complex to implement.

When an application makes a request as basic as, "Draw the string 'HELLO, WORLD' on the screen," there is a lot that is either specified or implied. Where shall it be drawn? In what style, size, and color? How does it get mapped into screen pixels? Where in memory are the pixel values stored? How shall the current values of those pixels be taken into account? If the line is too long to fit on the screen, should it be clipped, wrapped, or shrunk?

You can see that the fewer options offered to the application by the operating system, the simpler its job. But that avoids the problem instead of solving it. Leaving graphics implementation up to each application introduces the problems of inconsistency. The real solution is to build the graphics solutions into the operating system, as was done with QuickDraw.

▶ Introduction to Macintosh Graphics Environments

By offering services to applications, built-in graphics programming support insulates the applications from the low-level implementation detail necessary to effect the drawing that the applications direct. And this insulation works both ways. By isolating the applications from shared, systemwide resources like the graphics devices and their data structures, the operating system protects those resources from abuse at the hands of ignorant or careless applications. This is the essence of a graphics environment.

With QuickDraw built in as part of the Toolbox, the Macintosh provides a graphics environment to the applications that run on it. The environment is accessible to the applications through QuickDraw calls, but the integrity of the environment is maintained by QuickDraw. (This state of affairs is the ideal, of course, and will never be fully achieved in practice; whether through ignorance or malice, you can still pollute the environment.)

With a graphics environment built into the operating system, graphics programming resembles other embedded services (like disk I/O) in two important respects:

- The variables that define the state of the environment are maintained by the operating system rather than by applications. Access to those variables and data structures is restricted by the operating system. Direct manipulation of variables by the application is replaced by information requests via operating system function calls.
- The burden of error checking must be assumed by the application. An application must be able to deal with a large number of possible error conditions and combinations of conditions encountered by the operating system because the operating system can only tell the application what happened; it can't make decisions about what to do next.

These aspects of the graphics programming environment go against the do-it-yourself style of some personal computer programmers; these conditions are sometimes perceived as being impediments to software development rather than a welcome relief from certain tedious programming responsibilities. Some microcomputer applications are written as if they were intended to replace the operating system rather than to coexist with it.

Important ▶

Some programmers chafe at the idea of restricted access to data structures in memory. If you think you have the need, you're free to make your application read (and in some cases, write) QuickDraw data structures directly. But there are risks associated with direct manipulation of system variables: They aren't guaranteed to be where you expect to find them. If you're running under System 7.0, A/UX, or something that installs virtual memory and/or privileged memory access protection via PMMU protection, you may find your hacks will stop working. Bear in mind that a little bit of restraint — a little bit of conformity — allows many applications to coexist peacefully on a single machine. The cooperative multitasking environment of the Process Manager (formerly MultiFinder) is a prime example of this kind of arrangement.

The requirements of graphic presentation, with the multiplicity of modes, control variables, and data types, becomes unwieldy without the support of the operating system. If you didn't have the operating system maintaining the graphics environment for you, you would have to redefine the environment with every call, passing arguments to describe the requested drawing in detail. For the kind of graphics programming required of an application in support of the Macintosh user interface, there's too much baggage to carry around through every function call. Simply writing and debugging the applications would get out of hand.

Drawing doesn't take place in a vacuum. Rather, the act of drawing is a sequence of actions that takes place within the graphics programming environment. The environment defines and constrains the drawing actions, just as a pencil and piece of paper are a graphics environment to a scribbling child. At first, maybe, the child just abuses the environment — tearing the paper with rough strokes, drawing on the tabletop, eventually inciting parental intervention. But with experience the child learns to keep the pencil on the paper, even attempting the occasional flat-perspective two-dimensional pastoral landscape shown in Figure 1-1.

In some ways you're like the child learning to draw. With a little practice and discipline, QuickDraw will become both an environment in which you can let your application express its true nature and a valuable tool for the user.



Figure 1-1. Child's sketch

▶ What's in the Toolbox?

Unlike a scribbling child, you probably approach your software problems from a generalized, abstract point of view. You shouldn't have to reinvent the wheel to do the graphical rendering required by your application. You can reasonably expect a graphics programming environment to provide you with certain capabilities:

- **Drawn objects** — You can't have graphics without drawing. An application needs simple atomic objects, such as points, lines, characters, and simple geometric shapes. Graphics programming requires simple methods for drawing familiar user interface objects such as menus, controls, and windows. You need the ability to develop application-specific graphic objects.
- **Basic calculations on graphic objects** — Preparing graphics for display requires functions to calculate the area in which the graphic will be displayed. The code to perform these calculations makes up the bulk of graphics programming. It usually exceeds lines of actual rendering code by a factor of 3 to 1. The graphics environment must provide at least one common unit of measurement, for example, the number of screen pixels. A number of measurement

units is desirable, along with routines to convert between them. Since most computer output is of a two-dimensional nature, you need routines that can calculate screen area in terms of basic set theory operations such as union, intersection, and complementation. You probably also need the intermediate set theory operations as well, such as differences and exclusive or-ing.

- **Full control** — You need the ability to control all drawing parameters that affect the important aspects of your application. Implicit in such control is the ability to sense the environment to detect and respond to the present device characteristics as configured by the user.

The paradigm espoused by the Macintosh designers is that you, a card-carrying member of the Hackers, Contractors, and Occasional Employees Union, have work to do, and they're providing you with a toolbox. A quick inventory of the QuickDraw part of the Toolbox includes the following basic tools: a boundaries and coordinate system answers questions like, "Where is 'right here'?" and knows what you mean when you say, "Draw a line from right here to over there." It provides line and curve drawing, pens and patterns, and text and simple geometric shape drawing. There's also color and pixels, pixel values, and display color mapping. It can transform numeric values into colors, or colors into other colors. It even supports you when you need to do printing and imaging on other, nonvideo devices.

Important ▶

It's possible that none of these outlooks quite describes the way that you conceptualize the tasks and requirements of graphics programming. That's OK. The Macintosh graphics environment accommodates a number of imaging techniques.

Perhaps you're new to graphics programming and you don't have any idea of what graphics programming is about. Start simply and build from what you learn. Don't expect it to happen overnight. What is important is that you eventually evolve some model, some way of thinking about graphics programming.

The Macintosh graphics environment has a lot in common with other familiar graphics environments. After all, they're doing basically the same job. Most of its departures from what might be thought of as standard graphics environments are in its support of multiple, concur-

rently executing applications. Most graphics environments were designed for single-user systems dedicated to graphics production. They therefore have little need for the capabilities that most Macintosh users and programmers take for granted — like window and desktop management.

▶ QuickDraw

The reason that the Macintosh is a dramatic step forward in popular computing is due largely to its user interface, which is implemented by groups of routines that are called *Toolbox managers*. There is a manager or a group of managers that implements virtually every functional aspect of the Macintosh, from the familiar visual objects like windows and menus to the not-so-visible NuBus devices (Slot Manager) and AppleTalk. Depending on how you count, there are from five to ten managers that implement Macintosh graphics capabilities. The most recognized name among Toolbox managers is QuickDraw.

A simple description of QuickDraw is that it is the Macintosh's graphics software engine. Its purpose is twofold: to drive the Macintosh's celebrated user interface and to provide applications with a solid foundation of graphics functionality. Virtually every Macintosh graphic object is either directly or indirectly drawn with a series of QuickDraw calls. Menus, controls, pictures, and text are all drawn with QuickDraw. There are few exceptions to this rule. (One notable exception is LaserWriter printing, which is driven by PostScript. We'll get to that in Chapter 3.)

QuickDraw provides Macintosh programmers with a number of important advantages over other graphics environments and hardware platforms:

- **Well-defined specifications** — QuickDraw is based on a published software specification — rather than a hardware design — that has a number of significant implications. The most important is that it allows for portability to future hardware with a minimum of software disruption. In other words, running with QuickDraw doesn't depend on its implementation, only its specification.
- **Compatibility** — QuickDraw has proven to be a fairly stable specification, endowing applications with the ability to stand the test of time without substantial modification to the underlying software architecture. While the Macintosh has evolved substantially

from the first 128K version, QuickDraw has continued to support those applications that were designed to use it in the old days while it has evolved to provide substantial new features.

- **Hardware independence** — QuickDraw encourages applications to take advantage of hardware improvements while insulating them from the idiosyncrasies of display devices and the strange combinations of devices that users often put together.
- **Full-featured graphics capability** — Besides supporting "basic" graphic operations, QuickDraw provides a number of tools to perform complex graphic analysis and manipulation.
- **Optimized for performance** — You might think that this assertion contradicts the compatibility arguments, but it doesn't. Although QuickDraw was designed to take advantage of Motorola's 68000 family of CPUs (the architecture of the 68000 family is extremely well suited to graphics processing), it would be difficult if not impossible to pinpoint QuickDraw design concessions to the processor.
- **Open-ended design for extensibility** — QuickDraw was designed to be extensible. The designers knew that they couldn't foresee all of the directions in which personal computing technology might go, so they designed QuickDraw to do its job without precluding future enhancements. So far, they've been rewarded with a robust, vital piece of software.
- **Customization** — QuickDraw, like many aspects of the Macintosh operating system, contains hooks on which an application can hang its own special-purpose code for its own way of doing things. At the points where these customized extensions can be installed, QuickDraw looks to see if there is a special way of doing something before proceeding to do it the QuickDraw way.

▶ Doing It Yourself: The Path Not Taken

Some programmers prefer to adapt new software environments to their own style of programming, bringing with them command scripts, utility function libraries, and other development tools. That's understandable. People like to do things in familiar ways. There can be anxiety attached to leaving an environment in which you feel comfortable and productive to go to a new one where you have to learn new ways of doing things. When you first start programming the Macintosh, you may be tempted to develop and use your own graphic function library instead of QuickDraw and the related managers. Particularly if you're porting an existing program from another envi-

ronment to the Mac, you might opt for a do-it-yourself approach. Do yourself a favor: *don't*. The effort required to adapt to QuickDraw will be repaid in robustness.

There are some arguments against doing things with QuickDraw, but we think that they're generally either overstated or rooted in misconceptions:

- "QuickDraw is hard to transport to other environments like X Windows, Motif, OPEN LOOK, SunView, Presentation Manager, and so forth."

  If you have to port, then you've probably planned on separating the user interface and the operating system–dependent code anyway. To begin with, user interface support is one of the least transportable kinds of software. The organization of your software, rather than the Macintosh's graphic architecture, is more likely to determine the difficulty of your port. Porting is usually a pain anyway, and QuickDraw doesn't make it worse. (Remember that most laser printers for the Macintosh render their graphics in a PostScript environment.)

- "Apple owns QuickDraw and I'd rather have my own graphics package, with 100 percent of the source code."

  Having access to the source code for your graphics environment may give you a warm feeling, but is it really that important? The difficulty in staying compatible with future releases of Macintosh software would cost you time and resources that could be spent more profitably on your applications.

- "What if Apple changes QuickDraw? I can't afford to be fixing my software every time a new System software release comes out."

  This is the flip side of the preceding complaint. The problem of future compatibility can never be eliminated, only reduced. The ongoing enhancements and improvements made to System software, along with the addition of new models to the Macintosh line, make the task of compatibility even greater. Generally speaking, Apple's track record at minimizing compatibility problems has been good. It understands the panic that ensues when new software and hardware releases break existing software applications and third-party hardware. Having developers, customers, and support staff in revolt isn't in anyone's interests, and Apple makes substantial efforts to provide developers with the tools to avoid such difficulties. You're better off with QuickDraw than without it.

- "QuickDraw isn't fast enough. It won't perform up to my applications' requirements unless I get faster hardware."

  If you really need to develop custom image-processing algorithms, there's no need to pitch QuickDraw entirely. There are substantial well-defined facilities for direct access to image memory. You can find examples in some of the mathematics and statistical software packages currently running on the Macintosh to appreciate QuickDraw's solid high-octane performance. But if you need even greater performance, you answered your own question: Buy some hardware. Besides, that's what hardware's for. If you're doing rocket science, you're going to have to buy some rocket technology! If your application is useful and powerful, your customers may find that enough of an inducement to make the hardware investment.

Any of these objections can be debated at length. The bottom line is that QuickDraw performs acceptably well by most accounts. Do yourself and your users a favor and leave the basics to the graphics Toolbox. The odds of building a better mousetrap by yourself, and having it work against the mouse yet to be born, are against you.

▶ Defining a Practical Programming Approach

Writing code can be fun. If you want to keep it interesting as well as productive, you'd be well advised to set and follow some sort of coding guidelines, whether they're your own or adopted, formal or informal.

There are few, if any, software systems that provide powerful functionality while simultaneously insulating programmers and their applications from abusing that functionality. QuickDraw is no exception: It's surprisingly easy to pursue ill-chosen designs that will ultimately fail to provide reliable application performance. These pitfalls can be avoided, however. The need for solid, systematic development techniques is acute when writing graphics-intensive software. Things are tough enough without the added burden of sloppy code or incomprehensible formatting. We're going to skip an in-depth discussion of our methodologies, but the following is a list of the style elements that we think promote successful programming. Consider them guidelines rather than laws.

- **Design clearly, design simply** — Don't write tricky code when simple will do. Some programmers began optimizing their routines before successfully executing for the first time. First make it run, then make it run fast.
- **Use comments** — Do it for yourself, if for nobody else. Programmers often treat their own software as though it were a personalized work of art, never thinking for a moment that someone else will ever need to understand it at the source code level. But even if nobody were to ever work on your source files, you can usually save yourself a hassle by writing reminders to yourself in comments. Write down your thoughts about questionable code, keep revision histories, remind yourself about problems discovered at 4 A.M. Help the others who may have to read your code — one of them may be you.
- **Check error conditions** — If a routine can fail, then check the results after calling it. The most common reason many Macintosh Toolbox routines fail to perform under certain circumstances is a shortage of available memory. If you don't already do it, force yourself to check error codes — you'll get used to it.
- **Keep routines small** — Small routines have a number of advantages over large ones: They are easier to read, easier to print, and easier to understand and debug. They are also easier to rewrite, if necessary. Small routines also facilitate optimal hardware register utilization.
- **Be stylistically consistent** — Consistency makes it easier to make global changes. It enhances readability and understanding.
- **Use NULL/NIL** — Indicate an absence of valid data in your variables by assigning them invalid values (use –1 if NULL is a valid value). This applies to initialized or deallocated handles and pointers. Doing this helps your application to manage these allocations properly.
- **Track file I/O reference tokens *very* carefully** — If you open a file, don't forget to close it. Avoid closing files that you didn't open and really avoid closing files that aren't open. Use invalid sentinel values (–1, 0) for initialized or otherwise invalid file and device reference numbers.
- **Keep global variable references to a minimum** — Excessive reliance on global variables causes problems in direct proportion to the number of global variables allocated.

- **Plan for debugging** — If the task is complex, implement diagnostic routines *before* you debug. Once debugging code is in place, consider making it conditionally compiled, rather than just removing it. Plan for enough time to test and debug.
- **Understand and follow Apple's software development guidelines** — We know that other people's guidelines are annoying. No one likes to be told what to do, or how to do it, and Mac programmers are no exception. Still, when it comes to software reliability, an ounce of prevention is worth a ton of cure, and Apple's guidelines are usually prescriptions for the right kind of prevention. You may not always like the guidelines, and you can't always comply with them, but at least try to understand them.

▶ Naming Conventions

Gather any number of programmers together to discuss naming conventions and you're likely to start a war. While you don't have to fight about it, you should take one side or another: either use someone else's naming conventions or make up your own. And then follow them. Expending the effort to choose meaningful names for routines, variables, constants, and other programming language artifacts is an important investment in lucid, maintainable software. Don't sell your software short. The huge number of objects and operations available to Macintosh programmers makes the use of naming conventions even more important. Here are some ways to make names work for you instead of against you:

- **Avoid cryptic symbol names** — Some people don't like to use vowels in names. Some people take pity on their linkers and assemblers, thinking that their tools are less likely to get overworked and act up if names are kept short. But that's what these tools are built for — so six months later you won't have to wonder if "cPoau" means "pixel count of absolute underflow" or is a type of tropical bird.
- **Get similar** — Name similar things with similar names.

```
long  carLength, truckLength;     /* good */
long  carLength, truckHSize;      /* less good */
```

When naming local variables of a common type and purpose, use the same names. If you like to use single letters for indexing variables, don't use twenty-six different letters for twenty-six

indexing variables. It's easier to cut, copy, and paste code if you've used the same variable name in similar contexts.

- **Use common prefixes and suffixes for related routines** — You'll see this trend show up in a number of Toolbox managers.

  Color QuickDraw PixMap handling routines
  NewPixMap(), DisposePixMap(), CopyPixMap()

  Memory Manager Handle handling routines
  HLock(), HUnlock(), HPurge(), SetHState()

  Oh well, three out of four ain't bad.

- **Pick names that describe** — The days of linkers restricted to seven significant leading characters have passed us by, at least in the Macintosh world.
- **Use case conventions** — Pascal programmers, take note: Use uppercase and lowercase characters differently. The MacApp naming conventions make good use of case and are pretty good all around, even if you're a C/C++ programmer.
- **Use your tools** — Most development systems are bundled with a number of useful analytical tools, such as symbolic debuggers, breakpoint debuggers, and execution profilers. Learn to use them skillfully and often.

Again, these are only guidelines. Any one of them can be the proverbial ounce of prevention that wards off tons of debugging misery.

▶ A Nickel's Worth of Objects

Object-oriented programming software (OOPS) is all the rage these days. It's on everyone's lips and is making substantial inroads in the software tools market. Object-oriented techniques are particularly useful in solving graphics programming problems; the parallels between object and graphics hierarchies reinforce and resonate with each other. A few of the source code examples in this book will be in an object-oriented format. The desire for useful, reusable, and flexible software is behind much of its popularity. At the time of this writing, nearly every major Macintosh development system has some OOPS support, either in the form of syntactical extensions such as C++ and Object Pascal, or in the form of software libraries, such as MacApp and the Think Class Libraries. As far as bandwagons go, OOPS is a pretty good one to jump on, and it will probably take us a long way. At some

point, you will have to make the leap; how quickly is up to you. If you have never been exposed to OOPS, you may be surprised at how easy it is to employ useful object-oriented techniques. As with any coding technique, clarity is a matter of style: The code can vary in complexity from simple to the utterly incomprehensible.

One of the practical aspects of OOPS architecture is the marriage of composite data structures (Pascal Records or C structs) with the routines that use them. The routines that are declared within an object are called *methods*, and the corresponding fields of the object's data structure are called *instance variables*.

The following two sections each contain two examples that compare OOPS techniques with corresponding procedural techniques. Notice the similarities between them. The most important one to notice is that the object doesn't have to pass a reference to its data structure because its instance variables are referenced within each method as though they were local variables.

Declarations: Data Structures Versus Objects

Listings 1-1 and 1-2 reveal a quick comparison of procedural versus object-oriented programming styles in C:

Listing 1-1. Procedural structure declaration

```
/* data structure */
typedef struct
{
  long        twinkle;
  long        realSize;
  long        objSize;
  long        objCount;
  Handle      buffData;
  SignedByte  buffHState;
}
Bag;

/* routines that utilize the data structure */
/* by accessing and modifying the data structure's fields */
void Bag_Init(
  Bag* theBagInfo );

Boolean Bag_Valid(
  Bag* theBagInfo );
```

```
OSErr Bag_Update(
  Bag* theBagInfo );

Handle Bag_Freeze(
  Bag* theBagInfo );

OSErr Bag_Thaw(
  Bag* theBagInfo,
  Handle *theHdl );

void Bag_Print(
  Bag* theBagInfo );

void Bag_Dispos(
  Bag* theBagInfo );
```

Listing 1-2. Object declaration

```
class Bag
{
  /* field variables */
  long        twinkle;
  long        realSize;
  long        objSize;
  long        objCount;
  Handle      buffData;
  SignedByte  buffHState;

  /* methods: routines embedded in the data structure */
  void Init( void );

  Boolean Valid( void );

  OSErr Update( void );

  Handle Freeze( void );

  OSErr Thaw(
    Handle *theHdl );

  void Print( void );

  void Dispos( void );
};
```

With respect to the intent of developmental guidelines, objects and procedural libraries drink from the same well: An object definition should look like a well-designed library of routines that are dedicated to supporting a data structure. That idea is nothing new. What's interesting is that OOPS enforces and tightens the coupling of the code and the data. The differences between calling procedurally based routines and object-based methods is minor. You just have get used to seeing subroutine calling syntax attached to the end of a data structure reference.

Usage: Data Structures Versus Objects

The preceding section pointed out that defining objects is not much different from declaring data structures and procedural routines. Using objects is a similarly easy transition to make. Listing 1-3 shows an example of using the data structures in Listings 1-1 and 1-2 and a set of specialized routines. Listing 1-4 shows an equivalent object-oriented implementation.

Listing 1-3. Using the procedural data structure

```
void SomeRoutine( void )
{
Bag    aBag;     /* the data structure itself */
OSErr  errStat;  /* error status variable */

  /* initialize the data structure */
  Bag_Init( &aBag );

  /* check if the data structure is "OK" */
  if (Bag_Valid( &aBag ))
  {
    /* do something to the Bag */
    errStat = Bag_Update( &aBag );

    /* if it worked, then print it out */
    if (errStat == noErr)
      Bag_Print( &aBag );
  }

  /* cleanup: dispose of any allocated handles */
  Bag_Dispos( &aBag );
}
```

Listing 1-4. Using an object

```
void SomeRoutine( void )
{
Bag    *aBag;    /* a pointer to an object */
OSErr  errStat;  /* error status variable */

  /* allocate memory for the object */
  /* Pascal and C object versions */
  aBag = new( Bag );
  /* C++ version: invoke the constructor */
  aBag::Bag;

  /* initialize the object */
  aBag->Init();

  /* check if the object is "OK" */
  if (aBag->Init())
  {
    /* do something to the Bag */
    errStat = aBag->Update();

    /* if it worked, then print it out */
    if (errStat == noErr)
      aBag->Print();
  }

  /* cleanup: dispose of any allocated handles */
  /* Pascal and C object versions */
  delete( aBag );
  /* C++ version: invoke the destructor */
  ~aBag;
}
```

Important ▶

If these object examples are really foreign to you, you might want to brush up on elementary OOPS theory and technique. See the Bibliography for a list of some helpful reference materials. Although this book isn't steeped in OOPS syntax and examples, a number of examples are presented in this format.

Object-oriented design isn't a panacea for all past and present software development headaches, but its emphasis on additive design techniques and hierarchical organization make it a natural and welcome addition to the graphics programmer's set of tools.

▶ Use of Atomic Data Types

In the interest of consistency with the largest possible number of popular compilers, the standard data types shown in Table 1-1 will be used in this book.

Table 1-1. Standard data types

Generic type	*Size in bits*	*C*	*Pascal*
integer	16	short	Integer
long integer	32	long	LongInt
short fixed point	16	SmallFract	SmallFract
long fixed point	32	Fixed	Fixed
short floating point	32	float	Real
long floating point	80/96	Extended	Extended

▶ Important QuickDraw Clients

In addition to QuickDraw, there are other areas of the Toolbox that are essential to programming a graphical user interface but aren't part of QuickDraw proper. In practice, every other Toolbox manager that has a graphics component uses QuickDraw. The Toolbox managers listed below rely on QuickDraw to implement many of their important features:

- **Window Manager** — The visible components of windows are built from basic QuickDraw data structures.
- **Control Manager** — This manager handles the display and tracking of the standard Macintosh controls: buttons, check boxes, radio buttons, and scrollbars. It also provides mechanisms for using custom controls. Controls are handled by 'CDEF's (Control Definition procedures), which are resource-based routines that typically call a small number of simple QuickDraw traps to draw controls in a number of different states: highlighted, normal, disabled, and so on.

- **TextEdit** — This manager implements display and management of small pieces of formatted multi-typeface text — typically less than a few Kilobytes worth of text at a time. TextEdit isn't suitable for industrial-strength text editing. Its primary role is to handle small amounts of text in dialog windows. TextEdit is also capable of handling text in languages other than English, which can make up for its lack of horsepower.
- **Dialog Manager** — This manager is built on the three managers just described to present dialog boxes and handle the events pertaining to them.

▶ Underneath QuickDraw

Just as QuickDraw supports various higher-level managers, it is supported by other Toolbox managers. The following managers support some of QuickDraw's functionality at the lower levels. A nominal understanding of these dependencies can help when undertaking Macintosh programming.

- **Memory Manager** — A number of QuickDraw data structures contain handles and pointers or are themselves handle- or pointer-based. The Graphics Toolbox relies on Memory Manager routines to create, resize, and dispose of handle-based data structures such as Regions, Polygons, and many color-related data types.
- **Resource Manager** — Some graphics data structures can also be created and initialized from resources. The rules regarding the usage and maintenance of resources are applicable to these data structures. All standard user interface objects and their associated color information data types are usually loaded from resources.

The following managers are involved with the generation and display of text:

- **Font Manager** — This manager implements the lower-level management of typeface information, including measurements for character and style modifications.
- **Print Manager** — This manager dynamically translates the QuickDraw commands issued by an application into those that can be understood by printers and other hardcopy output devices. The most familiar example of this translation mechanism is the LaserWriter's QuickDraw-to-PostScript conversion. The Print

Manager is also responsible for presenting the familiar Print and Page Setup dialogs through the lower-level print drivers.

- **International Utilities** — The Macintosh is quite successful in many countries outside the United States, in large part because of its ability to display, format, and manage text in languages other than English. The International Utilities package provides most of these capabilities, along with English text formatting for dates, times, and monetary amounts. It won't make writing a Japanese word processor easy, but it'll help. TextEdit and other text-based services call the International Utilities package directly.

▶ Summary

The Macintosh provides a powerful and flexible graphics environment based on the Toolbox manager called QuickDraw. QuickDraw gives applications developers powerful and flexible means with which to render graphics both on and off the screen. These capabilities require careful programming and a fairly thorough understanding of computational graphics concepts. Ultimately, the software developer's willingness to address the issue through thoughtful software design and sensible organization will be the greatest factor in determining any software application's utility and success. Many of the techniques referred to in this chapter were generally stated and could be applied to development strategies for hardware platforms other than the Macintosh.

The strengths and weaknesses of the Macintosh graphics environment were also presented in general terms. In particular, the chapter emphasized the graphics environment's functional capabilities and durability of design.

This chapter also presented a case for object-oriented design. The authors like the compact representation of object syntax. Finally, some of the lower-level managers were described.

2 ▶ Basic Drawing Technique and Theory

This chapter presents QuickDraw's basic drawing and calculation functionality. You may already be familiar with QuickDraw's basic shape drawing and calculation support. If so, you may want to skim this chapter. For those readers who are new to Macintosh programming, it's essential that you read this chapter, along with the QuickDraw and Window Manager chapters of *Inside Macintosh,* Volume I.

QuickDraw is expansive: There are nearly a hundred QuickDraw-specific data structures and a few hundred Toolbox routines. Large software environments can be difficult and daunting. QuickDraw is less hostile than most, but it's a vast body of functionality, implementing a large family of basic graphic constructs. We begin by introducing the basics of digitized images. The middle portion of the chapter discusses shape drawing, and we conclude with the integration of QuickDraw concepts into a single graphics environment.

(Note that because of the complexity of color graphics programming, color issues are temporarily set aside; for now, consider drawing to be in black-and-white. We discuss color support in the next chapter, and we will build on the basic functionality presented in this chapter.)

By the Way ▶

Throughout the course of this book, we use the term *trap* when referring to executable code in the Toolbox and the term *routine* when referring to executable code outside the Toolbox. The authors prefer these terms because they are relatively language-independent. You may use other terms for "routine," such as "procedure" or "function" if you're a Pascalite, or "subroutine" if you're from a Fortran or Basic school of programming. In any case, we hope that our choice of terminology is at least comprehensible, if not appreciated.

▶ Raster Graphics

The world of computer graphics includes a variety of different graphic output devices. There are line printers, laser printers, plotters, storage tubes, and raster-scan displays, to name the most familiar ones. At the high levels of interacting with QuickDraw, you're describing shapes and patterns that are fairly independent of the device on which they're drawn. But at the next level down, the level at which QuickDraw is creating and storing the results of drawing operations, QuickDraw reflects the display device for which it was designed: the raster-scan CRT.

A Macintosh display monitor and a TV set are both raster-scan devices. They create an image by shooting an electron beam (or beams) at the excitable phosphor coating on the face of the screen. The beam is aimed magnetically and is swept across a horizontal line from one side of the screen to the other, and then another horizontal line is scanned just below the first. Line after line is drawn until the screen is filled from top to bottom. As the electron beam is scanned across a horizontal line, it is modulated to create adjacent spots where the phosphor is excited to different intensities. The result is a two-dimensional matrix of differently illuminated spots that the eye blends together as a single image. Figure 2-1 illustrates the composition of a raster-scan image.

Raster-scan displays come in a lot of different sizes. For example, a thirteen-inch Apple RGB monitor typically displays 480 lines of 640 dots each; the Classic Macintosh's nine-inch screen has 342 lines of 512 dots. For reasons described later, a color monitor has not one but three independent electron guns: one each for red, green, and blue spots. Various colors are created on the screen by mixing the intensities of red, green, and blue for a given spot on the screen.

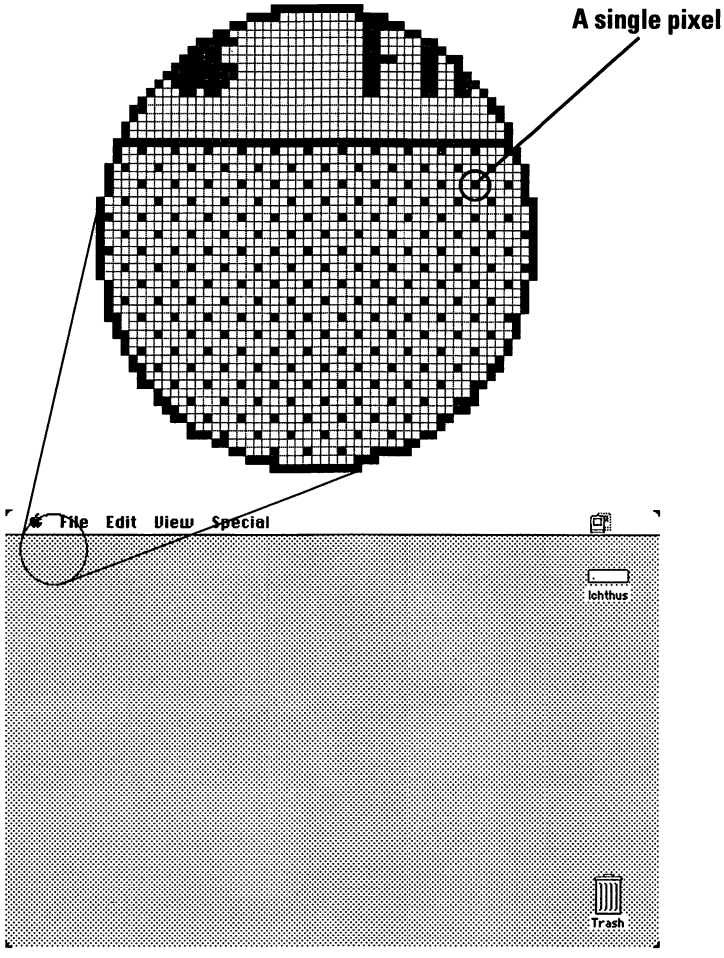


Figure 2-1. Display monitor, with inset of magnified pixels

The Macintosh's memory is, of course, digital in nature, while a raster-scan display is an analog device. A picture stored in its digital form in memory is sent to special electronic circuitry that creates from it an analog signal (video) for the monitor. The monitor decodes the video signal and uses it to direct the modulation of the sweeping electron beams.

▶ Digital Images

The picture on a raster-scan CRT is composed of a two-dimensional matrix of spots called *pixels* (short for picture element), which are the smallest unit of a screen image. Pictures can be stored in Macintosh memory as a sequential stream of pixels, each of which describes one point in the picture.

▶ Points

A point is the atomic unit of graphic drawing and calculations. Locations on a graphics coordinate plane are referenced by variables of type Point, a data structure that is composed of two integers that indicate a horizontal and a vertical coordinate. Points are most often used in one of two ways:

- to define the boundaries of shapes. Examples of shape definition are the endpoints of a line, the vertex points of a triangle, or the control points of a curved line segment.
- to hold the results of calculations involving the mouse location, which has a point coordinate at any given instant. The current mouse position can be obtained by calling the **GetMouse**() trap, which has the following interface:

```
void GetMouse(
  Point local MousePos);
```

When handling mouse-based events, the mouse coordinate is returned in the EventRecord's *where* field, which is also of type Point.

Measuring with Points and Pixels

A Point is the address of a single indivisible unit of drawing, not a drawing unit itself. Points have no inherent size, but they are used to measure distances and define location (see Figure 2-2) in the same way that mile markers on the highway indicate distance and location.

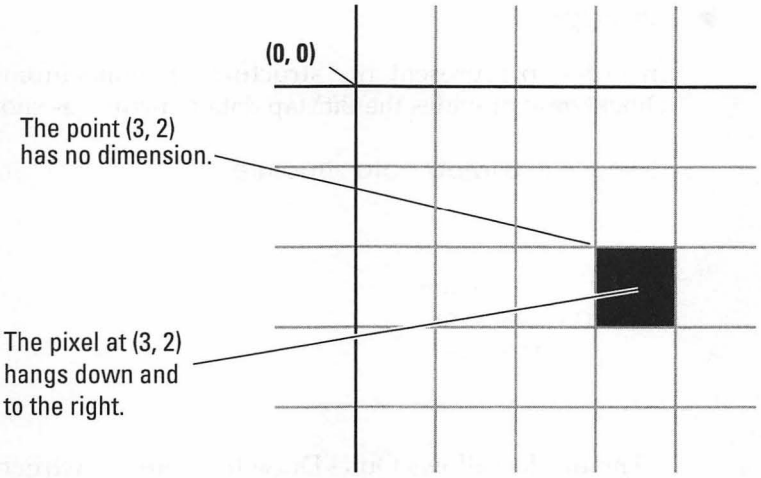


Figure 2-2. The difference between a point and a pixel

Resolution

The translation from a point to an actual pixel is known as the *resolution*. When referring to the screen itself, a point is the location of the upper-left corner of a single pixel. The resolution of the typical Macintosh monitor is 72 pixels per inch. Hardcopy devices (printers, scanners, film recorders, and the like) usually have a much greater resolution: Standard LaserWriter resolution is 300 dots per inch (dpi), and many other printers have resolutions of 600 dpi or greater.

Important ▶

It's important to understand that there is no inherent resolution of the QuickDraw coordinate plane; resolution is arbitrary, and it is only important when the results of drawing operations are transferred to a physical device, such as a screen or printer.

▶ BitMaps

In order to represent the structure of monochrome image memory, QuickDraw provides the BitMap data structure, as shown in Listing 2-1.

Listing 2-1. BitMap data structure

```
typedef struct
{
  Ptr    baseAddr;
  short  rowBytes;
  Rect   bounds;
}
BitMap;
```

The BitMap allows QuickDraw to express unstructured memory in a rectangular format that parallels the raster-scan orientation of video displays. The role of the BitMap's fields is as follows:

- ***baseAddr*** — This is simply a pointer to some memory allocation, usually the result of a **NewPtr**() call.
- ***rowBytes*** — This field holds the number of bytes in a single row of pixel values. Since each monochrome pixel requires a single bit (0 for white, 1 for black), the value of *rowBytes* is the number of pixels in a row, rounded up to the nearest multiple of 16 and then divided by 8. For example, a BitMap with 53 pixels per row would have its *rowBytes* value calculated in the following manner:

  53 pixels per row
  = 64 (nearest multiple of 16)
  = 64 / 8 bits per byte
  = 8 bytes per row

  The additional 11 bits at the end of each row are unused and therefore wasted.
- ***bounds*** — This field is a Rect variable, a type used to express a rectangle that describes the BitMap's coordinate system. Rects are discussed at length later in this chapter.

BitMaps are very important to QuickDraw; the results of drawing operations ultimately end up in a BitMap, or else those results cannot be displayed. For almost all QuickDraw operations, the drawing destination BitMap is determined by QuickDraw and isn't directly specified. We'll see how this is determined later in this chapter.

Important ▶

On black-and-white Macintoshes, the screen is described by a BitMap whose *bounds* value is equal to the screen dimensions in pixels and a *baseAddr* value that points to the actual screen image memory.

▶ Basic Drawing Operations

When we think about drawing, we often consider the limitations of pen and pencil to be an integral part of drawing itself. But that is really only semantic baggage: One could just as easily say that a painter "draws" with a brush, although most people say that a painter "paints." When people draw, they draw with lines and dots, and build more complex shapes from collections of lines and dots. Computers typically draw complete shapes all at once.

▶ Using Shapes

As Figure 2-3 illustrates, simple computer-generated shapes have three important attributes:

- **Filling pattern** — The digital-world equivalent of texture.
- **Border or perimeter** — The shape's visual skeleton, often drawn with a different pattern than the filling.
- **Control points** — The shape's mathematical skeleton. Control points are coordinate values used to define specific instances of a given shape. These can be a shape's corner points for shapes such as polygons, or points situated near the shape, such as the foci of conic curves or the defining points of complex spline curves.

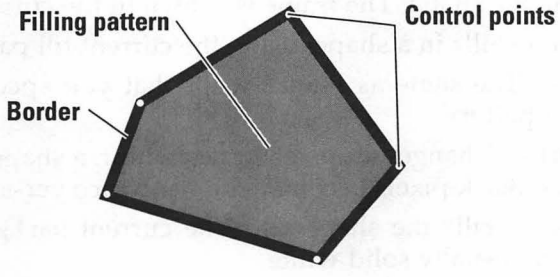


Figure 2-3. Attributes of computer-generated shapes

Figure 2-4 shows examples of the basic QuickDraw shapes: Rectangles, Round Rectangles, Ovals, Arcs (Wedges), Polygons, and Regions. Lines and text are exceptional shapes, included here only because QuickDraw can draw them as well as the others.

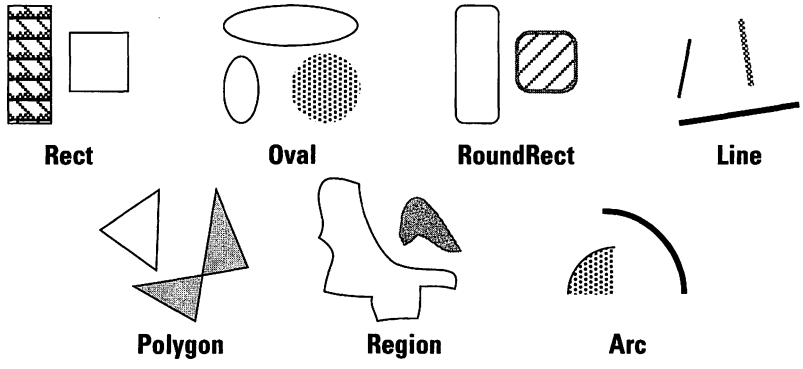


Figure 2-4. Basic QuickDraw shapes

Recognize that many other shapes can be derived from these basic ones; for example, circles and squares are special cases of ovals and rectangles, hexagons and triangles are simply types of polygons.

▶ GrafVerbs

The standard QuickDraw shape-drawing commands are known as GrafVerbs. The five QuickDraw GrafVerbs are Frame, Fill, Paint, Invert, and Erase. These are implemented for all six basic QuickDraw shape types:

- **Frame** — Draws the outline of a shape, inside of the points that define the shape. The frame is drawn in the current pen pattern.
- **Paint** — Fills in a shape, using the current fill pattern.
- **Fill** — The same as Paint except that you specify explicitly the filling pattern.
- **Invert** — Changes all pixel values within a shape to their complements. Black pixels become white and vice versa.
- **Erase** — Fills the shape with the current background color and pattern, usually solid white.

The names of QuickDraw drawing traps are usually formed by concatenating a GrafVerb with a graphic object name, for example FillRect. These graphic verbs are very much like their linguistic verb counterparts in that they refer to actions without referring to the objects that they operate on: "fill" means fill, whether the object is a rectangle or a gas tank. Figure 2-5 shows the visual effects of GrafVerbs.

The actions of a given GrafVerb depend on two factors:

- the state of the graphics environment variables that define the pen and color information
- the pixel values in the area where the drawing operation is to take place

These factors are discussed later in this chapter.

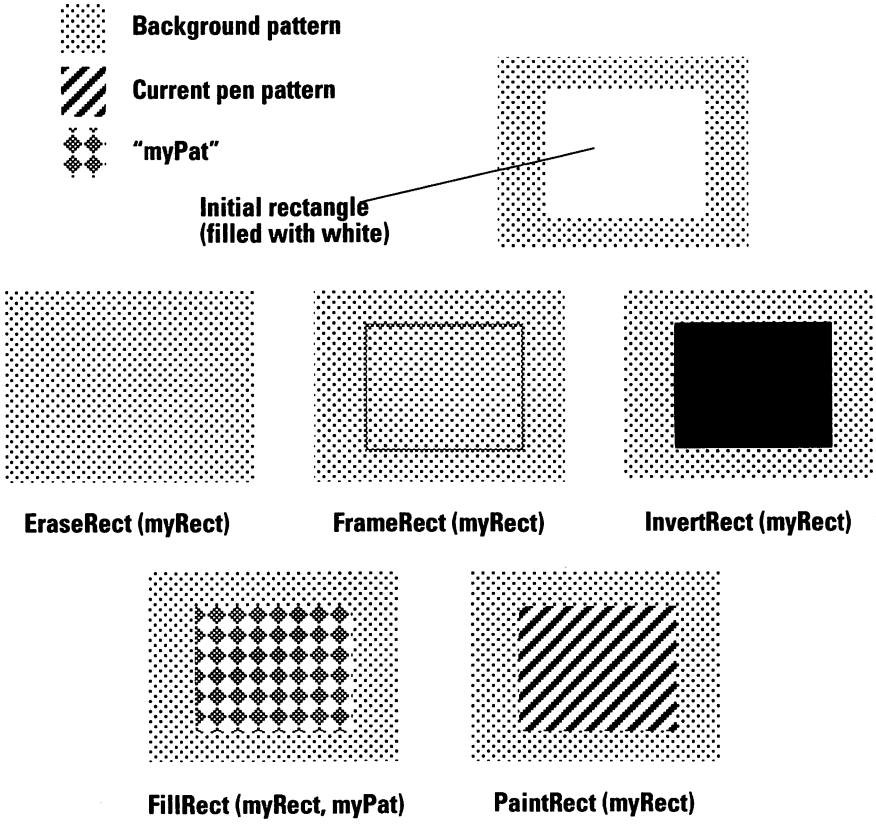


Figure 2-5. Results of GrafVerb operations on a rectangle

In many cases, QuickDraw provides a trap to perform a single operation on a given shape. Much of this drawing functionality can be expressed simply as a table of shapes and shape operations, as shown in Figure 2-6.

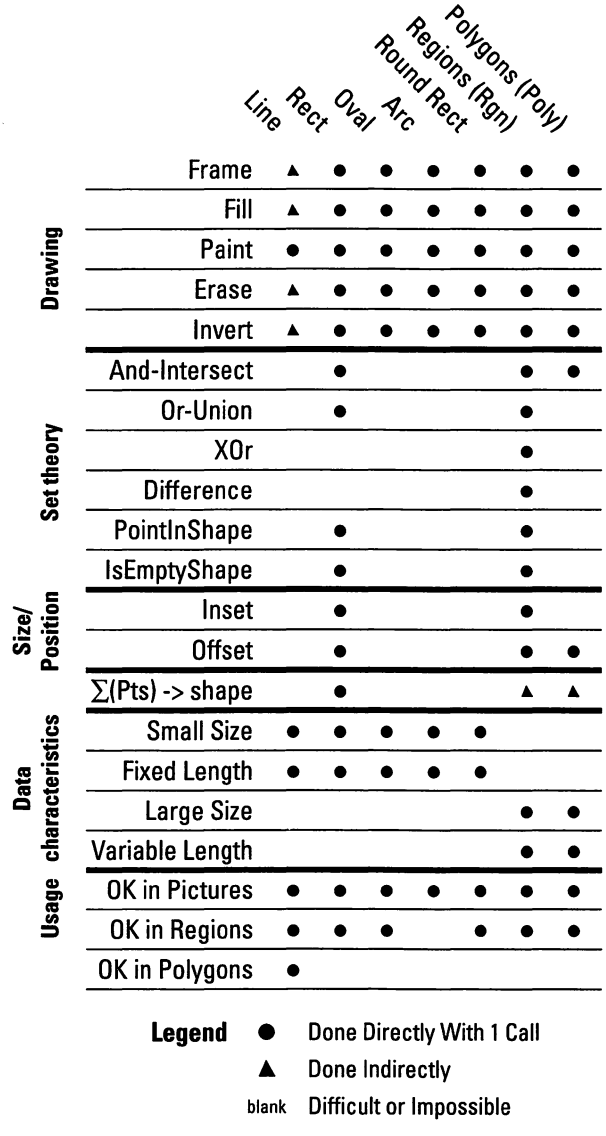


Figure 2-6. QuickDraw shapes and operations

▶ Pens and PenStates

When QuickDraw shapes are drawn, there are five basic parameters that determine the shape's appearance. These are collectively referred to as the *pen* or *PenState*. The PenState contains the following information:

- **Visibility** — Loosely put, visibility indicates whether the pen is actually touching the digital "paper." The pen can be hidden, in which case all line-drawing and framing operations will not produce visible results. The visibility is maintained as an integer whose initial value is zero; the **HidePen**() trap increments the "level" value, whereas the **ShowPen**() trap decrements it. Zero or negative values indicate a visible pen. A zero or negative number pen size has the same effect as hiding the pen; however this doesn't affect the pen level value.
- **Location** — The pen location affects all line-drawing operations. Calling **LineTo**() with integer values "x" and "y" draws a line from the current pen location to (x, y) and sets the new pen location at (x, y). Lines can also be drawn relative to the current pen position: Calling **Line**() with the same arguments will draw a line from the current position to a point *x* pixels to the right and *y* pixels down.
- **Size** — The pen shape is rectangular, having a horizontal and vertical size. The **PenSize**() trap sets both the horizontal and vertical dimensions. Figure 2-7 shows the effects of **PenSize**() calls on the results of **FrameRect**().

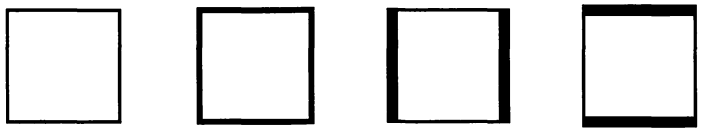


Figure 2-7. Pen sizes

The pen size affects all line-drawing and framing operations. If either pen size dimension is zero or negative, the pen will behave as though it's hidden, and will not draw during GrafVerb and line-drawing traps.

- **Pattern** — When shapes and lines are drawn, they aren't necessarily filled in solid. The digital "texture" or "filling" is called a *pattern*. A pattern is an 8-by-8 array of bits, with each bit designating a pixel: a bit, if set, means black; if clear, it means white. Listing 2-2 shows the Pattern data structure.

Listing 2-2. Pattern data structure

```
typedef char[8] Pattern;
```

The Pattern data structure holds the pattern's pixel values as they occur from left to right and top to bottom. The upper-left pixel value is stored in the most significant bit of the Pattern's first byte; the last pixel value is stored in the least significant bit of the Pattern's last byte.

Important ▶

Because the Pattern's declaration is not a struct or Record, the respective C and Pascal compilers will treat it as an array type when passing pattern variables as arguments to traps and routines. In C, this always means the address or the array, rather than the array data itself. In Pascal, this is usually the behavior, although it has been known to vary from one compiler to the next.

Figure 2-8 shows the bitwise nature of some patterns (top row), compared to what they appear to be at a normal viewing distance (bottom row).

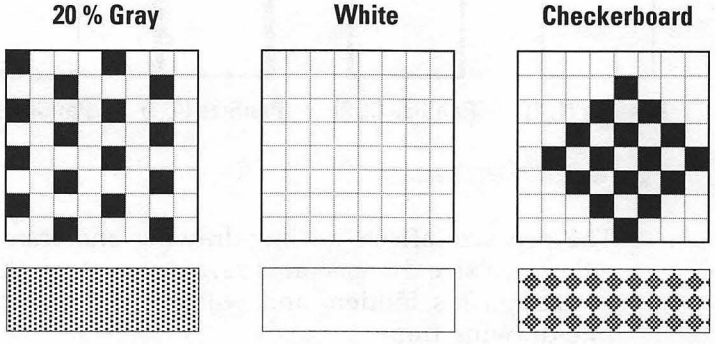


Figure 2-8. Close-up of patterns

The pattern most commonly used is a solid black pattern, where every bit in the pattern is set. Shapes and lines drawn in a black pattern change every pixel underneath. The opposite of a solid black pattern is white, naturally. Patterns are often used in place of color to express differences between visual objects, especially when color isn't available.

Patterns are usually accessed from resources. The System file contains an array of patterns in resource 'PAT#' 0. These patterns can be inspected with a resource editor such as ResEdit or accessed from an application by calling the **GetIndPattern**() trap. Additionally, there are five standard patterns that are available to applications as named QuickDraw global variables: black, white, ltGray, gray, and dkGray.

- **Transfer mode** — The pen transfer mode determines the way the "ink" is absorbed by the "paper" being written on. When a drawing operation is performed, the actual pixels changed in the destination image depend on the values of the pixels under the new drawing operation. The rules defining these combinations are called *transfer modes*. These transfer modes provide a number of flexible and powerful capabilities, many of which will be discussed throughout the course of the book. Figure 2-9 shows the results of drawing "shape one" first and then drawing "shape two" on top of it using one of eight different modes in turn.

These transfer modes are really Boolean operators that define the results of combining two images as though they were simple bit patterns.

The effects of the pen state on GrafVerb trap calls vary from verb to verb, as shown in Figure 2-10.

Given that the pen's state can be modified in many ways, it only makes sense that there would be ways to obtain and preserve that state. QuickDraw provides a PenState data structure and two traps, **GetPenState**() and **SetPenState**(), to maintain pen state information. Listing 2-3 shows the PenState data structures.

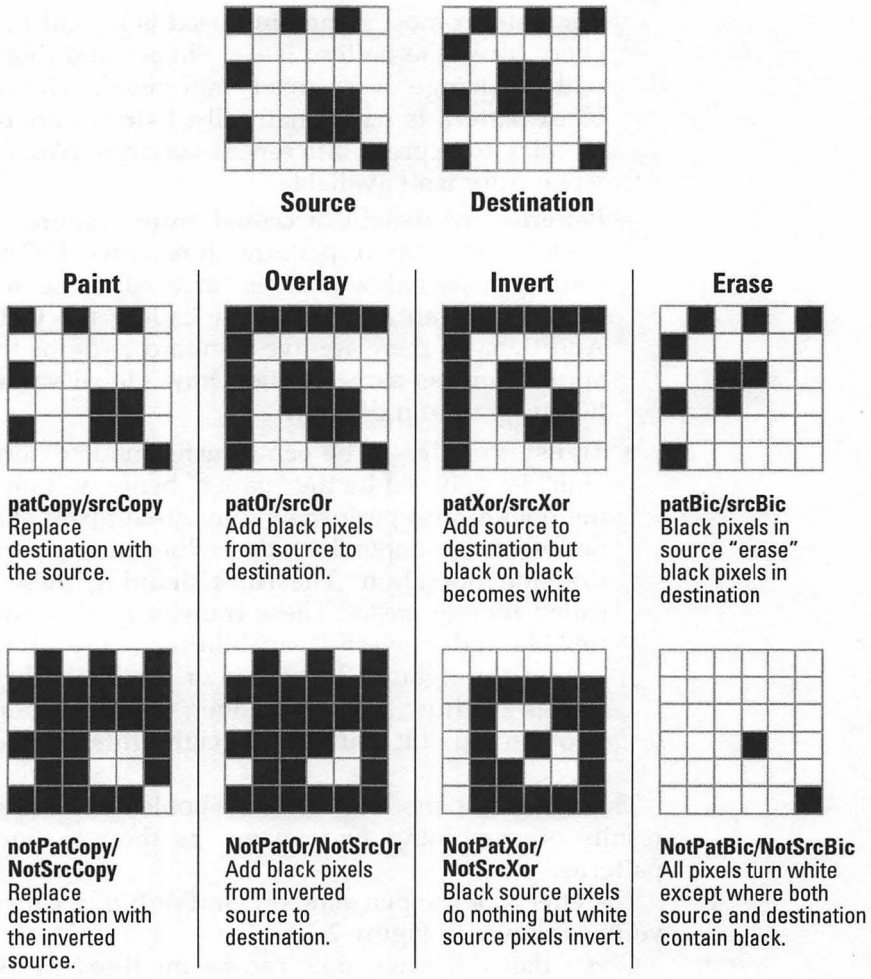


Figure 2-9. Pen transfer modes

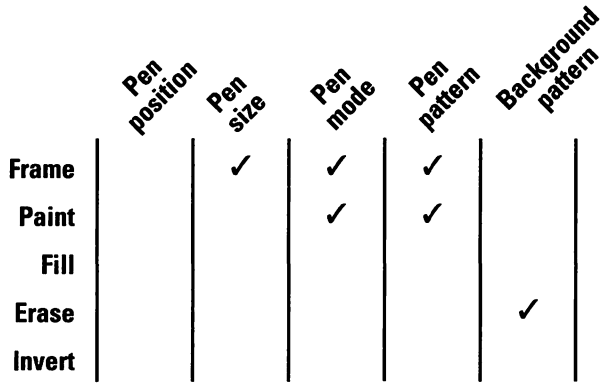


Figure 2-10. GrafVerbs and PenStates

Listing 2-3. PenState data structures

```
typedef struct
{
  Point    pnLoc;    /* default: 0, 0 */
  Point    pnSize;   /* default: 1, 1 */
  short    pnMode;   /* default: srcCopy */
  Pattern  pnPat;    /* default: black */
}
PenState;
```

Notice that the pen's visibility state isn't accounted for in the PenState data structure; it's maintained by QuickDraw in a separate data structure, as you will see later in this chapter. The **PenNormal**() trap allows you to set the current pen values to the defaults listed here. These three pen traps can be used together to facilitate robust drawing without causing side effects to subsequent drawing operations. Listing 2-4 is a routine that shows an example of maintaining the pen state while drawing.

Listing 2-4. Maintaining the pen state

```
void ARobustDrawingRoutine( void )
{
PenState savedPen;

  /* obtain the current pen state */
  GetPenState( &savedPen );
```

```
    /* set the default pen state */
    PenNormal();

    /* do a bunch of drawing here... */

    /* restore the original pen state */
    SetPenState( &savedPen );
}
```

▶ Lines

The simplest drawing operations are the line-drawing operations. Lines are drawn from one point to another. The starting point is known as the current pen position and is the finishing point of the previous line-drawing operation. Lines can be drawn from the current pen position to a new absolute coordinate or to a specified horizontal and vertical distance relative to (that is, away from) the pen position. The pen can also be moved without drawing a line, again to either an absolute or relative position. The following traps are particularly important, implementing most of QuickDraw's line drawing capabilities:

```
void LineTo(
  short  x,
  short  y );
```

Draws a line from the current pen position to (x, y) and makes that the new pen position.

```
void Line(
  short  x,
  short  y );
```

Draws a line from the current pen position x points horizontally and y points vertically. The new pen position becomes (old horizontal position + x, old vertical position + y).

```
void MoveTo(
  short  x,
  short  y );
```

Positions the pen the same way as **LineTo()** does, but without drawing anything.

```
void Move(
  short  x,
  short  y );
```

Positions the pen the same way as **Line**() does, but without drawing anything.

Listing 2-5 shows a simple example of line drawing.

Listing 2-5. Example of line drawing

```
void DrawLines( void )
{
short  i;

  for (i=1; i<10; i++)
  {
    MoveTo( 80, 20 );       /* move to the starting point */
    LineTo( i*10, i*15 );   /* draw a line outwards */
  }
}
```

Figure 2-11 shows the results of calling DrawLines().

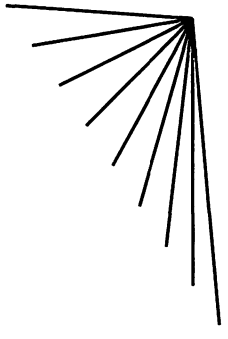


Figure 2-11. Results of calling DrawLines()

Although line-drawing traps aren't implemented by GrafVerbs, the appearance of drawn lines is determined primarily by the pen's transfer mode, size, and pattern.

▶ Rectangles

Next to the Point, the Rect is the second most fundamental QuickDraw data type. A Rect is the simplest data type that can define an area. Rects can be expressed in two forms, as shown in Listing 2-6.

Listing 2-6. Rect data structure

```
/* primary form */
typedef struct
{
   short  left;
   short  top;
   short  right;
   short  bottom;
}
Rect;

/* alternate form */
typedef struct
{
   Point  topLeft;
   Point  bottomRight;
}
Rect;
```

Figure 2-12 shows the structure of Rects as they relate to coordinate planes.

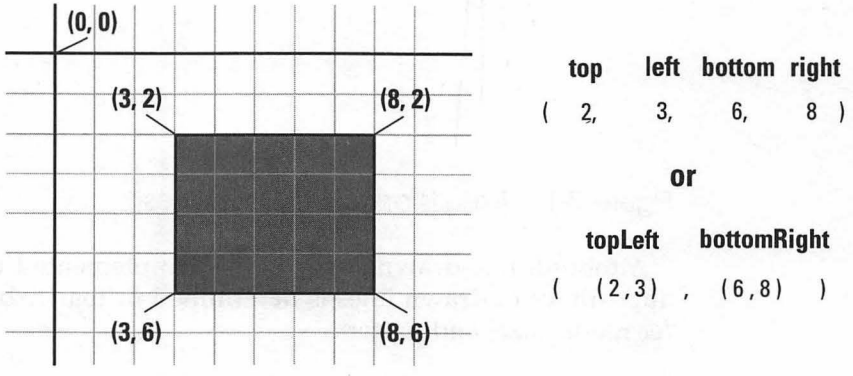


Figure 2-12. Rect structure, expressed visually

Note ▶

Some compilers define both types of Rects together, using union types (C) or variant records (Pascal). Others define only the four-magnitudes form and optionally provide macros and other syntactical esoterica to allow your source code to fool the compiler into thinking that you want the first form. You'll probably have your choice of forms, although we prefer the primary form.

Many coordinate-naming variations will spring up: x-y, h-v, left-right-top-bottom. You can keep yourself out of trouble by remembering that the X components always come before the Y components in trap argument lists: *X*-y, *H*-v, *Left*-top-*Right*-bottom, and so on.

Notice that only two of the four corner points are needed to define any rectangle, the other two are redundant. Because of its compact representation, the Rect is one of the most important QuickDraw data types. Virtually every user interface object uses a rectangle to define a "simple" boundary.

The fields of Rect variables are commonly assigned via the **SetRect()** trap, but any of them can be assigned directly, as shown in Listing 2-7.

Listing 2-7. Assigning values to Rect variables

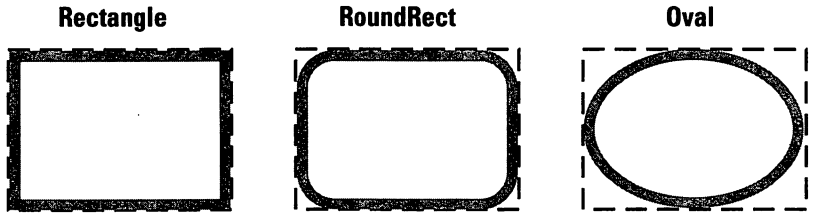
```
void AssigningRects( void )
{
Rect someRect;

  /* A Rect can be assigned this way… */
  SetRect( &someRect, 20, 25, 100, 80 );

  /* …or this way instead. */
  someRect.left = 20;
  someRect.top = 25;
  someRect.right = 100;
  someRect.bottom = 80;
}
```

▶ Rectangle-Based Shapes

Rects aren't used only to define area; they are also used to express three basic shapes: rectangles, rounded rectangles, and ovals. Examples of these shapes are shown in Figure 2-13.



The bounding rectangle is actually infinitely thin and invisible.
These shapes are framed with a 4 point medium-gray pen.

Figure 2-13. Rectangle-based shapes

Each of these three shapes has the full brace of five GrafVerb drawing traps. The RoundRect shape traps require two additional arguments to specify the horizontal and vertical curvature of the corners. These rounded corners are also known as *fillets*.

Note that there is a difference between calling a framing GrafVerb trap and drawing the shape with successive **LineTo**() or **Line**() calls. The C routine shown in Listing 2-8 illustrates this point amply.

Listing 2-8. Line drawing versus framing

```
void NotTheSameRectangle( void )
{
Rect thisRect;

   PenSize( 2, 2 );
   SetRect( &thisRect, 10, 10, 80, 80 );

   /* will draw black, completely inside "thisRect" */
   PenPat( black );
   FrameRect( &thisRect );

   /* will draw gray, to the right and down of "thisRect" */
   PenPat( gray );
   MoveTo( thisRect.left, thisRect.top );
   LineTo( thisRect.right, thisRect.top );
   LineTo( thisRect.right, thisRect.bottom );
   LineTo( thisRect.left, thisRect.bottom );
   LineTo( thisRect.left, thisRect.top );
}
```

The results of calling NotTheSameRectangle() are shown in Figure 2-14.

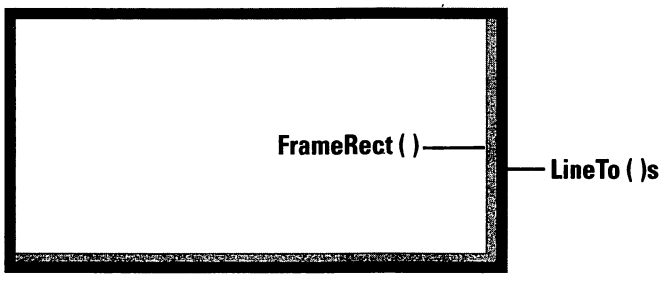


Figure 2-14. Results of calling NotTheSameRectangle()

Regions

A region is the QuickDraw data type used to describe a set of one or more closed curves. Regions are one of the more innovative elements of QuickDraw and are the primary data type used to describe an irregularly shaped area. Like Rects, regions have five associated GrafVerb traps, shown in Figure 2-15.

Regions are dynamically resizable. As with most variable-length Macintosh data types, regions are relocatable and referenced through Handles (RgnHandles, actually). The amount of memory allocated to a region is maintained by the traps that modify it.

Regions contain a 16-bit integer value indicating their memory size and a Rect at the beginning of the data structure that describes the smallest rectangular bounding area, called the *rgnBBox* (region bounding box). The relationship between a region and its bounding box is shown in Figure 2-16.

The remainder of the region is composed of scanline-oriented data describing the actual shape of the region within the bounding rectangle. The Region data structure is shown in Listing 2-9.

Listing 2-9. Region data structure

```
typedef struct
{
  short  rgnSize;
  Rect   rgnBBox;
  char   rgnData[1];
}
Region, *RgnPtr, **RgnHandle;
```

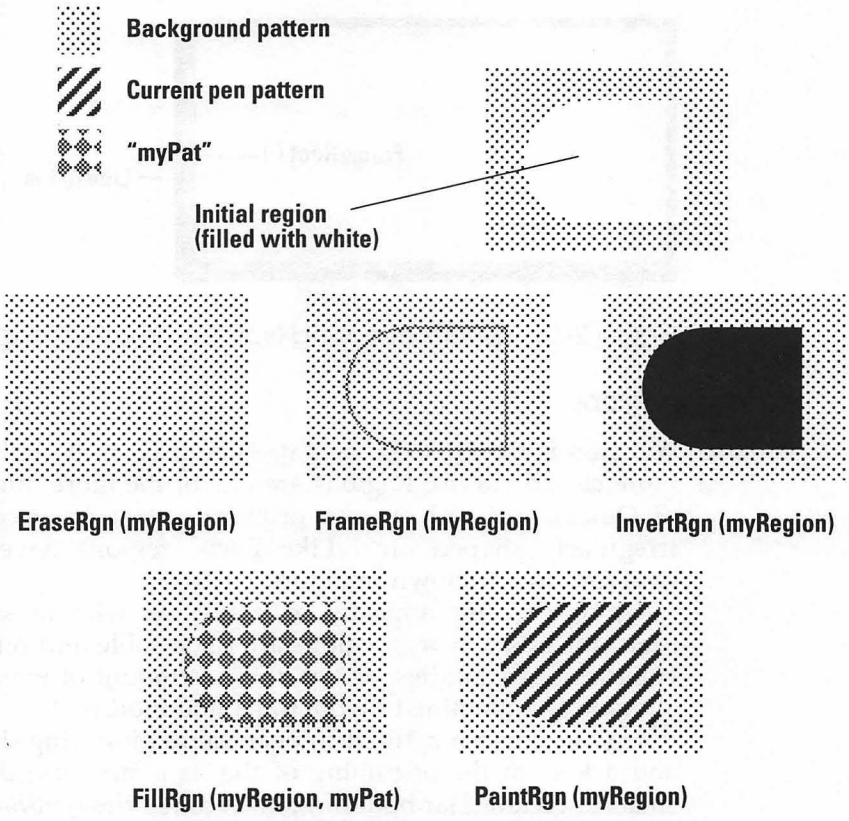


Figure 2-15. Results of GrafVerb operations on a region

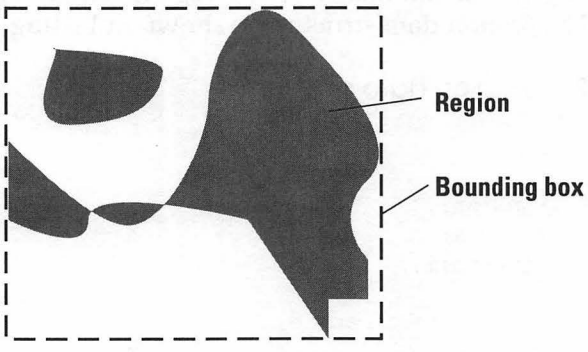


Figure 2-16. Region and bounding box

Notice that the *rgnData* field is declared as a single-element array; in practice, it can be anywhere from 0 to 64K. The actual size of a region is maintained dynamically by the region traps. This syntax is yet another of the compiler-twisting games that we'll get involved with over the course of this book. Regions and other variable-length data types are declared with the last field in the structure being a single-element array in order to facilitate array referencing of the dynamically allocated data at the end of a Handle. Such data types are usually accessed through Handles; rarely, if ever, are they referenced through their base type. For example, regions are always accessed through RgnHandles; variables of type "Region" should never be explicitly declared or used.

Creating Regions

Unlike Rects and other simple fixed-length data types, region data cannot be directly assigned; rather they are created by "opening" them and calling other QuickDraw line-drawing and frame-drawing traps to define their extent. When the desired area is defined, the region is then closed. Listing 2-10 shows an example of creating a region.

Listing 2-10. Creating a region

```
RgnHandle   CreateSampleRegion( void )
{
RgnHandle     sampleRgn;

  sampleRgn = NewRgn();  /* allocate new region */

  OpenRgn();             /* open for region drawing */
    MoveTo( 50, 50 );    /* sketch a rectangle */
    Line( 80, 0 );       /* with triangles at the corners */
    Line( -10, -10 );
    Line( 20, 0 );
    Line( -10, 10 );

    Line( 0, 130 );
    Line( 10, 10 );
    Line( -20, 0 );
    Line( 10, -10 );

    Line( -80, 0 );
    Line( -10, 10 );
    Line( 20, 0 );
    Line( -10, -10 );
```

```
    Line( 0, -130 );
    Line( -10, -10 );
    Line( 20, 0 );
    Line( -10, 10 );
  CloseRgn( sampleRgn ); /* close the region, accumulating */
                         /* the result into sampleRgn */

  return sampleRgn;
}
```

Calling CreateSampleRegion() would produce the region in Figure 2-17, framed here with **FrameRgn**() to illustrate it.

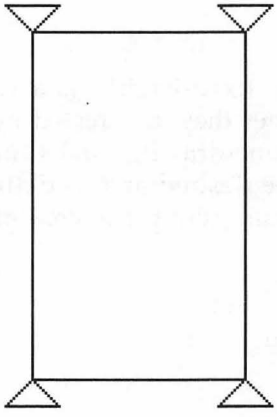


Figure 2-17. Results of calling CreateSampleRegion()

Important ▶

When a region is opened, the pen is hidden until the region is closed; consequently, no visible drawing occurs. If drawing is to take place at the same time that a region is being built, the **ShowPen**() trap should be called immediately after the region has been opened. The **HidePen**() trap will hide the pen and should be called before closing the region to balance the **ShowPen**() call.

Additionally, only one region may be open at a time per GrafPort, and that region must be closed before handling an event. The GrafPort data structure is described later in this chapter.

Regions and Set Theory Traps

Regions can also be defined from other regions using QuickDraw traps that perform familiar set theory operations over the area that each region operand describes. The set theory traps are:

```
void SectRgn(
  RgnHandle   rgn1,
  RgnHandle   rgn2,
  RgnHandle   destRgn );
```

Set "destRgn" equal to the intersection of two regions.

```
void DiffRgn(
  RgnHandle   rgn1,
  RgnHandle   rgn2,
  RgnHandle   destRgn );
```

Set "destRgn" equal to the difference of two regions.

```
void UnionRgn(
  RgnHandle   rgn1,
  RgnHandle   rgn2,
  RgnHandle   destRgn );
```

Set "destRgn" equal to the union (inclusive or) of two regions.

```
void XorRgn(
  RgnHandle   rgn1,
  RgnHandle   rgn2,
  RgnHandle   destRgn );
```

Set "destRgn" equal to the union minus the intersection (exclusive or) of two regions.

In each case, "destRgn" must already exist (created via **NewRgn()**) before the set theory trap is called. Additionally, the "destRgn" argument can be the same as the "rgn1" or "rgn2" argument.

Important ▶

As of the System 6.0.5 software release, all four of these region set theory traps have the additional restriction that the sum of the first two source regions' data must be less than 32K, or the trap may corrupt the application's heap. This is true regardless of what the size of the destination region would have been had the trap succeeded. Check the sum of the sizes of the RgnHandles before calling one of these traps.

Figure 2-18 shows examples and results of using these set theory calculation traps.

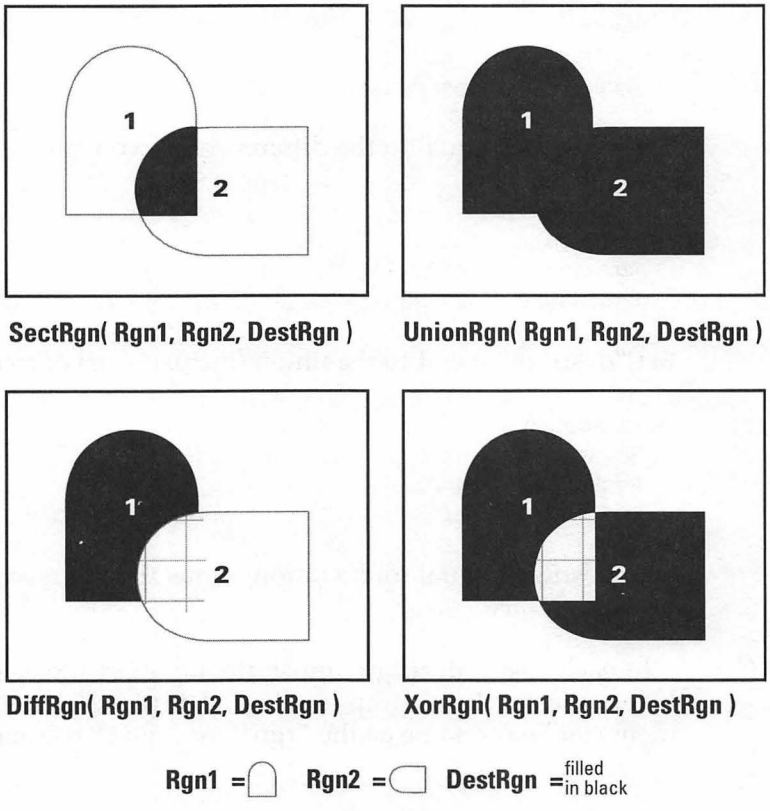


Figure 2-18. Effects of the region set theory traps

Other Region Calculation Traps

Regions can also be created from other regions using QuickDraw traps that perform set theory operations over the area that each region operand describes. These traps are:

```
void OffsetRgn(
  RgnHandle     theRgn,
  short         dx,
  short         dy );
```

Move "theRgn" by "dx" pixels to the right and by "dy" pixels toward the bottom. Negative values shift the region to the left and upward.

```
void InsetRgn(
  RgnHandle     theRgn,
  short         dx,
  short         dy );
```

Inset "theRgn" by "dx" pixels on each horizontal size and by "dy" pixels on each vertical size. Negative values "outset" the region. Warning: Insetting a region by large amounts may cause it to go empty. If you're unsure of a region's dimensions before calling this trap, make sure to check it with **EmptyRgn**() afterward.

```
void MapRgn(
  RgnHandle     theRgn,
  Rect          *srcRect,
  Rect          *destRect );
```

Inset "theRgn" to fit inside of "srcRect" and then scale it up to "destRect." Warning: Calling this trap with an empty region causes application failure.

Important ▶

Apple discourages programmers from tinkering directly with a region's data, and it hasn't published the format of the data itself. Fortunately, Apple has provided a pretty complete set of traps to manipulate the data. There should be no need to determine that data format, but tinkerers with time on their hands can feel free to discover it on their own.

As you can see, regions play a number of roles in Macintosh applications programming. Refer to volumes in the *Inside Macintosh* series for other nifty region calculation traps.

Using Rectangles and Regions

Rects and regions permeate the QuickDraw environment and are used in most calculation and drawing operations. Virtually all familiar Macintosh graphic objects are visually defined by some group of Rects and regions. It's quite likely that Rects and regions will end up defining your application's specialized graphics objects, too. Listing 2-11 shows an example of drawing a simple object based on drawing rectangle-based shapes.

Listing 2-11. Drawing simple graphic objects with Rects and regions

```
void DrawTruck( void )
{
RgnHandle   tiresRgn;
Rect        hitchR, boxR;
Rect        cabR, tiresR;

  PenSize( 2, 2 );
  PenPat( black );

  /* draw the tires first */
  tiresRgn = NewRgn();
  OpenRgn();
    SetRect( &tiresR, 0, 0, 30, 30 );
    OffsetRect( &tiresR, 30, 150 );
    FrameOval( &tiresR );
    OffsetRect( &tiresR, 110, 0 );
    FrameOval( &tiresR );
    OffsetRect( &tiresR, 60, 0 );
    FrameOval( &tiresR );
    OffsetRect( &tiresR, 45, 0 );
    FrameOval( &tiresR );
  CloseRgn( tiresRgn );
  PaintRgn( tiresRgn );
  DisposeRgn( tiresRgn );

  /* build the hitch */
  SetRect( &hitchR, 0, 0, 40, 20 );
  OffsetRect( &hitchR, 160, 125 );
  FrameRect( &hitchR );
```

```
  /* build the cab */
  SetRect( &cabR, 0, 0, 60, 45 );
  OffsetRect( &cabR, 190, 55 );
  FrameRoundRect( &cabR, 16, 16 );

  PenPat( white );
  SetRect( &cabR, 0, 0, 100, 70 );
  OffsetRect( &cabR, 190, 95 );
  PaintRoundRect( &cabR, 16, 16 );

  PenPat( black );
  FrameRoundRect( &cabR, 16, 16 );

  /* finally, draw the box itself */
  PenPat( ltGray );
  SetRect( &boxR, 0, 0, 160, 125 );
  OffsetRect( &boxR, 20, 40 );
  PaintRect( &boxR );

  PenPat( black );
  FrameRect( &boxR );
}
```

Calling DrawTruck() produces the trucklike drawing shown in Figure 2-19.

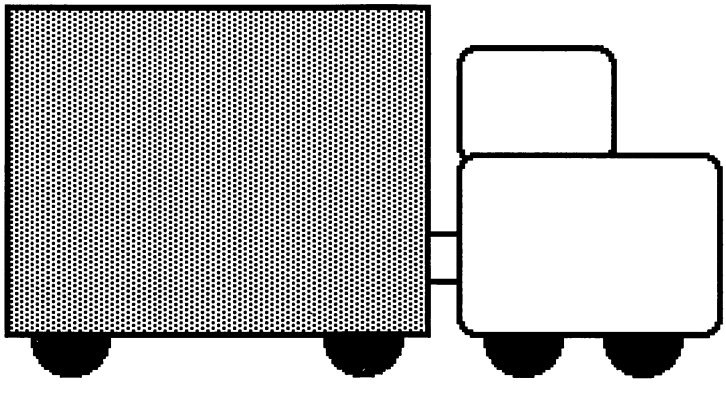
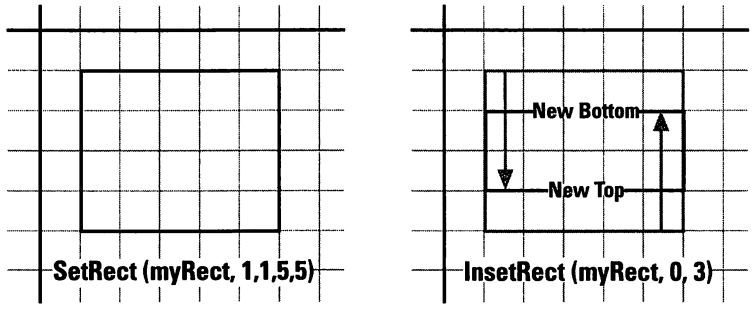


Figure 2-19. Results of calling DrawTruck()

Emptiness

Rects and regions are the only shapes that can be used in point membership calculations. QuickDraw provides two traps, **PtInRect**() and **PtInRgn**(), that are used to determine whether a point (for example, the current location of the mouse) is inside the borders of one of these boundary structures. Since Rects and regions are the only shapes that can be tested for point membership, you may want to create regions out of other shapes that need to be tested for point membership, for example, when deciding if a user clicked inside of a rounded rectangle.

Not all Rects and regions define a boundary: If a Rect's right value is less than or equal to its left value, or if its bottom value is less than or equal to its top value, then that Rect is said to be empty. An empty region is somewhat harder to describe. Basically, a region is empty if it encloses zero points. For example, a region created by a single **LineTo**() call will always be empty. Rects and regions can and should be tested for "emptiness" before using them in drawing and calculation operations. QuickDraw provides **EmptyRect**() and **EmptyRgn**(), which are used to determine whether any point could be inside the borders. In simple terms, nonempty Rects and regions define an area, whereas empty ones define *no* area. Drawing an empty structure will not draw anything: No pixels will be changed, no ink will laid down. Figure 2-20 shows how **InsetRect**() can cause a Rect to become empty.



Modifying a rectangle so that its bottom < top (or right < left) makes it "empty."

Figure 2-20. Empty versus nonempty Rects

Common Region Problems

There are a few mistakes that can trip you up when you are using regions. Checking the following conditions can help to avoid most problems.

- **Was the region allocated via NewRgn() before you tried to use it?** — Trying to use an unallocated region usually results in quick application failure.
- **Is the region empty?** — A number of conditions can cause a region to be empty, which when drawn, yield nothing. Common cases are: trying to create a region with line-drawing operations that don't quite enclose an area; using drawing traps other than **Frame-** to define area; reducing to nothing via **InsetRgn**(); and numerous cases involving the set theory region traps. Use **EmptyRgn**() often; it will save hours of debugging.
- **Is the region corrupt?** — A corrupted region behaves very much like an empty one. Corrupted regions result from operations that would result in a region that is too large (greater than 64K) or too complex (greater than 25 crossings per horizontal scan line).
- **Are the region's coordinates reasonable?** — If the region's bounding rectangle begins at 20,000 instead of 200, you'll need a pretty big window to see it when it is drawn.

▶ Polygons

Polygons are used to draw shapes composed of three or more line segments. Listing 2-12 shows the Polygon data structure.

Listing 2-12. Polygon data structure

```
typedef struct
{
  short  polySize;
  Rect   polyBBox;
  char   polyData[1];
}
Polygon, *PolyPtr, **PolyHandle;
```

In many ways, polygons are similar to regions; they are a Handle-based, variable-length data type, created by opening, drawing lines, and closing, and they have the same five GrafVerb drawing traps as

regions. If you recall Figure 2-6, "QuickDraw shapes and operations", you'll notice that regions have all the functionality of polygons, and more. So why do polygons exist and what are they good for?

- **Data size** — Polygons are based on a compact point list, rather than the scan line change format of regions. Generally speaking, polygons take up much less memory than regions. The region example a few pages back required 1004 bytes; a similar polygon requires only 66 bytes.
- **Accuracy of representation** — Polygons are endpoint-based, as compared to regions, which are scan line–based. Because of this, polygons can be scaled horizontally and/or vertically without inducing severe shape distortion. Polygon scaling is accomplished via the **InsetPoly**() or **MapPoly**() traps, both of which do a much better job of preserving the polygon's proportions than either **InsetRgn**() or **MapRgn**() do of preserving a region's proportions.
- **Greater complexity** — Polygons don't have the 64K limitation that hampers regions.

Unfortunately, polygons lack many of the important features that exist for regions. There is no **PtInPoly**() trap; otherwise, polygons could be used for many of the things that regions are used for. Polygons cannot be combined via set theory operations, as no such traps are implemented. For obvious reasons, polygons cannot define curves.

So what good are polygons? Polygons are good for drawing! Regions are meant to be used primarily for defining areas rather than drawing with them. Together polygons and regions provide solid, sensible tools for the display and management of irregularly shaped areas. Listing 2-13 shows a routine that produces a polygon that defines the same area as the earlier region example in Figure 2-17. Notice that the rectangular area is defined with line drawing rather than a **FrameRect**(); polygons can only be defined using **Line**() and **LineTo**() calls.

Listing 2-13. Example of creating a polygon

```
PolyHandle CreateSamplePoly( void )
{
PolyHandle  samplePoly;

  samplePoly = OpenPoly();  /* open for polygon drawing */
    MoveTo( 50, 50 );       /* sketch a rectangle */
    Line( 80, 0 );          /* with triangles at the corners */
```

```
    Line( -10, -10 );
    Line( 20, 0 );
    Line( -10, 10 );

    Line( 0, 130 );
    Line( 10, 10 );
    Line( -20, 0 );
    Line( 10, -10 );

    Line( -80, 0 );
    Line( -10, 10 );
    Line( 20, 0 );
    Line( -10, -10 );

    Line( 0, -130 );
    Line( -10, -10 );
    Line( 20, 0 );
    Line( -10, 10 );
  ClosePoly();          /* close the polygon, accumulating the */
                        /* result into samplePoly */
  return samplePoly;
}
```

As with creating regions, opening a polygon hides the pen, and makes it visible when the polygon is closed.

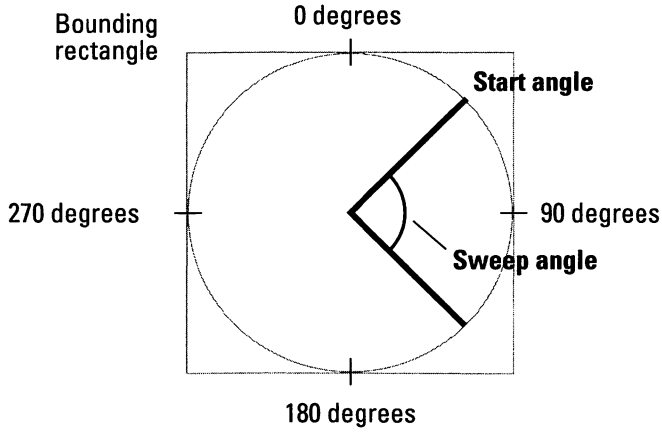
On the Source Code Disk, see last page, you can find **PolyUtils**, a pair of routines that generate regular polygons and vertex point lists, which can be used for a number of useful and interesting effects.

Note ▶

Some of the Polygon GrafVerb traps behave differently from other GrafVerb traps. Unlike all other fill shape traps, **FillPoly**() only fills inside the polygon; the lower-right edges *aren't* filled. **FramePoly**() draws *outside* the lower-right edges of the defined shape, unlike the other frame traps, which draw entirely inside the defined shape.

▶ Arcs and Wedges

Arcs are shapes that are portions of ovals, and wedges are filled arcs. Arcs are similar to RoundRect shapes: As shown in Figure 2-21, they are defined by a bounding rectangle and two integer values that define a start angle and an arc length. These values are specified as clockwise offsets from 12 o'clock.



Example arc, with a starting angle of 45 degrees, a sweep angle of 100 degrees, and a square bounds rectangle.

Figure 2-21. Arc definition

Arc and wedge drawing involves all of the familiar GrafVerbs. However, QuickDraw arcs are fairly limited. With the exception of **Pt2Angle**(), there are no arc calculation traps. Arc GrafVerb traps cannot be used to create regions. Still, arcs have their place, and you will probably find uses for them. Listing 2-14 shows an example of using arcs.

Listing 2-14. Example of arc drawing

```
void DrawArcsAndWedges( void )
{
Rect   arcBoundsRect;
short  i;

  /* set bounding rect */
  SetRect( &arcBoundsRect, 0, 0, 40, 40 );
  OffsetRect( &arcBoundsRect, 20, 20 );

  /* draw six framed arcs */
  for (i=0; i<6; i++)
  {
    FrameArc( &arcBoundsRect, i*60, 60 );
    OffsetRect( &arcBoundsRect, 50, 0 );
  }
```

```
    /* set bounding rect: move below previous one */
    SetRect( &arcBoundsRect, 0, 0, 40, 40 );
    OffsetRect( &arcBoundsRect, 20, 80 );

    /* draw six filled arcs (wedges) */
    for (i=0; i<6; i++)
    {
      FillArc( &arcBoundsRect, i*60, 60, gray );
      OffsetRect( &arcBoundsRect, 50, 0 );
    }
}
```

Executing DrawArcsAndWedges() produces the results in Figure 2-22.

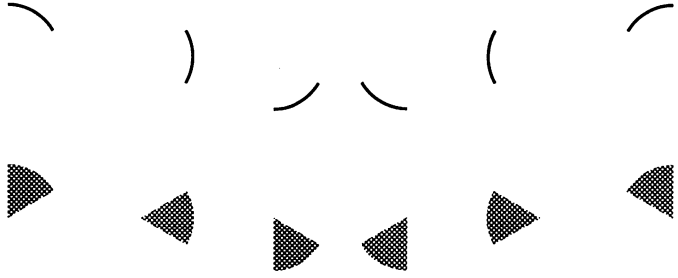


Figure 2-22. Result of DrawArcsAndWedges()

▶ Basic Text

A graphics computer without text capabilities would certainly be severely limited. The Macintosh provides a number of text-handling capabilities to replace the "hard-wired" character interfaces of other computer systems. These facilities involve a handful of managers. QuickDraw's primary role in text processing is the display of single characters and short character strings. But before we delve into text programming, a historical digression is in order.

Text and Computers, Then and Now

A computer system's ability to edit and display text is often taken for granted. Reading and writing from an input/output "hose" are standard programming language and library features familiar to most programmers. Screen-based text I/O involved writing code similar to that shown in Listing 2-15.

Listing 2-15. Supporting text interaction

```
<C>
  char      worldResponse[64];

  printf( "Hello, World! Are you there?" );
  scanf( "%d", worldResponse );

<Pascal>
  yourAnswer: String[64];

  writeln( 'James Worthy and Nick Wirth are soul brothers,
          T/F? ' );
  readln( yourAnswer );
```

In the recent past, computer users (and software developers) didn't expect much more than these capabilities. Computers generally weren't thought of as document processing machines. The successes of the graphical user interface and the advent of the high-resolution laser-based printer and its "realistic" (or if you insist, WYSIWYG) presentation for text services have turned those attitudes around. Users now *expect* high-quality text display on screen and on hardcopy. This complicates matters for programmers. Our little text file and stream editors are no longer sufficient for much of anything beyond writing software itself, and even then these text tools are often found lacking.

Basic Text Traps

At the ground level, QuickDraw has a small number of basic text drawing traps:

```
void DrawChar(
  char   aCharValue );
```

This is as simple as it gets: This trap draws a single character.

```
void DrawString(
  char   *aPString );
```

This trap draws a Pascal string of characters. If you're not a Pascal programmer, a Pascal string is a single unsigned byte indicating the number of characters in the string, followed by the character values themselves. Because an unsigned byte can only represent values between 0 and 255, the maximum length of a Pascal string is limited to 255 characters.

```
void DrawText(
  char   *textPtr,
  short  textLen );
```

This trap is similar to **DrawString**(), except that the number of characters to be drawn is passed separately from the character data. Therefore, the 255-character limitation of **DrawString**() isn't applicable.

```
void TextBox(
  Ptr    textPtr,
  short  textLen,
  short  textJustify,
  Rect   *boxRect );
```

This trap is similar to **DrawText**(), except that it draws text inside of a specified rectangle, with a specified text justification. It's a convenient way to painlessly display text within a given area, automatically computing breaks between lines and words. Note that **TextBox**() is a TextEdit trap, and not part of QuickDraw.

These traps draw characters using the environment's current typeface settings. With the exception of **TextBox**(), these traps don't format text. They don't compute line breaks, nor do they handle special characters, such as returns, tabs, and backspaces. We did warn you that using text wasn't easy.

Important ▶

Characters within a font are defined in fixed-point measurements. Because of this fact, drawing a string with **DrawString**() doesn't always produce the same result as drawing each character with **DrawChar**(). **DrawString**() accumulates the fractional portion as each character is drawn and positions the next character appropriately.

Additionally, calculations are performed to use the font and vertically position characters for each **DrawChar**() or **DrawString**() call. Since **DrawString**() performs these calculations only once for the entire string, it will execute substantially faster than a corresponding sequence of **DrawChar**() calls.

Mixed Typefaces

The appearance of a mixed typeface capability required a whole new layer of text support, which Apple began to support with the introduction of the Lisa computer in 1982. This machine was the first to combine a fast processor with good text-drawing routines and a large base of digitized typefaces.

The Macintosh provides software access to these text facilities in three pieces: the Font Manager, QuickDraw, and TextEdit, which correspond roughly to low-, medium-, and high-level text-handling capabilities. The linkages between these managers and other text-related facilities in the Macintosh operating system are illustrated in Figure 2-23.

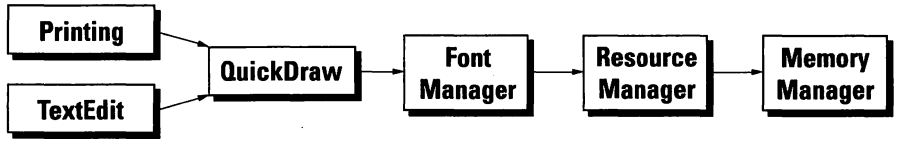


Figure 2-23. Text managers and functional areas

The most familiar text attributes are:

- **Font face or font family** — A group of typefaces that comes from a common design, commonly referred to as a *font*. Every font has a name and an ID.
- **Type size** — The height of the tallest characters plus a bit of spacing between lines. Type size is usually measured in points, not to be confused with the QuickDraw Point data structure. A point is approximately equal to the size of your average screen pixel: 72 pixels per Macintosh SE screen-inch; 72.27 points per inch. This size difference is important when you are drawing strings of characters and will be discussed in later chapters.
- **Type style** — Additional stylizing of the font face. Common stylings are **bold**, *italic*, underline, outline, and shadow. The styles condensed and extended are merely adjustments to the spacing between characters. A lack of style is just "plain." There are a large number of other styles you may be familiar with that QuickDraw doesn't directly support, such as strike-through, double underline, and so forth. If you feel imaginative, you could even make up a few yourself.

Conveniently, QuickDraw provides traps to control each of those typeface attributes:

```
void TextFont(
  short  fontNum );
```

This trap sets the current font ID. There are constants for many of the more familiar font IDs: Chicago, Monaco, Helvetica, and so forth.

In the past, only the font ID was used in drawing. Font IDs are obtained by passing a font name to the **GetFNum**() trap and using the returned font ID. Similarly, a font's name can be obtained from a font ID using the **GetFontName**() trap.

Note ▶

The **SetFontName**() trap is a recent and welcome feature addition, introduced in version 1.2 of 32-bit QuickDraw. It provides the capability to use font names to directly specify the current typeface without first obtaining the font ID. This avoids some of the problems that result when switching between Macintosh systems where the matching font names and IDs are different.

Prior to System 7.0, the System font ID (used in user interface titles and controls) was zero, and the default application font ID was one. These font ID numbers are now obtained using the new **GetSystemFontID**() and **GetApplFontID**() traps.

```
void TextSize(
  short  sizeValue );
```

This trap sets the current font size, in points. A sizeValue of zero indicates the standard System font size, which is typically 12 points.

```
void TextFace(
  short  stylesValue );
```

This trap sets the current typeface style to any combination of the seven standard type styles or to plain. Styles are usually defined in development system interface files as an enumerated type or small set of integers.

```
void TextMode(
  short  modeValue );
```

This trap is identical in functionality to the **PenMode**() trap, except that the results apply to only text drawing rather than shape drawing. The default text mode is srcOr, compared to PatCopy for the pen mode. Like styles, modes are also typically defined as an enumerated type or a small set of integers.

srcCopy writes all of the bits in the character's rectangle. Because srcOr writes only those bits that make up the actual character, using srcOr generally produces more aesthetic results, especially when drawing text on top of previously drawn graphic objects.

Listing 2-16 shows an example of using these text traps.

Listing 2-16. Example of typeface control

```
/*
**draw some simple text, while
**preserving the current text state
*/
void DrawSomeText( void )
{
GrafPtr   savedPort;
Style     savedFontFace;
short     savedFontID, savedFontSize, savedTextMode;

  /* get current text state */
  GetPort( &savedPort );
  savedFontID = savedPort->txFont;
  savedFontSize = savedPort->txSize;
  savedFontFace = savedPort->txFace;
  savedTextMode = savedPort->txMode;

  /* do some texting here */
  TextFont( helvetica );
  TextSize( 24 );
  TextFace( bold | outline );
  TextMode( srcXor );

  MoveTo( 40, 40 );
  DrawString( "\pOoh...   Scary!" );

  /* restore original text state */
  TextFont( savedFontID );
  TextSize( savedFontSize );
  TextFace( savedFontFace );
  TextMode( savedTextMode );
```

```
    MoveTo( 40, 70 );
    DrawString( "\pNot so scary." );
}
```

Executing DrawSomeText() produces the visible result shown in Figure 2-24.

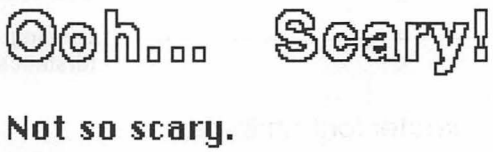


Figure 2-24. Results of DrawSomeText()

By the Way ▶

Apple has created something of a permanent misnomer by calling typefaces "fonts" and styles "faces." For consistency's sake, Apple's usage of these terms will be used from here on out. If you find yourself getting confused, just repeat the mantra *Multi-Font-Size-Style* until you feel at ease.

Typeface Management Details

The Font Manager handles most of the ugly math-intensive details of maintaining and using typefaces. It manages a family of about twenty resource types that contain font information (font metrics) and generates small bit patterns known as *BitMaps* for each character in that font. BitMaps are discussed is some detail in the next two chapters. Each character has a specific horizontal and vertical size along with a placement on the line of text currently being drawn. A bitmapped font is an example of what is known as a *screen font*. Most screen fonts are formatted as QuickDraw bitmaps at a resolution of 72 dots per inch or greater. The Font Manager does such a nice job of managing these sordid details that most applications can ignore them. Figure 2-25 shows the basic attributes of a typeface.

Understand that Macintosh typefaces have never been restricted to these bitmap-based resource formats. The QuickDraw-to-PostScript translation process has been with us since the introduction of the LaserWriter. But the Macintosh Toolbox hasn't been expanded to handle the display of text beyond the Font Manager's resource formats. True,

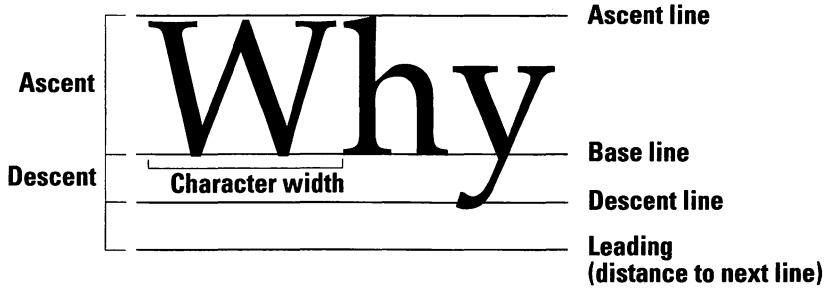


Figure 2-25. Character font attributes

PostScript fonts can be rendered into GrafPorts and displayed, but they generally look inferior to their screen-based cousins.

In early 1990, a number of font management applications made their debut that have all but eliminated the bitmapped dominance of the screen by bitmapped fonts. The tools and standards developed by Apple, Microsoft, Adobe, and a handful of other developers have extended the realm of high-quality typeface far beyond the Macintosh horizon, providing numerous facilities for the import and export of typeface descriptions between the Macintosh and other popular computer platforms. For many applications, the Macintosh developer needs only to follow a small number of simple rules regarding typeface management.

With the advent of System 7.0, QuickDraw fonts can be defined by a series of drawing commands, or *strokes*. These fonts are referred to as *outline fonts* and are handled by the system software called TrueType. Outline fonts have a number of advantages over bitmapped fonts. The most significant advantage is that they can be *scaled*, or drawn at any resolution, without suffering the affliction known as *aliasing*. An example of aliasing is shown in Figure 2-26.

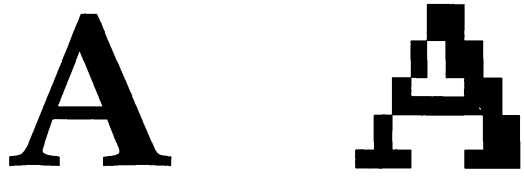


Figure 2-26. Smooth A versus jagged A

Aliasing is the visual symptom of accumulated roundoff error. Aliasing is the scourge of graphics programmers everywhere and is probably the leading cause of WIGWNWIS (What I Got Was Not What I Saw), the mortal enemy of WYSIWYG.

The **GetFontInfo**() trap can be quite useful when drawing lots of small text fragments, or if the exact placement of these text fragments is critical. It fills a FontInfo variable with point measurements of the basic font attributes, which can be used to place lines of text properly in relation to one another. The Font Manager trap can also be employed to provide more extensive information, as well as the ability to generate fractional Fixed-point font metrics.

Advanced Text Management?

Brace yourself for a shock: there isn't any advanced text management! Consider this warning carefully: TextEdit isn't a text editor. TextEdit was originally created to support small amounts of text in dialog windows. Many of its original design requirements remain severe function limitations to this day. Although TextEdit appears chock-full of useful multiple typeface-editing features, it's limited to handling a maximum of 32K of text at a time, and it behaves sluggishly even when handling a lot less. The complexities of multiple typeface editing may induce you to develop some TextEdit-based scheme; you wouldn't be the first to try. If your text-editing needs are moderate, then TextEdit is for you. If you need industrial-strength text editing, you're better off building a TextEdit of your own using the primitive text traps.

▶ Pictures

The Picture is one of the most powerful QuickDraw features. It provides the ability to collect the results of multiple drawing commands in a single package and display them in a single stroke. The Picture data structure is shown in Listing 2-17.

Listing 2-17. Picture data structure

```
typedef struct
{
  Rect   picFrame;
  short  picSize;
  char   picData[1];
}
Picture, **PicHandle;
```

You may notice the similarity between the Region, Polygon, and Picture data structures. It's more than a coincidence: Pictures are created and used in a fashion similar to that employed with regions and polygons. Creating a picture involves the by now familiar three-step process of open-draw-close:

- A picture is opened via the **OpenPicture**() trap, and the pen is hidden. Unlike polygons and regions, a picture doesn't have to be allocated before drawing into it. That's taken care of by **OpenPicture**(). The rectangle argument passed into **OpenPicture**() defines the picture's coordinate system when the picture is later played back.
- The desired shape- and line-drawing traps are called. The results are "tokenized": GrafVerbs get converted into QuickDraw opcodes and the shape data structure values are saved as operands. The size of the open picture is maintained by QuickDraw.
- The picture is closed via **ClosePicture**(). The pen is shown. The resulting picture is stored in memory allocated by QuickDraw, and referenced through a PicHandle variable.

The tokenized format produced by **OpenPicture**() is known as the PICT format and is the primary format for displayable image data on the Macintosh. The PICT format is extremely important to Macintosh graphics programming and is recognized and understood by other Toolbox managers, particularly the Dialog and Scrap (Clipboard) managers. Apple has encouraged application developers to provide some degree of PICT understanding, and developers have, by and large, complied. The result? PICT is a near-universal means of graphics data interchange between many Macintosh applications. In addition to being a RAM-based format, the PICT format is also used as a file format for pictures in resources and data files.

The creation of pictures is subject to some of the same restrictions imposed on region creation. Only one picture can be open at a time in a given GrafPort. However, many pictures can be open in an application, and they can stay open across calls to the **WaitNextEvent**() trap.

Using Pictures

A picture thus created can be drawn in its entirety with a single, simple call to the (what else?) **DrawPicture**() trap. This trap takes a PicHandle and a boundary Rect (or boundsRect) argument. The "boundsRect" can be a copy of the rectangle found in the PicHandle

itself, or it can be altogether different. Offsetting the rectangle allows a picture to be drawn anchored to a position different than the one where it was created. Changing the dimensions (the difference between top and left, bottom and right) will cause a picture to be scaled to fit the new rectangle. Drawing a picture into a new rectangle will cause distortion if the new rectangle is not proportional to the original. Figure 2-27 shows the effects of picture scaling.

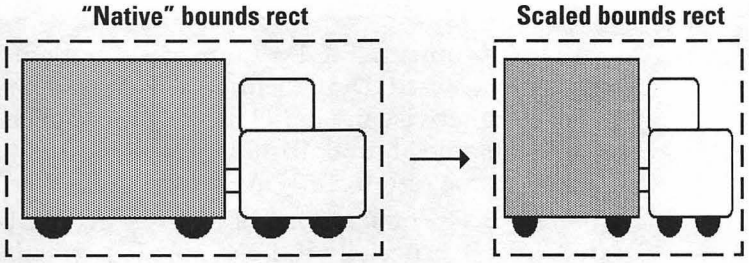


Figure 2-27. Scaling a picture into a different rectangle

Caution ▶

You may have noticed that a picture contains a *picSize* field. It isn't always correct and should never be used to determine a picture's size in memory! Originally, pictures were limited in size to 32K; the 16-bit *picSize* field was guaranteed to hold all legal picture size values. With the introduction of Color QuickDraw, pictures were allowed to be larger than 32K, out of absolute necessity. The proper way to obtain a picture's byte count is shown in Listing 2-18.

Listing 2-18. Obtaining a picture's byte count

```
long FetchPictSize(
  PicHandle    somePicHandle )
{
  if (somePicHandle != NULL)
    return GetHandleSize( (Handle)somePicHandle );
  else
    return (long)0;
}
```

Using Picture Resources

Pictures are commonly accessed from resources. The **GetPicture**() trap retrieves picture data from a 'PICT' resource with a specified resource ID number and returns a PicHandle to that resource's data. If the Picture resource cannot be found, a NULL is returned. The PicHandle thus retrieved can be directly used without any additional work other than determining the desired bounds rectangle to render the picture into.

Caution ▶

As with all resources, 'PICT' resources are purged if their home resource file is closed. **DrawPicture**() will crash if passed a NULL, purged, or otherwise invalid Handle. If an application needs to keep a PicHandle around, it should disassociate it from the 'PICT' resource's home resource file, as shown in Listing 2-19.

There have been some reports of pictures getting purged *during* **DrawPicture**(), especially if the picture is particularly large. Beware!

Listing 2-19. Accessing a 'PICT' resource

```
void DrawPictResource(
  short  pictRsrcID )
{
PicHandle somePict;

  somePict = GetPicture ( pictRsrcID );
  if (somePict != NULL)
  {
    /* note: not necessary under System 7.0 */
    HNoPurge ( somePict );

    DrawPictures ( somePict );
    KillPicture ( somePict );
  }
}
```

Limits of Pictures

Given that pictures have so much flexible functionality, you might wonder when not to use them. Every perfect thing in the world seems to have some thorns, and pictures are no exception; their drawbacks include:

- **Limited command repertoire** — There are some QuickDraw drawing traps whose results cannot be captured by **OpenPicture**(). Familiar graphic operations not currently supported by Apple (particularly curve drawing) can be supported using PicComments and substitute bottleneck GrafProcs (described only briefly in *Inside Macintosh* and various Tech Notes), but making PicComments universally understood by other applications is impossible without universal cooperation. PicComments and bottleneck routines are still very useful.
- **Slower drawing speed** — Pictures can draw slower than the corresponding trap calls. The **DrawPicture**() trap is really a PICT interpreter. As with most interpreters, there is a performance penalty to be paid. The larger and more complex the picture, the greater the penalty.
- **Difficult to analyze** — Once a picture is created, it's cumbersome to extract information regarding its contents.

Note ▶

Until recently, writing complex custom bottleneck procs was the only way to look at a picture's innards. The System 7 version of QuickDraw introduced the Picture Utilities Package, which implements a number of picture analysis tools. This Package-based Manager is examined at length in Chapter 8.

Despite their limitations, pictures are an important and useful part of Macintosh graphics programming.

▶ The GrafPort

We've been discussing shape and text drawing as if it just happens, avoiding any discussion of *where* it happens. Now it's time to talk about drawing *in* something, whether that "something" is a window, a document to be printed, or a drawing area offscreen. As discussed in Chapter 1, only a small number of arguments are passed explicitly to drawing routines; the vast majority of the drawing parameters are maintained by the graphics environment. In the world of Macintosh graphics programming, the state of the graphics environment is described by a **GrafPort**. The results of drawing actions are determined largely by the values contained within fields of a GrafPort. The shape-drawing and calculation traps discussed earlier either modify GrafPort

fields or rely on existing GrafPort field values to make drawing decisions. GrafPorts are most commonly used to define and manipulate the contents of a window, but they are also used to define offscreen drawing and printing as well. A GrafPort contains functional information for the following domains:

- **Pen state information** — Size, pattern, location, pen transfer mode. There are actually three Patterns: the current pen, the filling, and the background (erasing) pattern.
- **Text information** — Font ID, size, style, text transfer mode.
- **Drawing destination** — A BitMap, which may belong to an on-screen window; an offscreen drawing buffer or other hardcopy output device, for example, a printer.
- **Drawing boundary regions and coordinate system** — The *portRect* field, together with its *portBits.bounds* field, defines a GrafPort's coordinate system.
- **Pen color** — Appearance of individual pixels, whether they are black, white, gray, or colored.
- **Addresses of low-level GrafVerb routines** — These routines, or GrafProcs, are pointers to routines that implement GrafVerb functionality.
- **"Scratch" workspace variables** — Used by QuickDraw while accumulating open regions, polygons, and pictures.

The actual C GrafPort data structure definitions looks like Listing 2-20; the fields have been reordered to show groups of fields with related functionality.

Listing 2-20. GrafPort data structure

```
typedef struct
{
  short       device;     /* •• private •• */
  Rect        portRect;   /* drawing boundary and */
  BitMap      portBits;   /* destination information */
  RgnHandle   visRgn;
  RgnHandle   clipRgn;

  Pattern     bkPat;      /* pen state information */
  Pattern     fillPat;
```

```
    Point       pnLoc;
    Point       pnSize;
    short       pnMode;
    Pattern     pnPat;
    short       pnVis;

    short       txFont;     /* text state information */
    Style       txFace;
    short       txMode;
    short       txSize;
    long        spExtra;

    long        fgColor;    /* color information ("old style") */
    long        bkColor;
    short       colrBit;

    short       patStretch; /*  •• private •• */
    QDHandle    picSave;
    QDHandle    rgnSave;
    QDHandle    polySave;
    QDProcsPtr  grafProcs;
}
GrafPort, *GrafPtr;
```

As you can see, a GrafPort contains a wealth of control information. As a general rule, GrafPort field values are maintained by the QuickDraw traps and shouldn't be accessed directly by an application. With few exceptions, there are methods provided for saving and setting the states of each functional domain. In some cases, you will need to read GrafPort field values, particularly for determining text-drawing states.

Note ▶

If there is a trap to access GrafPort fields, use it! Apple's ability to maintain compatibility between your applications, future system software, and hardware rely heavily on this guideline to the extent that you should consider it a rule of Macintosh software development.

Important ▶

GrafPort information is treated by QuickDraw as its primary set of "global information." Virtually all QuickDraw traps operate within the *current GrafPort*. At any given time within a single application, there may be several GrafPorts in existence, but *QuickDraw traps access environment data structures within the current GrafPort only.* This simple fact is extremely important and has numerous ramifications for your Macintosh programming style and technique. At a minimum, keeping track of the current GrafPort is essential. Drawing into the wrong GrafPort often causes unpredictable events, from the humorous (drawing in the wrong window or on the desktop itself) to the deadly (crashing).

Listing 2-21 shows the use of the ever-popular **SetPort**() and **GetPort**() traps in preserving the current GrafPort.

Listing 2-21. Preserving the current GrafPort

```
/*
**preserving the current GrafPort
*/
void DrawInSomeWindow(
  WindowPtrtheWindow )
{
GrafPtr  savedPort;

  GetPort( &savedPort );
  SetPort( (GrafPtr)theWindow );

  /* do some drawing here */

  SetPort( savedPort );
}
```

Important ▶

Notice that we're type-coercing the window reference to be a GrafPort reference instead. Why are we twisting the arm of the compiler (at least one that checks prototypes!) in this fashion? QuickDraw and many other Toolbox managers have traps that take Ptr or Handle arguments whose permissible type may come from a family of type rather than from a single type. A good example of this is that all of the Handle-based types discussed so far can be "measured" via the Memory Manager's **GetHandleSize()** trap. The type-checking facilites of C and Pascal cannot adequately account for this multiplicity of permissible argument types in their respective routine prototyping syntaxes, so the programmer is left with three inferior solutions when using or developing such routines:

- to type-coerce routine arguments on a call-by-call basis
- to declare the argument type to be "void*" (C) or "universal ptr" (Pascal)
- to not prototype the routine at all (C only)

The first of these is the path most often taken by developers and development system interface files. You will encounter this data typing "short-circuit" many times throughout your Macintosh programming adventures. In particular, many of the color extensions to Classic QuickDraw presented in the next chapter rely heavily on these techniques.

▶ Boundaries and Coordinate Systems

Macintosh graphics are drawn (rendered) in Cartesian coordinate space, better known as an X-Y plane. Unlike the coordinate space familiar to high school algebra alumni, which starts in the center at (0, 0) and increases as you move toward the upper-right corner, the QuickDraw coordinate system starts with (0, 0) at the top-left corner of the screen, moving down and to the right for increasing positive coordinates and up and to the left for decreasing negative values. This is consistent with common video-based displays, which draw from left to right, top to bottom.

Note ▶

Interestingly enough, PostScript uses the algebraic coordinate system. Fortunately, the conversions between QuickDraw and PostScript coordinates are handled by print drivers and can be ignored at the application level.

Every GrafPort has its own coordinate system, as shown in Figure 2-28.

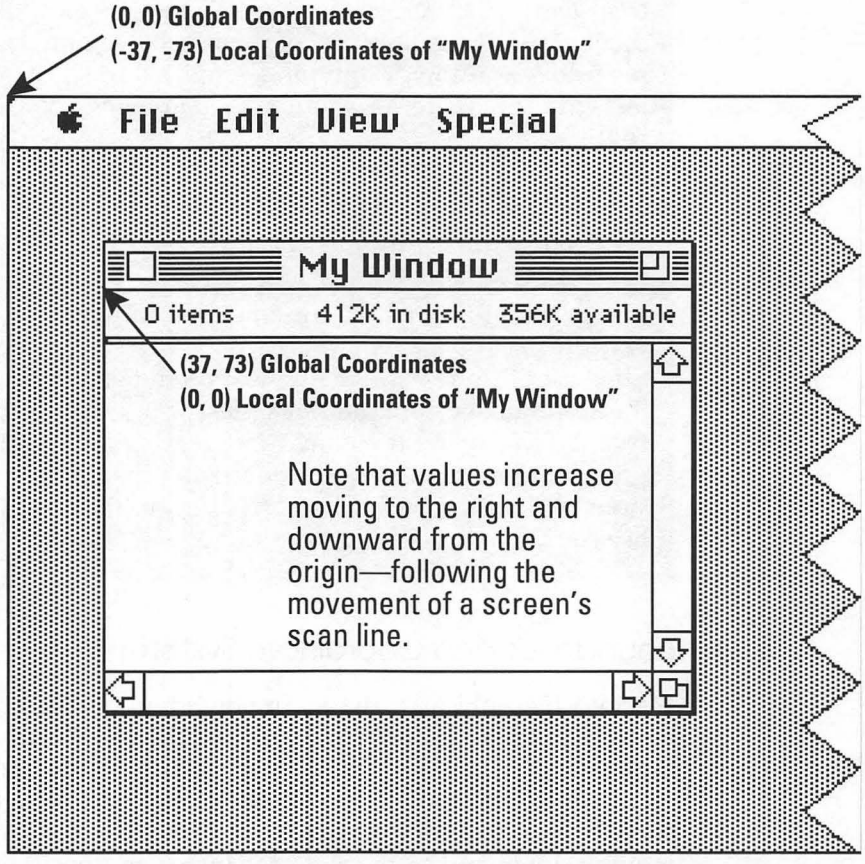


Figure 2-28. QuickDraw coordinate system

The coordinate plane defines (or circumscribes) most QuickDraw operations in these important ways:

- Its point coordinates are expressed as signed 16-bit integers in the range –32,768 to 32,767. For most applications, this will be more than enough area to draw in. The resolution of drawing operations can be adjusted for those applications that require a larger coordinate space, such as scanners and CAD applications.
- It's the yardstick that we use to draw and measure. This means questions like "how long is that line segment?" can be asked when the segment's endpoints can be given as coordinate points. At the same time, the "arbitrariness" of pixels and points as units of measurements allows for a wide range of resolution interpretations to match drawing results to hardcopy devices.
- Its grid lines are Euclidian: They have no thickness. Drawn lines of pixels have thickness. Recall the discussion of points and pixels earlier in this chapter.
- It's used to define the boundaries beyond which drawing (changing of pixels) cannot occur. The concepts of *visibility* and *clipping* are used to transparently limit the effects of drawing operations to the desired destinations, which are usually only a portion of the screen, window, or offscreen document image.

The GrafPort is only the first of four or five data structures that define the graphics environment, as you will soon discover. But the GrafPort is the most important one, if for one reason only: Windows are based on GrafPorts. But more on that later.

▶ Visible Area

The current visible area of a GrafPort (or port) is known as the visRgn, or visible region. It defines the drawable boundary of the digital "paper." Unlike paper, the drawing boundary can be resized as necessary; however, it's strongly recommended that applications refrain from modifying the visRgn directly, and only in rare circumstances should the application modify the visRgn through the port-resizing or the region-calculation traps.

When a GrafPort (or port) belongs to a window, then its *visRgn* is maintained by the Window Manager, which alters it to reflect the changes in the visible portion of the window as it gets covered up by other windows or uncovered by being brought to the front.

▶ Clipping

The visRgn's access restrictions seem substantial; the ability to clip, or temporarily restrict the extent of drawing operation to a small portion of the entire drawable area, is useful and often necessary. The reason for the visRgn limitations is that the clipping capability is assigned to the clipRgn, or clipping region. As with most GrafPort features, the clipping region should be maintained and not just adjusted carelessly. Listing 2-22 shows an example of using clipping to restrict drawing, along with the appropriate clipRgn maintenance.

Listing 2-22. Example of clipped drawing

```
/*
**draw something with clipping,
**preserving the current clipping region
*/
void DrawSomethingClipped( void )
{
RgnHandle   savedClip, newClip;
Rect        thingR;

  /* create an oval region to use for a new clipping area */
  newClip = NewRgn();
  OpenRgn();
    SetRect( &thingR, 20, 20, 100, 80 );
    FrameOval( &thingR );
  CloseRgn( newClip );

  /* copy current clip region */
  savedClip = NewRgn();
  GetClip( savedClip );

  /* draw the unclipped area */
  OffsetRect( &thingR, 20, 20 );
  FillRect( &thingR, ltGray );

  /* show the clipping area-to-be */
  PenPat( black );
  FrameRgn( newClip );

  /* set the clip region equal to the new oval region */
  SetClip( newClip );

  /* blacken the area as allowed by the clip region */
  FillRect( &thingR, black );
```

```
    /* restore original clip region */
    SetClip( savedClip );
    DisposeRgn( savedClip );

    /* finished with the region */
    DisposeRgn( newClip );
}
```

If DrawSomethingClipped() was called without its clipping code present, it would produce a result similar to Figure 2-29.

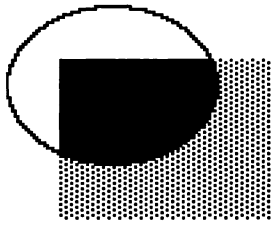


Figure 2-29. Results of calling DrawSomethingClipped()

Without clipping, the filled rectangle would be drawn in its entirety. Clipping is a powerful tool and will be visited again in Chapter 5.

▶ Bottleneck Procedures

QuickDraw owes much of its flexibility to its separation of drawing commands from the software that performs them. This separation is implemented through the mechanism of *bottleneck procedures*. Bottlenecks are procedures that carry out the actual work associated with GrafVerbs and other QuickDraw drawing operations. A GrafPort stores references to its bottleneck procedures in its *qdProcsPtr* field. This is a pointer to a QDProcs structure, which is shown in Listing 2-23.

Listing 2-23. QDProcs data structure

```
typedef struct QDProcs
{
    Ptr  textProc;
    Ptr  lineProc;
    Ptr  rectProc;
```

```
    Ptr  rRectProc;
    Ptr  ovalProc;
    Ptr  arcProc;
    Ptr  polyProc;
    Ptr  rgnProc;
    Ptr  bitsProc;
    Ptr  commentProc;
    Ptr  txMeasProc;
    Ptr  getPicProc;
    Ptr  putPicProc;
}
QDProcs, *QDProcsPtr;
```

QuickDraw has a "standard" trap for each bottleneck procedure: **StdLine**() is the trap that implements standard line drawing, **StdRect**() is used to implement drawing of rectangles, and so forth. A GrafPort, when opened, has a NULL *qdGrafVars* field, which QuickDraw understands to mean "use the standard bottleneck routines." A GrafPort's bottleneck procedures may be accessed directly by reading or assigning the fields within the GrafPort's *qdGrafVars* field, or by calling the **SetStdProcs**() trap to initialize a QDProcs record with pointers to the standard bottleneck routines. An example of this technique is shown in Listing 3-25 in the next chapter.

The beauty of bottlenecks may not be readily apparent, but the flexibility afforded by bottlenecks is fundamental to the flexibility of QuickDraw. Customized versions of the standard bottleneck procs are installed in the current GrafPort when a page is open for printing. This allows QuickDraw to transparently compensate for the differences between graphics display and the requirements of PostScript and other page-description languages found on hardcopy devices.

Applications may also use their custom bottleneck procedures to enhance QuickDraw functionality or simply to obtain more information about the graphic operations it performs. A custom bottleneck may augment the standard bottleneck by calling it directly, either before or after performing whatever operations are required of the custom procedure.

▶ Summary

Understanding contemporary graphic interface designs and principles, and how the Macintosh implements them, is fundamental to simple and successful programming. This chapter presented basic shape-drawing concepts, along with the standard Macintosh terminology.

GrafVerbs were discussed as the implementation and theoretical foundation for basic shape-drawing operations. Rects and regions were introduced as the means to define boundaries for drawing operations and calculations. Other graphics primitives, such as line drawing and basic text display, were also discussed. The chapter also presented the GrafPort data type as the container for basic graphic environment state variables that determine the results of graphic operations. Finally, bottleneck procedures were discussed as the device-independent implementation for all standard QuickDraw graphics functionality.

Many of the subjects in this chapter are discussed in the QuickDraw chapter of *Inside Macintosh,* Volume I. If you haven't read it already, you should consider doing so soon.

3 ▶ Drawing in Color

In Chapter 2, basic QuickDraw concepts and technique were discussed, limited to the world of monochrome drawing: black "ink" on white "paper." To those drawing and calculation constructs, we can now add color, without invalidating the techniques learned in the last chapter. With the introduction of Color QuickDraw, Classic QuickDraw's black-or-white-only "pen" is augmented by a whole new box of colored crayons.

But color graphics programming involves much more than selecting from an expanded range of pen colors: It also involves significant conceptual understanding and programming effort. The concepts and tools necessary for full-featured color graphics programming may be new, even to experienced programmers.

This chapter integrates the theoretical constructs needed to define graphics functionality with the software tools available to the Macintosh programmer in the form of Toolbox managers. The discussion starts with a quick introduction to color description theory. The discussion then moves to the organization of Color QuickDraw and the basics of Macintosh color graphics programming: The key data structures, trap calls, algorithms, and places where Classic QuickDraw has been changed to accommodate color are all discussed in detail.

▶ Software Components of Color

In order to facilitate color display, QuickDraw's implementation and design had to be changed: Many new features and concepts were added, and a few old ones had to be modified. The rationale for this reorganization is as follows:

- **Descriptive power** — The conception of a graphics environment as defined by a single data structure, the GrafPort, has been replaced by a number of data structures.
- **Resource management** — Color display hardware resources require substantially more complex and comprehensive management strategies than its monochrome counterparts.
- **Greater systemwide management role** — The data structures that constitute the interface between graphics operations and display monitors are generally maintained by the application; however, many of the data structures associated with the color characteristics of display monitors are maintained by new Toolbox managers.

▶ Graphics Toolbox Organization

When the Macintosh was introduced in 1984, its graphics functionality was limited to the following features:

- **Basic drawing** — Basic data structures (Points, Rectangles, Regions); boundaries and coordinate systems (clipping and visible regions); shapes (lines, rectangles, ovals, rounded rectangles, polygons, regions, arcs); pens and pen attributes (size, location, pattern, transfer mode); text and text attributes (font, size, style, transfer mode); and pixel memory (bitmapped display memory: one bit per pixel).
- **Windows and user interface** — Onscreen graphics display shared between applications management of boundaries between overlapping windows support simple controls.
- **Printing** — Basic user printing dialogs.

These graphics features were distributed among various Toolbox managers, as shown in Figure 3-1.

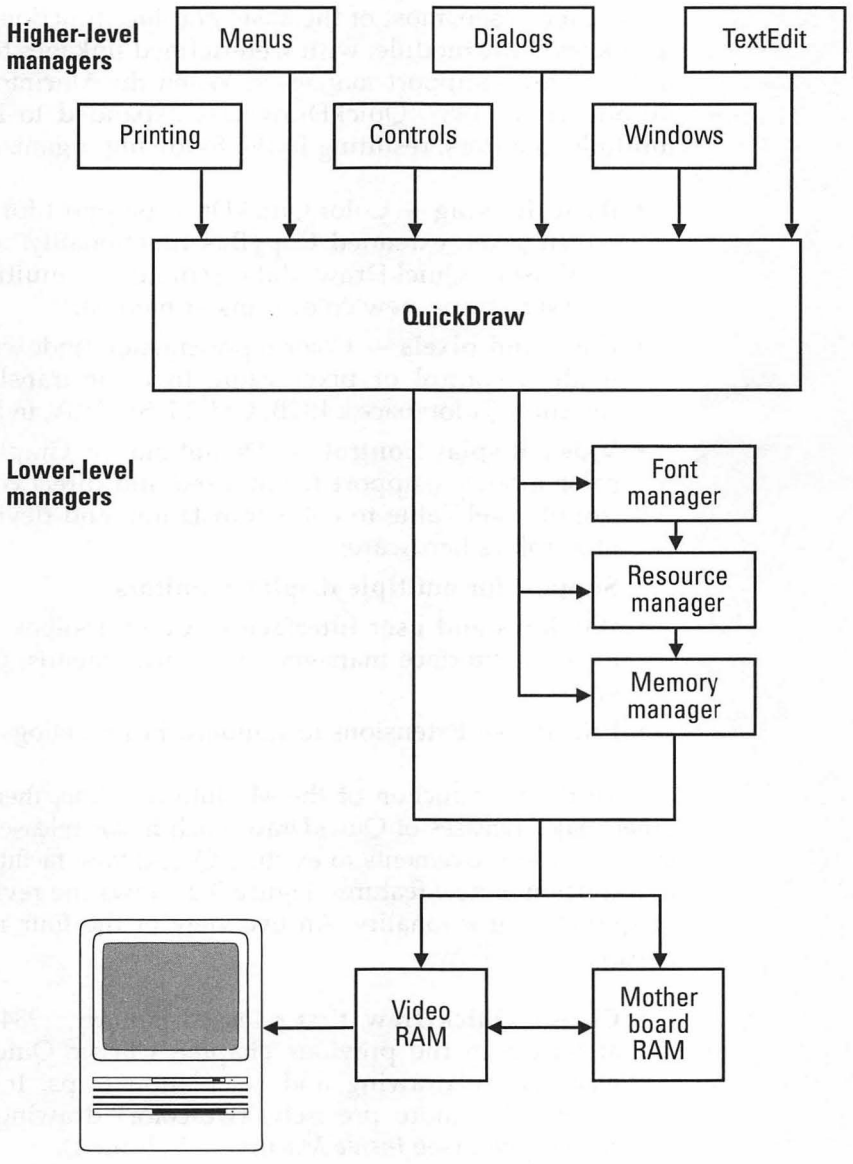


Figure 3-1. Classic QuickDraw functional organization

As you can see, most of the basic graphics functionality is located in the QuickDraw module, with well-defined linkages to a small number of lower-level support managers. When the Macintosh II family was introduced in 1987, QuickDraw was expanded to handle color and multiple monitors, resulting in the following organization of features:

- **Basic drawing** — Color QuickDraw (support for multiple bits per screen pixel, extended CopyBits functionality, multi-bit versions of Classic QuickDraw data structures, multi-bit versions of GrafVerb traps, new color transfer modes);
- **Color and pixels** — Color representation (indexed and direct color models, control of pixel value to color translation) and color selection (colorspaces: RGB, CMY, HSL, HSV, index space, others);
- **Video display control** — Definitions of Graphics Devices and color models (support for indexed and direct color models, control of pixel value to color translation, and device driver control of graphics hardware;
- **Support for multiple display monitors**;
- **Windows and user interface** — Color Toolbox (color extensions for user interface managers: windows, menus, controls, dialogs); and
- **Printing** — Extensions to standard Print Dialogs.

Since the introduction of the Macintosh II line, there have been two other major releases of QuickDraw. Each major release has incorporated significant improvements to existing QuickDraw facilities, as well as the introduction of new features. Figure 3-2 shows the revised organization of graphics functionality. An overview of the four major QuickDraw releases is as follows:

- **Classic QuickDraw**, first released January, 1984 — As discussed at length in the previous chapter, Classic QuickDraw provides basic shape drawing and calculation traps. It supports monochrome (or more precisely, two-color) drawing only: 1 bit per screen pixel (see *Inside Macintosh*, Volume I).
- **Color QuickDraw**, first released March, 1987 — This version adds color capabilities to Classic QuickDraw. It supports multi-color

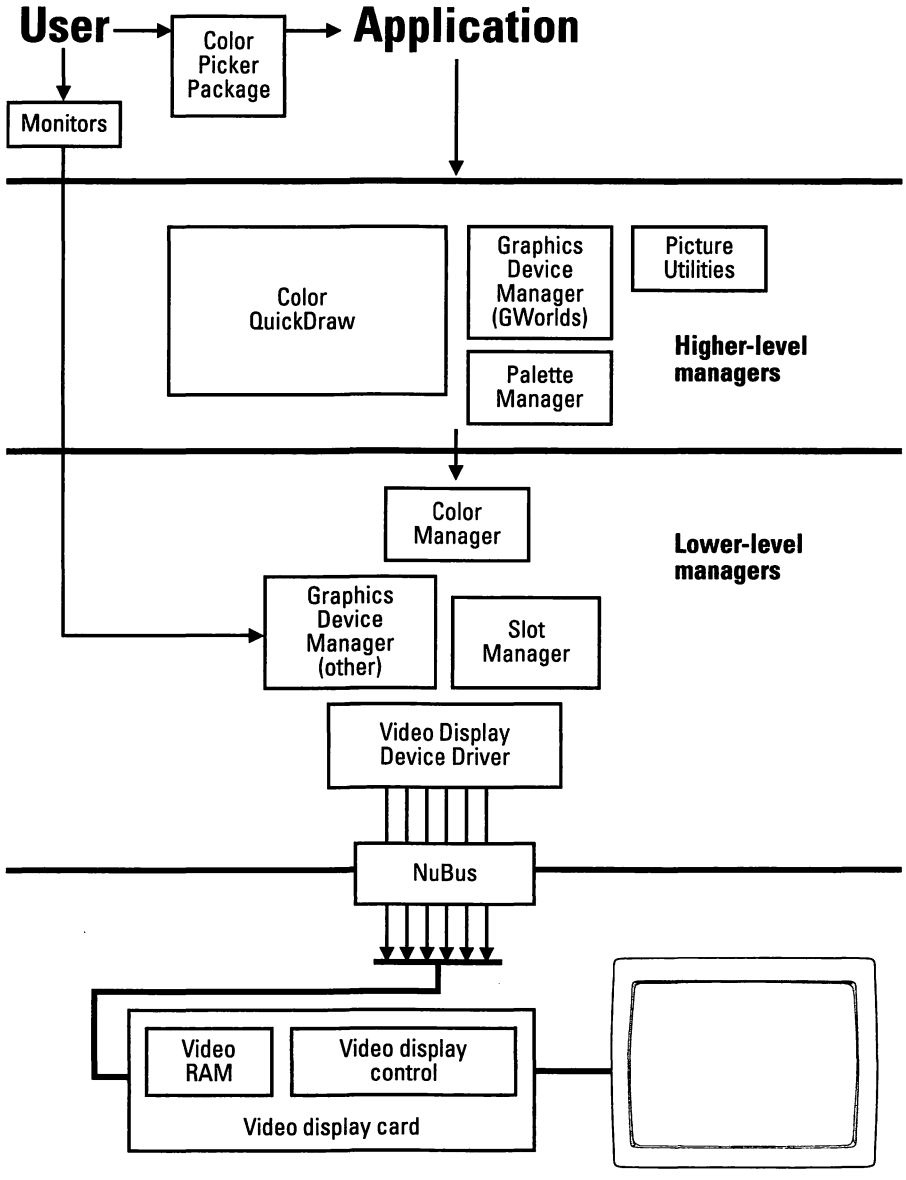


Figure 3-2. Macintosh II QuickDraw functional organization

drawing, using multiple bits per pixel, and introduces expanded versions of certain Classic QuickDraw data structures, particularly the BitMap, Pattern, and GrafPort, which become the PixMap, PixPat, and CGrafPort, respectively. In addition, the existing QuickDraw traps will draw in pen colors other than black and white (see *Inside Macintosh*, Volume V).

- **32-bit QuickDraw**, first released May, 1989 — This version introduces the concept of *true color*, where color specification is as good or greater than what the human eye can perceive. The implementation of true color includes 16- and 32-bit pixels and other "direct" data structures capable of holding more color information than their 8-bit predecessors. Other important 32-bit QuickDraw features include an improved Palette Manager and a reliable and comprehensive offscreen imaging library known as *GWorlds*, as well as more high-level software access to common display hardware features (see *Inside Macintosh*, Volume VI).
- **System 7 QuickDraw**, first released May, 1991 — This version of QuickDraw unifies Classic, Color, and 32-bit QuickDraw. All Color QuickDraw traps and data structures previously unavailable under Classic QuickDraw are implemented, freeing developers from having to implement separate drawing routines for Classic and Color environments. GWorlds are implemented under System 7 Classic QuickDraw. Additionally, a number of new routines and managers are included in this release (see *Inside Macintosh*, Volume VI).

Each major QuickDraw release has seen the introduction of new traps and/or entire managers. Briefly, these new graphics managers perform the following duties:

- **Slot Manager** — The Slot Manager manages the initialization of graphics display cards on the NuBus, as well as supporting the transfer of image data and control information between display cards and motherboard memory (see *Inside Macintosh*, Volumes V and VI, and *Designing Cards and Drivers*, Second Edition).
- **Graphics Device Manager** — The Graphics Device Manager describes the relationships between pixel memory values and visual display characteristics. It supports multiple display monitors and provides access to standard and specialized graphics display hardware features, particularly display color mapping and pixel depth

control (see *Inside Macintosh*, Volumes V and VI, and *Designing Cards and Drivers*, Second Edition).

- **Color Manager** — This manager really should be named the Color Mapping Manager. Its primary role is to make determinations of color mapping, the process of matching color values stored in QuickDraw data structures to those currently available on the current graphics device (see *Inside Macintosh*, Volume V, although the chapter is sketchy).
- **Palette Manager** — This manager arbitrates requests for displayable colors on a per window basis across all active applications. Working with the Window Manager, it determines which colors are available to each window at any given moment, making the necessary Color Manager trap calls to update the affected display devices (see *Inside Macintosh*, Volumes V and VI).
- **Color Picker** — This package implements a simple dialog interface for color selection on any display monitor, even on monochrome displays. It also provides a family of utility traps to facilitate simple value conversions between standard color description systems, or colorspaces (see *Inside Macintosh*, Volume V).
- **Picture Utilities** — This package, introduced with System 7.0, allows applications to analyze and gather statistics on the shape, text, and color operations that contained in pictures. It also provides color and histogram analysis functions on pixel images (see *Inside Macintosh*, Volume VI).

Like many aspects of the Macintosh, color graphics support involves a number of interrelated concepts, which serves to complicate discussions about the various subtopics.

Aside ▶

Throughout the book you will find us referring to these managers collectively as the *Macintosh Graphics Toolbox* or even just *QuickDraw*; this is because many functional components are so tightly interrelated and similar that distinctions between them are almost academic. The functionality of each manager will be addressed in detail, but it's important to recognize each one as an integral part of a larger graphics package.

In general, the new Color Managers interact with data structures differently than Classic QuickDraw. The following aspects are particularly noteworthy:

- **Memory management** — Most important data structures need to be resized dynamically and consequently are Handle-based. Greater care must be taken to ensure that variables of these data types are created properly and disposed of reliably.
- **Nested structures** — The number and depth of data structure nesting is greater, particularly with important Handle-based data types. This adds to the application's memory management responsibilities.
- **NULL pointer references** — On the good side, the new color traps recognize NULL Ptr and Handle arguments, either by reporting them as errors and exiting without processing, or by treating them as a special value.
- **Error reporting** — Most of the newer traps report errors encountered during their computations indirectly through the **QDError**() trap. Additionally, some Classic traps now register their failures with this trap. Most of these errors are a result of insufficient heap memory or an inability of the graphics hardware to provide the requested graphics service.

▶ Multi-Bit Pixels

The Classic Macintosh family machines, with their black-and-white monitors, have pixels that are either black or white. The informational content of such a pixel can be represented with a single bit of RAM. We use the terms *pixel value mapping* and *color mapping* to refer to the translation of pixel values in memory into visible, displayable quantities. The Classic Macintosh value mapping is quite simple: If a bit holds a one, then the corresponding pixel is black, or filled; if the bit holds a zero, the pixel is white, or blank. Mathematically speaking, value mapping is arbitrary. The black-one, white-zero mapping was chosen by Apple simply because of its similarity to the "real world" of pencil and paper. If you can remember back to 1983 when Apple released the Lisa, most other computer systems chose the reverse mapping: one for white, zero for black.

Multi-bit pixels are necessary for drawing in more than two colors. In a color system, pixel values must represent more than two different colors and therefore must occupy more than 1 bit of memory.

Suppose that a pixel's size (or depth) is increased from 1 to 2 bits. This pixel can now assume any one of four possible values, twice as many as before. Each value can theoretically represent a distinct color. As with integer value representations, the number of possible colors represented by a given pixel depth is

number of different colors = $2^{(\text{number of bits per pixel})}$

Color QuickDraw facilitates 1-, 2-, 4-, or 8-bit pixels; 32-bit QuickDraw adds 16- and 32-bit pixel depths. The most common pixel depth, or size, is 8 bits, which allows each pixel to select from 2^8, or 256 possible colors.

▶ Color Models

We've mentioned the increased information-carrying capacity of multi-bit pixels without regard to how those pixel values are translated into displayable colors. There are a number of methods for performing this translation, each of them requiring a particular configuration of pixel data in memory. We refer to these methods as *color models*.

Color, 32-bit, and System 7 QuickDraw together support four distinct color models: *classic*, *fixed*, *indexed*, and *direct*. Of these, the indexed and direct models are most important: They represent two different ways of interpreting pixel values as color information. The primary difference between these two models is the number of simultaneously displayable colors allowed: The indexed model is used for pixel depths of 1, 2, 4, and 8 bits per pixel; the direct model supports pixel depths of 16 and 32 bits.

▶ Indexed Color

In an indexed color model, each pixel value serves as an index (or lookup) into a table that holds the actual RGB color information. The ColorTable therefore defines the pixel's color; without it, an indexed pixel value doesn't describe any color at all. The relationship between different indexed pixel values may or may not be arbitrary. Adjacent pixel values may reference color values that are dramatically different or extremely similar.

An advantage of the indexed model is that the relationship between pixel and color values can be changed simply by altering the ColorTable's color values. The indexed color model has a few other names;

it's sometimes called the *CLUT (Color LookUp Table) color model*, or the *mapped color model*.

There are many applications where the indexed model doesn't provide enough color specification. For example, a photograph of a forest might have a few hundred shades of green; the 256-color limitation of an 8-bit representation causes such images to lose some shades of color, making the picture look blotchy and unreal.

▶ Direct Color

With the advent of Color QuickDraw Version 1.0, a new pixel value-to-color representation, called the *direct color model*, was introduced whereby pixel values contain actual RGB color information rather than ColorTable index values. Direct pixels are called "direct" because their values can be directly interpreted as RGB color information without the "indirection" of a CLUT transformation. Direct pixels contain 24 bits of color information in 32-bit pixels, and 15 bits of color information in 16-bit pixels. The differences between the pixel value representations in indexed and direct devices are shown in Figure 3-3.

Note that in Figure 3-3, both an index value and a direct value can be used to produce the same color (represented by the 48-bit RGB value). The main difference is that an index value relies on the CLUT to store the actual value.

Direct pixel images can coexist in an application's heap with indexed images, but each individual image must be entirely composed of one kind of pixel. Each color model has its advantages and drawbacks: The trade-offs between them will become clear as you read on. It's up to you to choose either indexed or direct pixels for your application's graphics requirements.

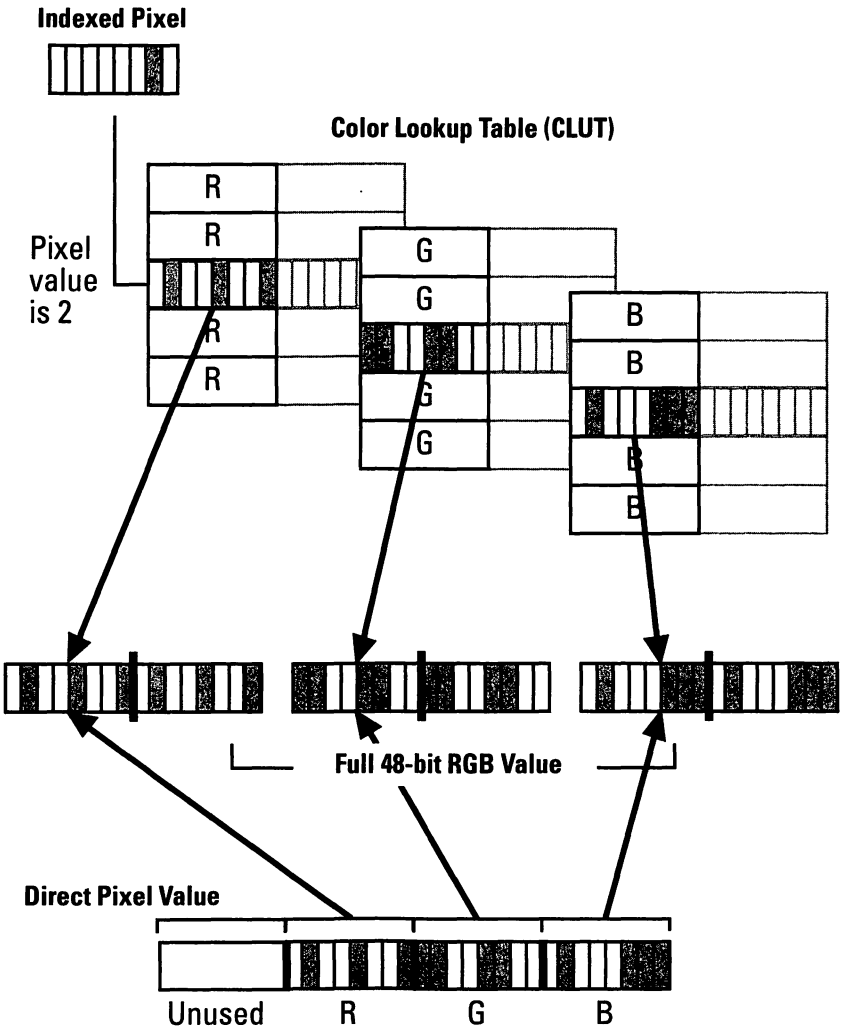


Figure 3-3. Indexed versus direct color interpretation

▶ Fixed Color

The fixed model is essentially the same as the indexed model, except that a fixed GDevice's ColorTable cannot be changed. Its design is intended to represent hardcopy devices where the available colors cannot be changed. This model combines elements of both indexed and fixed models. Due to the relative lack of popularity among Macintosh hardware offerings, we will not describe the fixed model in further detail.

▶ Classic Color

The Classic model is the color model used by the Classic Macintosh family. This model is quite similar to the fixed model and has eight constant color choices. Yes, it's true: The Classic black-and-white Macintoshes can draw in color to a limited degree — it's just that no one could see the colors on their monochrome screens. The Classic color model has limited support for bit-plane-oriented output to hardcopy devices.

The Classic color model is supported under Color QuickDraw within both the indexed and direct color models. With respect to color selection, its functionality is quite limited, compared to the indexed and direct color models. Despite its limitations, however, the Classic color model is ideal for applications that require a limited number of fixed color choices.

Why are there all these color models? Most of the reason lies in the fact that graphics hardware architectures have been evolving, and that software color models have been created to match the hardware. Pixel image organization is a reflection of the raster-scan orientation of display hardware architecture. The indexed and direct pixel models reflect two different strategies in graphics display hardware. The most common type of display hardware currently in use employs indexed color; the monitor can display a limited number of colors simultaneously. The colors available for display are stored on the display card, and can be changed to match the colors of the image to be displayed. Direct display devices can display an essentially unlimited number of colors simultaneously within a color range, or *gamut*, but they're more technologically complex and therefore more expensive. The graphics display market is quite competitive, and as a consequence, 16- and 24-bit direct display controllers will become more affordable and find their way into more and more machines.

▶ Basic Color Theory

The color theory presented in this section is straightforward and includes frequent references to the Macintosh programming paradigm. We've included it to establish a common ground for the discussions of the various color programming data structures and tools that follow. A detailed discussion of the physics and physiology of color perception is beyond the scope of this book; there are many good sources of information on these fascinating topics, some of which you can find in the Bibliography.

▶ RGB Color

Color is a matter of perception. Light energy (electromagnetic radiation) falling on your retina excites your eyes' light-sensitive color receptors, which send information to your brain via your optic nerve; this stimulus is perceived as color.

There are three components of visual color:

- **Hue** — This is what appears to us as "pure" color. Hues run from red at one end to violet at the other. A rainbow is a sequence of bright hues starting from low (red) values to high (purple) values. Hue is also referred to as *chroma*.
- **Saturation** — This refers to the amount of hue in the color. Less saturated colors are "whiter" than those that are heavily saturated. For example, pink is a red of low saturation.
- **Brightness** — Brightness is similar to saturation, except that it denotes the amount of black or gray in the color. For example, a dark red may be as saturated as a bright red.

For better or worse, color display monitors use an entirely different color paradigm for defining color values: the RGB color model. The basic RGB technique can be directly observed by placing a magnifying glass on a color TV set or color monitor. You should be able to see that each screen pixel is actually composed of three closely arranged spots or stripes. The three pixel spots, or components, are red, green, and blue. A wide range of colors can be achieved by mixing these three colors. When two light sources are geographically so close together that the limits of your visual acuity cause you to see them as only one source, then the colors of the emitted sources are blended and perceived as a single color that is different from either of the original two sources.

Note ▶

In the graphics display world, the combination of colors yields a color that is brighter than any of the original colors; that is, the sum of color mixing tends toward white light. This is known as *additive color.* In the world of paints, inks, and pigments, color mixing tends toward black. This is known as *subtractive color.* Color QuickDraw provides both additive and subtractive color mixing operations.

▶ Chromaticity and Primary Colors

Figure 3-4 is a black-and-white representation of the CIE Chromaticity Diagram. Inside the wing-shaped area between the axes are all the colors (hues) visible to the human eye, disregarding differences in light intensity. The visible colors vary continuously throughout the region with the greenish hues at the tip of the wing, the blues on the lower left, the reds on the right, and white in the middle surrounded by the pastels. See CP-1 on the color plate for a color representation of the chart.

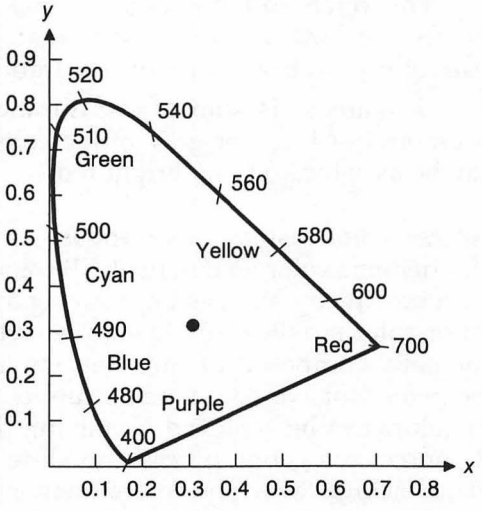


Figure 3-4. CIE chromaticity diagram

The chromaticity diagram is useful to us because it describes the mixing of colors. If you choose two colors — two points on the chromaticity diagram — then the colors that can be achieved by mixing them are the points on the line segment that you can draw from one point to the other. Take, for example, the two colors at RED 1 and WHITE 2 shown in Figure 3-5.

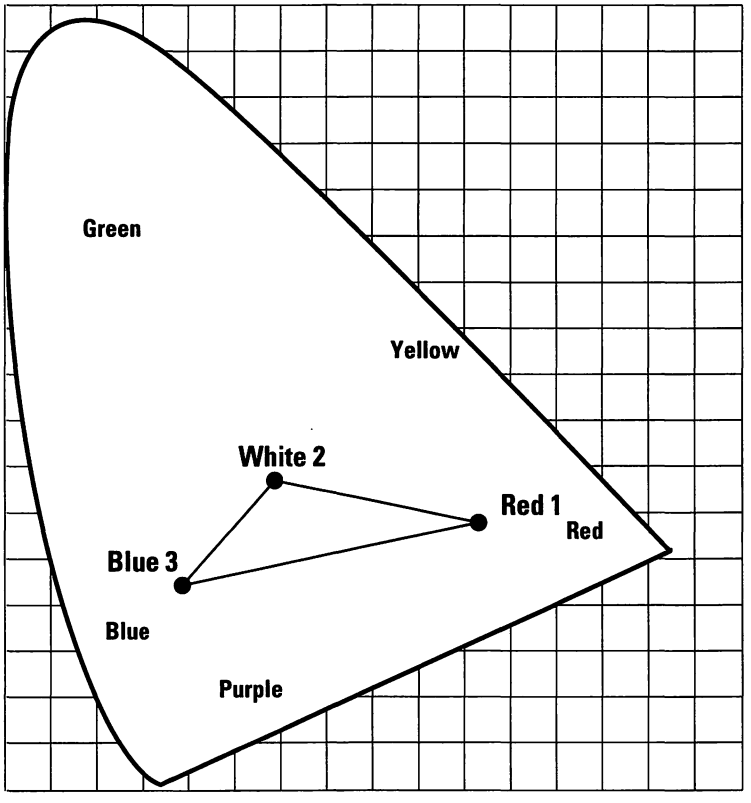


Figure 3-5. Example of color chromaticity and mixing

These colors can be combined to create any of the hues on the line between them. The result of combining them depends on the relative intensities of the two colors. If you mix in more red than white, the result lies closer to RED 1 than WHITE 2. This result is true no matter which two colors are used as primary colors. If a third primary color is used for mixing, like BLUE 3, then the result of mixing all three colors lies along the line between BLUE 3 and the point that was the

result of mixing the first two primaries. In this example, the result is a color in the violet range. The resulting colors depend on two factors: the choice of primary colors and the relative intensities of those primary colors. The gamut of colors available as a result of mixing three primary colors are those colors at the points inside the triangle created by drawing line segments between the three primary colors.

Two important color mixing concepts are illustrated by the chromaticity diagram:

- The choice of primary colors is arbitrary. Color mixing can be accomplished using any colors in the diagram. Four or nine primary colors could be used instead of three. Because the area of visible chromaticity is roughly triangular, three primary colors are sufficient to provide a broad gamut of available colors — again, refer to Figure 3-5. It's equally easy to decide that red, green, and blue are good candidates as primary colors, simply because their locations in the visible area defines a triangle that covers the largest possible area of chromaticity. In subtractive color environments, the primary color choices are cyan, magenta, and yellow, all of which are relatively brighter than their RGB counterparts. Many of us were taught early on that the primary colors are red, yellow, and blue. These are reasonably good choices, but they define a much narrower range of displayable colors that can be achieved through color mixing, simply because green is closer than yellow to the highest, centermost chromaticity point. Again, the term "primary" is somewhat arbitrary. For practical reasons, the computer world has chosen red, green, and blue for primary colors, and it isn't likely to change anytime soon!
- Because of the shape of color space as represented by the chromaticity diagram, choosing a tri-color primary system for mixing colors means that there are visible colors that can't be created by mixing the three primaries. In Figure 3-5, any color inside the triangle can be created from some proportional mixture of the primary colors, but all colors outside the triangle *cannot* be created by *any* combination of the three primaries. In fact, no three colors can be selected from the visible colors and then used as tri-color primaries to map all of the visible colors.

▶ RGB Color Values

The RGBColor data structure is used by Color QuickDraw to describe RGB color values and is shown in Listing 3-1.

Listing 3-1. RGBColor data structure

```
typedef struct
{
   SmallFract  red;
   SmallFract  green;
   SmallFract  blue;
}
RGBColor;
```

An RGBColor variable specifies a color by describing the amounts of each component in the color. The SmallFract type is a 16-bit unsigned quantity that represents a fixed-point decimal value between 0 and 255 + 255/256, in increments of 1/256. A SmallFract can also be thought of as a 16-bit unsigned integer ranging from 0 to 65535, as shown graphically in Figure 3-6.

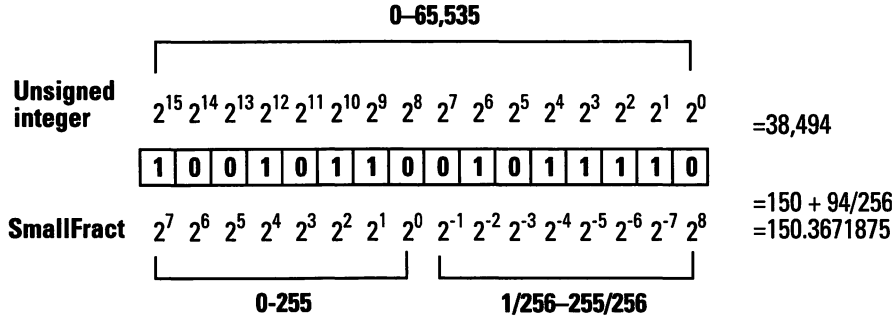


Figure 3-6. SmallFract and unsigned integer value ranges

Because the RGB color system has three components for each color, we can use a three-dimensional model to explore some properties of Color QuickDraw's displayable colors. To help you visualize the gamut of displayable colors, let's introduce a popular color abstraction, the *RGB color cube*, as shown in Figure 3-7.

The color cube is a representation of a color space in which three orthogonal axes measure the three color components (red, green, and blue) as nonnegative real numbers. At any point within the cube, the

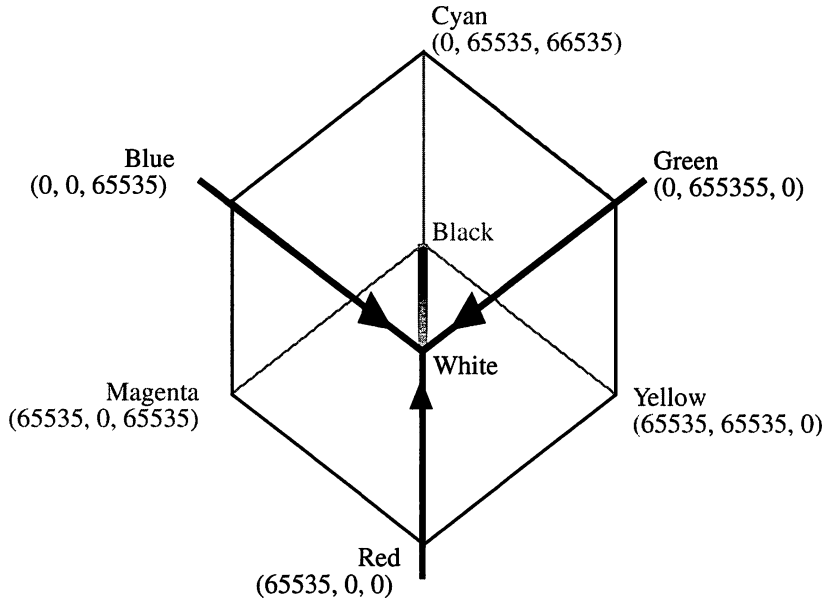


Figure 3-7. RGB color cube

color of the point is the result of combining red as measured on the R axis, green on the G axis, and blue on the B axis. Black is at the corner of the cube where RGB = (0, 0, 0); in other words no red, no green, and no blue — the absence of light. White is the color at the diagonally opposite corner of the cube. But what are the RGB coordinates of white? Black was intuitive: no light, black. But how much red, green, and blue does it take to make white? The answer is, enough. It takes enough electromagnetic radiation composed of equal parts of red, green, and blue wavelengths to make your brain perceive white; more than that just makes you squint. Similarly, black isn't really an absence of light (there's always some light either being reflected or emitted), it's just a level of light composed of equal parts red, green, and blue that is so low that your brain perceives blackness.

The manufacturer of a graphics display monitor defines the absolute range of colors that its devices can display. QuickDraw maps the available range of displayable colors (whatever that may be) into the range of integer values from 0 to 65535 for each of red, green, and blue. A range of 65536 different shades of red is far more than any eye can distinguish and more than most electronic devices can discern; 256 different reds are plenty. For this reason, 24-bit color is sometimes called *true color.*

The RGB color cube that describes the colors available on the Macintosh is granular because the numbers on the R, G, and B axes are integers; in other words, there are points on the R axis at 0, 1, 2, 3, and so on, but not at 1.5 or 2.83. The cube itself is made up of 2^{24} discrete, different colors (although adjacent colors don't differ by much).

By the Way ▶

A word about the term *color*. In this discussion, all combinations of red, green, and blue are referred to as "colors" even though black, white, and all shades of gray are included. Gray is defined as any color whose red, green, and blue components are equal. For example, white is defined as RGB = (65535, 65535, 65535) and black, not surprisingly, is RGB = (0, 0, 0). In the RGB color cube, the range of grays are those colors on the diagonal line from black (0, 0, 0) to white (65535, 65535, 65535), where all points satisfy the equation R = G = B.

▶ Limits of Color Display

The actual range of displayable colors is further limited by the architecture of the graphics display control hardware on the card that drives the monitor. The value of each component gets translated into a voltage that is applied to the corresponding color gun in the monitor. Realize that the simultaneous display of 16.8 million colors would require a 4096-by-4096-pixel monitor!

These are two basic distinctions between direct and indexed graphics devices:

- **Direct color means more colors at once** — A 24-bit direct display controller doesn't have the indexed controller's restrictions on the number of colors displayed: A pixel displayed on a 24-bit direct display can have any value in RGB colorspace, regardless of any other displayed pixel's value. This offers a lot more variability in building a picture for display. A monitor controlled by a direct display card has the same range of displayable color values as an indexed display controller: It has the same restrictions imposed by the choice of primary colors used by the monitor, and it doesn't have any greater resolution within the gamut of displayable colors. In addition, it uses the same 24 bits of precision in memory while simultaneously displaying a wider range of colors.

- **Indexed color means more accurate color** — The range of displayable RGB colors, or *colorspace*, is precisely the number of possible combinations of the significant component values, which can be determined for a 24-bit pixel image as follows:

  number of displayable colors
  = $2^{(\text{number of bits per pixel value component})(\text{number of components})}$
  = $2^{(8 \times 3)}$
  = 2^{24}
  = 16777216 = 16.8 million (approximately)

  Surprisingly, this calculation yields a different result for indexed pixel images, since those values are determined by the resolution of the CLUT's ColorSpec entries (48 bits) and not the pixel values themselves (16 or 24 bits).

  number of displayable colors
  = $2^{(\text{number of bits per CLUT entry component})(\text{number of components})}$
  = $2^{(16 \times 3)}$
  = 2^{48}
  = 16777216 × 16777216 = <a really big number!>

▶ Color Picker

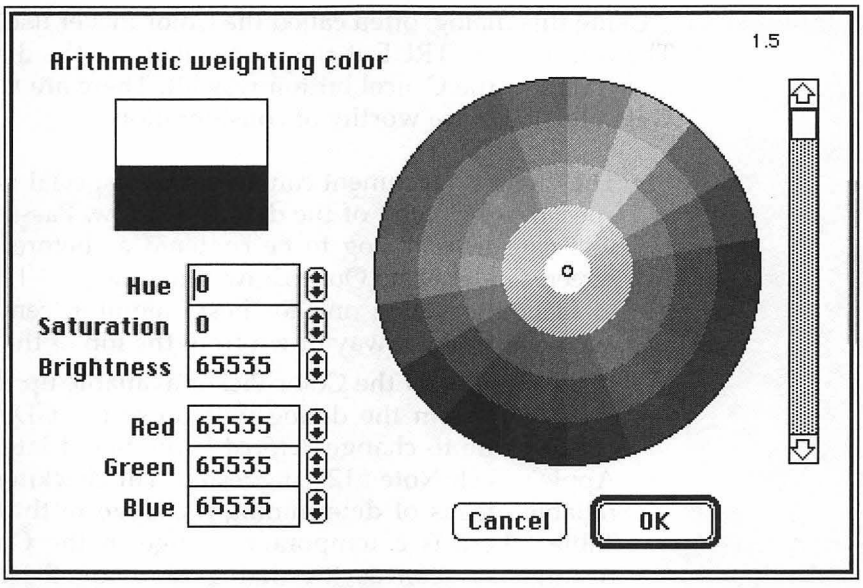
Although RGB color is Color QuickDraw's standard color currency, there are other ways of describing colors. The Color Picker is a package-based manager that contains a number of colorspace conversion traps. The Color Picker also includes a dialog-based color selection interface that presents a comprehensive yet intuitive means of selecting color values in RGB and hue-based colorspaces.

▶ Color Picker Dialog

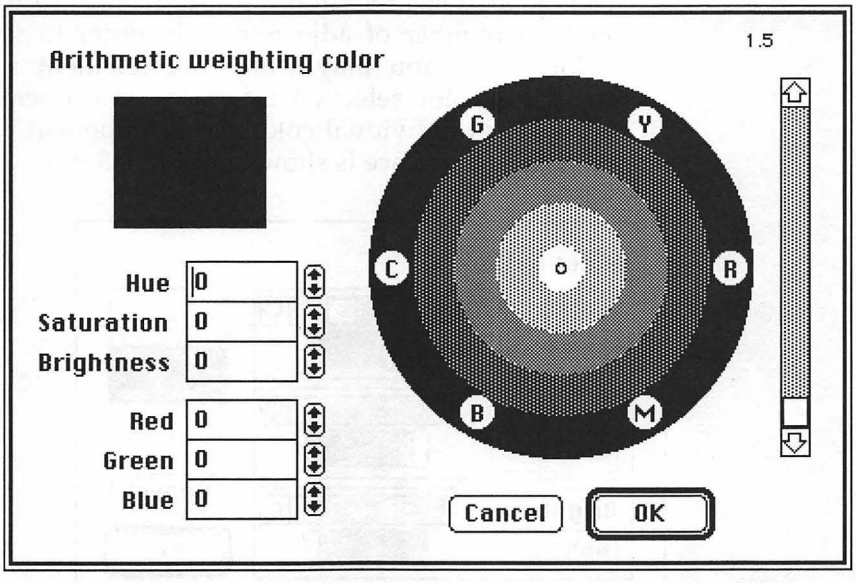
The color selection dialog is accessible from a single trap, **GetColor()**. Its interface is as follows:

```
Boolean GetColor(
  Point     where,
  char      *promptString,
  RGBColor  *inColor,
  RGBColor  *outColor );
```

This trap displays the dialog box shown in Figure 3-8.



Version on color computers



Version on black and white computers

Figure 3-8. Color picker dialog

Using this dialog, often called the Color Picker itself, is fairly simple. The trap returns TRUE if the user exited via the dialog's OK button, and FALSE if the Cancel button was hit. There are a few details about **GetColor**() that are worthy of consideration:

- The "where" argument can accept two special values that select a special positioning of the dialog window. Passing the value (0, 0) will cause the dialog to be centered as before, but on the main screen. Under 32-bit QuickDraw, the value (–1, –1) directs **GetColor**() to place the dialog on the "best" monitor, centered horizontally and a third of the way down from the top of the display.
- In all versions of the Color Picker available up to System 6.0.5, an "OK" exit from the dialog will cause the GDevice's ColorTable *ctSeed* value to change. (*ctSeed* is discussed later in this chapter.) Apple's Tech Note #120 suggested that checking this value was a reliable means of determining a change in the GDevice's ColorTable. There is a temporary change in the ColorTable, but it's restored as soon as the dialog is removed from the screen. A "Cancel" response doesn't cause a *ctSeed* change.
- The Color Picker dialog box is huge, covering a large portion of a 640-by-480-pixel monitor. It's also modal and requires the user to make a number of adjustments in order to specify the desired color value. You may want to implement a smaller, simpler modeless color selection interface with a scrollbar or slider to adjust each individual colorspace component value. An example of such an interface is shown in Figure 3-9.

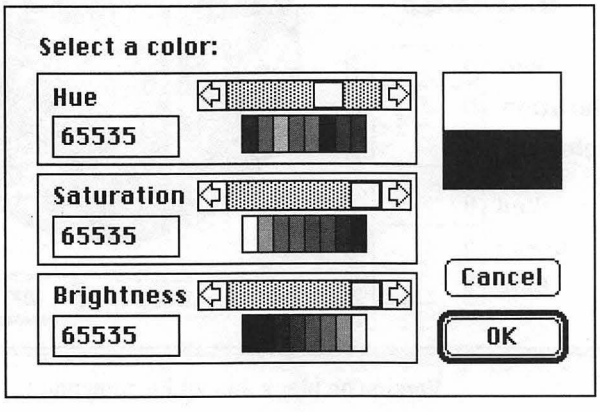


Figure 3-9. Custom color selection window

▶ HSV Colorspace

One such alternative to the RGB colorspace is the HSL colorspace, which stands for hue, saturation, and lightness. Lightness is also known as brightness or value. As discussed earlier, this colorspace aptly describes the human perception of color. The Color Picker package provides two traps, **RGB2HSV**() and **HSV2RGB**(), to translate values between RGB and HSV color spaces. The formulas for these conversions are laborous and not that remarkable. Using the conversion traps is highly recommended.

An interesting use of the HSV colorspace is to generate rainbows. An example of this capability is shown in Listing 3-2, which demonstrates how to construct a "rainbow" array of RGBColor values.

Listing 3-2. Building a "rainbow" array of colors

```
#define MaxSmallFract  (SmallFract)0xFFFF

void MakeRainbowArray(
  RGBColor *theColors,
  short    entryCount )
{
HSVColor theHSVColor;
Fixed    entryRatio;
short    i;

  theHSVColor.saturation = MaxSmallFract;
  theHSVColor.value = MaxSmallFract;

  for (i=0; i<=entryCount; i++)
  {
    theHSVColor.hue = Fix2SmallFract(
      FixRatio( i, entryCount ) );
    HSV2RGB( &theHSVColor, &(theColors[i]) );
  }
}
```

Listing 3-2 operates upon an array of RGBColors. The common way of maintaining a list of RGB values is through the use of the ColorTable data structure, which is discussed later in this chapter.

▶ CMY Colorspace

Another standard colorspace is the cyan-magenta-yellow (CMY) colorspace. CMY space is simply the inverse of RGB space: CMY color values can be obtained by bit-complementing RGB values: R = C'; G = M'; B = Y'. For example:

RGB(45877, 699, 11004)
= CMY(45877', 699', 11004')
= CMY(19658, 64836, 54531)

This colorspace is quite often used when performing subtractive color operations. Color separations are a critical part of pre-press image production and other high-end desktop publishing applications.

▶ ColorTables

It's often necessary to maintain lists of RGBColor values. In the QuickDraw environment, such a list is called a *ColorTable*. Each list item, or entry, is composed of an RGBColor value and an optional index value. QuickDraw expresses a ColorTable with the data types, as shown in Listing 3-3.

Listing 3-3. ColorSpec and ColorTable data structures

```
typedef struct
{
  short    value;
  RGBColor rgb;
}
ColorSpec;

typedef ColorSpec CSpecArray[1];

typedef struct
{
  long        ctSeed;
  short       ctFlags;
  short       ctSize;
  CSpecArray  ctTable;
}
ColorTable, *CTabPtr, **CTabHandle;
```

As suggested by these data structure definitions, a ColorTable is a dynamically sizable array and is typically managed through CTabHandles. The fields of a ColorTable are used in the following manner:

- *ctSeed* — This field holds a unique identifier, which is used by various Color Manager traps to establish a ColorTable's identity. QuickDraw considers ColorTables with matching *ctSeed* values to be identical, regardless of any other data values that they may contain. This value can also be used to identify a number of standard ColorTables defined by the System; these ColorTables are described in Chapter 4.
- *ctSize* — This field indicates how many entries are in a ColorTable, and effectively indicates the size in memory of remaining ColorTable data, since each ColorSpec entry is of a fixed length (8 bytes). The value of this field is one less than the actual number of entries.
- *ctFlags* — This field is treated as a set of bit flags and is used to determine the exact meaning of each color entry in the table. The usage of this field is discussed in Chapter 4.
- *ctTable* — The *ctTable* is the list of entries in the ColorTable. When a ColorTable is attached to a GDevice, the *value* field in each ColorSpec entry is reserved by the Color Manager to store the usages for a given entry. GDevices are discussed in detail later in this chapter.

A ColorTable entry's RGB value may be directly accessed by applications, *provided that the application created that ColorTable.* Some ColorTables are the "private property" of Toolbox managers and should only be accessed indirectly via specialized traps. The menubar's ColorTable is a good example of a private ColorTable. Specialized ColorTables usually place a special meaning for each entry. For example, a control's first ColorTable entry denotes the text color; the last entry denotes its frame color.

The ColorTable is an important data type. It's included by many other QuickDraw data structures, the most common of which is the description of the colors to be used for indexed pixel values. ColorTables are also used by other Toolbox managers to hold lists of colors used for specialized purposes, many of which are discussed in Chapter 4.

▶ Graphic Devices

Apple introduced the concept of a *graphics device* (or *GDevice*) in order to accommodate different graphics display architectures with a minimum of software specialization. A graphics device is a pseudo-device; that is, it is a software representation of a hardware graphics device.

Important ▶

The intent of Graphics Devices is often misunderstood and has been the cause of much confusion and wasted efforts among programmers. There are two factors that contribute heavily to this confusion:

- **High level of abstraction** — The GDevice is a high-level, abstract concept. It's intended to embody and define a large number of important graphical concepts: pixel memory organization, color mapping, device-driver–based hardware control, and so forth.
- **Lack of concise documentation** — The Graphics Devices chapter in *Inside Macintosh*, Volume V, is uncharacteristically vague and does little to convey the theoretical understanding or implementation details of GDevices. In particular, the relationship between the Graphics Device and Color Managers isn't well documented. The second edition of *Designing Card and Device Drivers for the Macintosh Family* has a number of useful sections on GDevice programming, but this text is not well known to most programmers.

Clarifying this confusion won't be easy, so try to be as thorough and patient as you can while reading this section.

From a technical perspective, GDevices perform the following services:

- **Define basic pixel attributes** — The GDevice determines which color model (indexed or direct) is used. It further determines the number of bits per pixel, the horizontal and vertical resolution, and colors currently available to the application. If a GDevice belongs to a display monitor or hardcopy graphics device, it contains a reference to the physical pixel image memory.
- **Facilitate color mapping** — A GDevice defines which colors are available for display and imaging purposes. It's possible that graphics data structures may contain color values that cannot be represented by a GDevice; in such cases, that GDevice contains

information used to determine how those values are translated into ones that are valid within that GDevice. A most obvious example of color mapping is the display of color pictures on a grayscale monitor. The actual determinations of color mapping are made by the Color and Palette managers.

- **Determine software control access** — A GDevice that is associated with a physical display or hardcopy hardware device can accommodate direct hardware access through an associated low-level device driver and provides QuickDraw drawing traps with the necessary addressing information to access VRAM (video RAM) directly. This allows graphics hardware manufacturers to implement the specific low-level video control functionality while allowing QuickDraw to function independently of graphics display hardware architecture.

In short, GDevices play an important role in describing capabilities of the available imaging hardware to applications and other managers in the Graphics Toolbox.

▶ Using the Graphics Device Manager

A graphics device is defined by the GDevice data structure shown in Listing 3-4.

Listing 3-4. GDevice data structure

```
typedef struct
{
  short         gdRefNum;
  short         gdID;
  short         gdType;
  ITabHandle    gdITable;
  short         gdResPref;
  SProcHndl     gdSearchProc;
  CProcHndl     gdCompProc;
  short         gdFlags;
  PixMapHandle  gdPMap;
  long          gdRefCon;
  GDevice       **gdNextGD;
  Rect          gdRect;
  long          gdMode;
  short         gdCCBytes;
  short         gdCCDepth;
```

```
  Handle        gdCCXData;
  Handle        gdCCXMask;
  long          gdReserved;
}
GDevice, *GDPtr, **GDHandle;
```

Most of these fields are managed by various traps and shouldn't be accessed directly by application code. The following fields are of particular importance to applications and must be assigned directly on creation of a GDevice:

- *gdPMap* — This field holds the PixMap that defines many of the GDevice's attributes. The PixMap's ColorTable is the GDevice's ColorTable, defining all colors available on that GDevice. If the GDevice is associated with a hardware (physical) graphics device, then the PixMap will contain a reference to the hardware image memory. PixMaps and ColorTables are discussed in detail later in this chapter.
- *gdRect* — This field holds the bounding rectangle that defines the maximum extent of all graphics operations on that GDevice. It should be the same as the PixMap's bounding rectangle.
- *gdType* — This field indicates the color model defined by the GDevice: indexed, fixed, or direct. With respect to graphics devices, the Classic QuickDraw color model is a functional subset of the other three color models and isn't considered a unique *gdType* value. The PixMap stored in *gdPMap*, and the values within it, must agree with the value stored in *gdType*.

These are some of the more important GDevice concepts:

- **Handle-based access** — As stated earlier, this is true of many important QuickDraw data structures. Variables of type GDevice shouldn't be used.
- **Public GDevice** — A "public" GDevice is one that is used as a software interface to a hardware graphics device, typically a graphics display controller. Public GDevices are usually created by the Slot Manager at boot time, according to information contained in the controller's NuBus board's ROM or System folder startup files. Public GDevices contain references to a software device driver, which is referenced by "higher-level" QuickDraw traps. The associated PixMaps data corresponds to actual hardware values.

- **Private GDevice** — GDevices may also be created by applications to define a "private" color environment, independent of all public GDevices and associated hardware constraints. Private GDevices typically don't utilize device-based control.
- **Current GDevice** — The current GDevice is similar to the current GrafPort and is utilized by various QuickDraw traps through a system global variable. The current GDevice may be either public or private.
- **Main GDevice** — This is the public GDevice that represents the display monitor that has the menubar on it. The origin of the main GDevice is (0, 0), which corresponds to the pixel in the menubar's upper-left corner. There is only one main GDevice per machine.
- **GDevice list** — A list of public GDevices is maintained by the Graphics Device Manager, which is accessible to applications through a pair of traps. Private GDevices should never be placed in the GDevice list.
- **GDevice attributes** — A GDevice has a number of possible attributes that can be tested for, and, in some cases, selectively enabled or disabled. Such attributes include whether: it is active/inactive, is the Main GDevice, has color capabilities, and has an associated device driver. Most GDevice attributes are only relevant to public GDevices. Under certain conditions, an application should determine which attributes are available before performing a graphics operation.

Creation and Initialization

Developers should realize that the need to create and dispose of GDevices has been substantially reduced due to the advent of GWorlds, which manage GDevice-related details internally. GDevice memory management is described here for informational purposes only. You should feel strongly encouraged to create and use GWorlds rather than GDevices.

A GDevice is a dynamic data structure and must be properly allocated and initialized before it's used. The traps to perform these tasks are as follows:

```
GDHandle NewGDevice(
  short  refNum,
  long   mode );
```

```
void InitGDevice(
  short     refNum,
  long      mode,
  GDHandle  theGD );

void DisposGDevice(
  GDHandle  theGD );
```

The **NewGDevice()** trap returns a partially initialized GDevice, allocating but not initializing its PixMap and ITab (inverse table) fields. These allocations are made in the System heap, not the application heap. The other GDevice fields must be initialized by the application. Information on how to perform these tasks is covered later in this chapter. An example of this involved process is shown in Listing 3-5.

Listing 3-5. Creating a GDevice

```
GDHandle GDevice_Create(
  Boolean  isClutType )
{
GDHandle gdh;

  /* allocate a fresh new GDevice */
  gdh = NewGDevice( 0, (long)(-1) ); /* -1 specifies no driver */
  if (gdh != NULL)
  {
    (**gdh).gdID = (-1);      /* arbitrary */

    (**gdh).gdType = (isClutType ? clutType : directType);
    (**gdh).gdResPref = 5;    /* maximum resolution for */
                              /* inverse table */
    (**gdh).gdNextGD = NULL;  /* generally, private GDevs */
                              /* aren't linked */

    /* a "standard" screen size */
    SetRect( &((**gdh).gdRect), 0, 0, 640, 480 );

    /* this is a color-capable gdevice */
    /* which doesn't have an associated device driver */
    SetDeviceAttribute( gdh, gdDevType, TRUE );
    SetDeviceAttribute( gdh, noDriver, TRUE );

    /* since these are hardware characteristics, */
    /* they're not applicable */
    SetDeviceAttribute( gdh, mainScreen, FALSE );
```

```
    SetDeviceAttribute( gdh, screenDevice, FALSE );
    SetDeviceAttribute( gdh, screenActive, FALSE );

    SetDeviceAttribute( gdh, burstDevice, FALSE );
    SetDeviceAttribute( gdh, ramInit, FALSE );
    SetDeviceAttribute( gdh, allInit, FALSE );
  }

  /*
  ** Note: an inverse table and pixmap will need to be set up!
  */

  return gdh;
}
```

This routine performs most of the work needed to create a GDevice; still, it leaves a fair amount of work undone. In particular, the GDevice created by this routine still requires data structures to be installed in the GDevice's *gdPMap* and *gdITab* fields.

The **InitGDevice**() trap should be called only to initialize a public GDevice that has an associated device driver. It associates a GDevice record with its device driver and inserts it into the GDevice list. There are few, if any, circumstances under which applications should call this trap.

As with most dynamic data structures, a GDevice should be disposed of when an application no longer requires it. The **DisposGDevice**() trap deallocates the GDevice and its associated PixMap and ITable fields. The dynamically allocated fields *inside* of the GDevice's PixMap will need to be disposed of before calling this trap; this process is described later in this chapter.

Accessing the GDevice List

The device list contains references to all public GDevices known to the Graphics Devices Manager. The traversal of this list and subsequent read-only access of individual GDHandles can be accomplished though the following six traps:

```
GDHandle GetGDevice( void );

void SetGDevice(
  GDHandle theGD );
```

These two traps are used to obtain and define the current GDevice, which will consequently affect all color mapping and pixel data trans-

fer operations. They are to be used in a manner similar to **GetPort()** and **SetPort()**.

```
GDHandle GetDeviceList( void );

GDHandle GetNextDevice(
  GDHandle theGD );
```

These traps provide direct access to the GDevice list. An example of traversing this list is shown in Listing 3-6, which simply counts the number of GDevices in the list.

Listing 3-6. Example of accessing the device list

```
short CountGDevs( void )
{
GDHandle    curGD;
short       gdCounter;

  gdCounter = 0;
  curGD = GetDeviceList();
  while (curGD != NULL)
  {
    gdCounter++;
    curGD = GetNextDevice( curGD );
  }

  return gdCounter;
}
```

These two traps

```
GDHandle GetMainDevice( void );

GDHandle GetMaxDevice(
  Rect   *globalBounds );
```

allow an application to obtain a GDHandle to a particular device, based on specific device attributes. **GetMainDevice()** returns the GDHandle of the main (menu bar) display GDevice. **GetMaxDevice()** returns a GDHandle to the GDevice of the greatest pixel depth that intersects the globalBounds argument in global coordinates. Many applications may find the need to obtain more detailed information about available public GDevices; in such cases, explicitly traversing the GDevice list is more appropriate than using either of these two traps.

Managing GDevice Attributes

The attributes of individual GDevices can be accessed via the following traps.

```
Boolean TestDeviceAttribute(
  GDHandle    theGD,
  short       attribute );

void SetDeviceAttribute(
  GDHandle    theGD,
  short       attribute,
  Boolean     value );
```

The GDevice attributes are maintained as a set of flags in a GDevice's *gdFlags* field. The **TestDeviceAttribute**() and **SetDeviceAttribute**() traps should always be used to access this field indirectly. Only one attribute may be set or tested for at a time.

Important ▶

The GDevice list is global to all applications. Setting GDevice attributes carelessly will almost certainly cause failure in some or all of the applications running at that time.

▶ Color Mapping

Color mapping is the term used to describe the process of substituting a color value for one that isn't available on the current indexed GDevice. On a direct GDevice, color mapping is academic, since all colors in the RGB colorspace are available, at least within a fixed tolerance.

The process of color mapping is undertaken jointly by the Color and Graphics Devices Managers and occurs under very specific circumstances. The circumstances may be related to copying data between structures that have ColorTables that differ from the current GDevice's ColorTable, or are related to the management of color requests associated with windows. The former case is discussed later in this chapter; the latter case is treated in detail in Chapter 4.

GDevice ColorTables and Color Mapping

ColorTables are of particular importance to indexed GDevices. An indexed display controller maintains a ColorTable to describe those colors currently available for display. When a graphics operation pro-

duces RGB values that aren't in the GDevice's ColorTable, color mapping must occur — otherwise the result cannot be assigned. The primary method of color mapping is the Color Manager trap **Color2Index**(), whose interface is shown here:

```
long Color2Index(
  RGBColor *anRGB );
```

On indexed GDevices, this trap returns the color entry index with the closest matching RGB value. On direct GDevices, this trap returns the appropriate 16- or 32-bit packed RGB value. This value is called a *pixel index value,* which is how a pixel value is represented within a pixel image variable.

The primary sources of "out-of-bounds" RGB values are pen color assignments and pixel image transfers. QuickDraw compares the *ctSeed* values of the source PixMap and the current GDevice's PixMap when copying data between PixMaps. If the two *seed* values are the same, the two ColorTables are assumed to be the same and no color mapping is performed; pixel values are just transferred from one PixMap to the other.

Inverse Tables

Going from a pixel index value to its corresponding RGB color value involves a simple table lookup. Going from a color value to an indexed value requires a special data structure known as an *inverse table.* Inverse tables are generally "owned" by GDevices and referenced by various Color Manager traps. Listing 3-7 shows the InverseTable data structure.

Listing 3-7. InverseTable data structure

```
typedef struct
{
  long      iTabSeed;
  short     iTabRes;
  char      iTTable[1];
}
ITab, *ITabPtr, **ITabHandle;
```

The *iTabSeed* field operates in conjunction with the similarly named *ctSeed* field in a ColorTable: It contains a unique identifying value, which is used by the Color Manager to indicate when an InverseTable

is out of synch with the GDevice's ColorTable. When an entry in the GDevice's ColorTable is changed, the ColorTable's *ctSeed* value is incremented. Any Color Manager trap that subsequently references that InverseTable will compare its seed value with the ColorTable's seed value; if the two seeds differ, the InverseTable is rebuilt immediately.

Inverse Table Resolution and Hidden Colors. Functionally, inverse table access is the reverse process of ColorTable access: Use an RGBColor value to point into a table of pixel index values and retrieve the associated index value. Unfortunately, inverse table lookup isn't quite as simple as it may seem. The inverse table takes an RGB value as an index and produces an index value as output. Now stop and consider the memory implications of this simple statement. To build a table with one entry for each possible unique RGB value would require a table with $(2^{16})^3$, or 2^{48} entries! The Color Manager shrinks the inverse table to a manageable size by reducing the precision of the RGB components. The *iTabRes* value defines the inverse table's resolution, that is, how many bits are used from each RGB component to build inverse table index values. The maximum resolution allowed is five. When an RGB triplet is looked up in an inverse table, the high-order bits of the three RGB components are combined into a 16-bit index value, as shown in Figure 3-10.

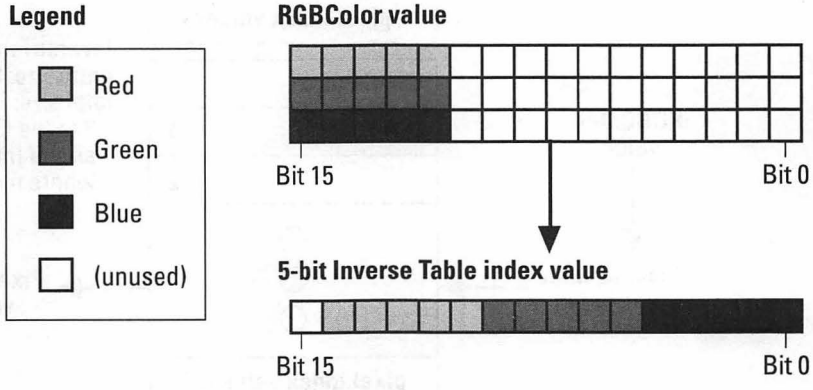


Figure 3-10. Building an inverse table index

This value is used to index into inverse tables in much the same way an indexed pixel value is used to point into a ColorTable, as shown in Figure 3-11.

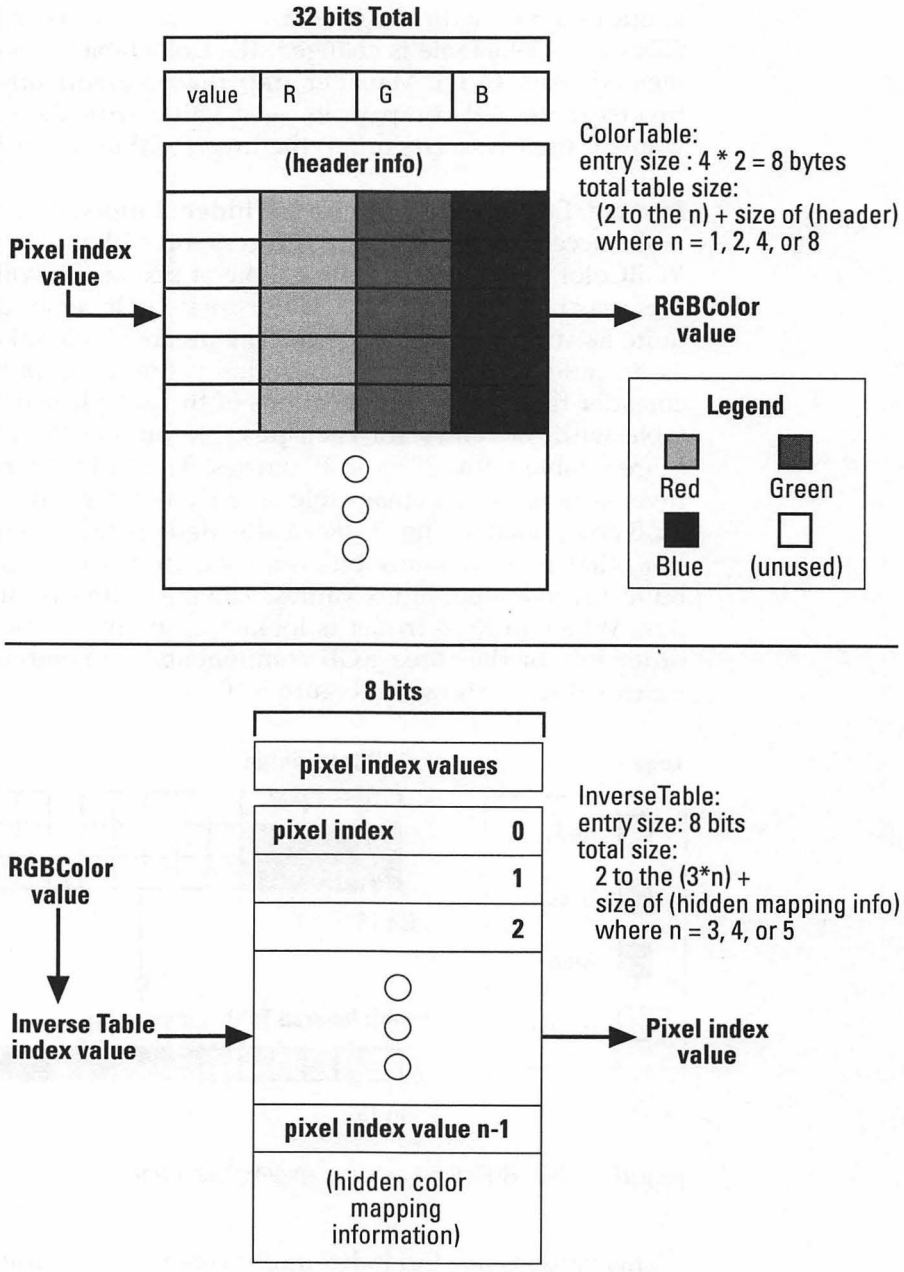


Figure 3-11. Inverse table indexing

Let's use 5-bit resolution as an example. By using a 15-bit index, the associated table has only $2^{15} = 32768$ entries — not zillions. But there is a problem with this reduced resolution: RGB values that differ in the lower-order bits are mapped into the same inverse table index. Consider two RGB triplets that reside in the current device's ColorTable and have the values shown in Table 3-1.

Table 3-1. Calculating inverse table index values

	Decimal	*Hex*	*Binary*
r1	2048	0800	0000 0100 0000 0000
g1	7168	1C00	0001 1100 0000 0000
b1	7168	1C00	0001 1100 0000 0000
r2	0	0000	0000 0000 0000 0000
g2	7168	1C00	0001 1100 0000 0000
b2	7168	1C00	0001 1100 0000 0000

Using the high-order 5 bits from each component to make a 15-bit inverse table index yields the same result in both cases, as shown in Table 3-2.

Table 3-2. Results of calculating inverse table indexes

	Decimal	*Binary*
iTabIndex1	99	00000/00011/00011
iTabIndex2	99	00000/00011/00011

You can see that although the two RGB triplets are different, the difference occurs in the sixth bit of precision. Go to the Color Picker dialog and check the RGB values listed in Table 3-1. You'll realize that the difference between the two colors specified is invisibly tiny — they differ in the sixth bit of one of three 8-bit numbers. Color values that differ from each other below the resolution are referred to as *hidden colors*. The lower the inverse table's resolution, the greater the potential number of hidden colors.

The Color Manager has a few tricks up its sleeve to deal with these indexing collisions; it appends to the inverse table information that allows it to break ties when two or more colors map into the same inverse table index. This information is used by **Color2Index**(), which

will return the "best" match. Keep this in mind: A GDevice's inverse table is built by the Color Manager from that GDevice's ColorTable, so the Color Manager knows which colors will cause inverse table lookup collisions before any color mapping is performed.

Note ▶

The arithmetic transfer modes are described in *Inside Macintosh*, Volume V, as not using the inverse table for colorspace calculations. This means that the resolution of hidden colors via the inverse table's "additional information" isn't performed, causing "ties" between hidden colors to be overlooked.

Using Inverse Tables. The inverse table doesn't really require much explicit management effort. *The only time that an application needs explicitly be involved with inverse tables is during the process of creating an indexed GDevice.* This task is performed with the Color Manager trap **MakeITable**(), which has the following interface:

```
OSErr MakeITable(
  CTabHandle    theCTable,
  ITabHandle    theITab,
  short         resolution );
```

The "resolution" can be 3, 4, 5, or 0; a zero tells this trap to use the value of the GDevice's *gdResPref* field. If either the CTable or ITable arguments are NULL, then the appropriate argument is the one associated with the current GDevice. As a consequence, inverse table installation is simple; just create one after creating the GDevice to which it belongs, as shown in Listing 3-8.

Listing 3-8. Example of installing an inverse table

```
void CreatingIndexedGDevice( void )
{
GDHandle    someGD, savedGD;
OSErr       errStat;

  /* allocate and initialize a GDevice */
  someGD = NewGDevice();

  /* initialize rest of the GDevice's fields */
  (**someGD).gdResPref = 5;
```

```
    /* initialize rest of the GDevice PixMap's fields */
    InitPixMap( (**someGD).gdPMap, 8 );

    /* create and install inverse table */
    savedGD = GetGDevice();
    SetGDevice( someGD );
    errStat = MakeITable( NULL, NULL, 0 );
    if (errStat != noErr)
      ;


    SetGDevice( savedGD );
}
```

There are circumstances where it may be desirable to know beforehand how specific colors will get matched without performing a drawing operation. The Color Manager provides such a tool in the form of the **GetSubTable**() trap, which has the following interface:

```
void GetSubTable(
  CTabHandle    inputCTab,
  ITabHandle    invTable,
  CTabHandle    outputCTab );
```

This trap runs "inputCTab" through "invTable" to produce "outputCTab." If required, "outputCTab" is resized as necessary. If "invTable" is NULL, then the one on the current GDevice is used instead. Both CTabHandle arguments must refer to valid, allocated CTables. The resulting ColorTable can then be analyzed for closeness of color mapping.

Search and Complement Procedures

Another degree of color control offered by QuickDraw comes through the use of search procedures. A search procedure is a routine for determining how a color should be represented on a given GDevice. Recall that destination color information comes from the current GDevice. For indexed devices, QuickDraw uses an inverse table to determine with which index to represent a given color. For direct devices, QuickDraw merely truncates each component of the RGBColor to the desired size: 5 bits for 16-bit color, 8 bits for 32-bit color.

By installing a search procedure on a GDevice, an application can control the way colors are represented on that GDevice. If a search

procedure is installed on a GDevice, QuickDraw calls the search procedure every time a color is needed to draw on that device. This allows full control over the way colors are mapped within the application.

Important ▶

If an application attaches a search procedure to a public GDevice, it must make sure it's removed immediately after performing the desired drawing operation, as it will affect drawing in all other applications. A search procedure should never be left installed on a public GDevice across a call to **WaitNextEvent**().

QuickDraw calls the current GDevice's installed search procedures whenever it needs to determine how a color should be represented. The only exception is for the arithmetic transfer modes in **CopyBits**(). If an application needs the result of an arithmetic transfer using **CopyBits**() to be processed by a search procedure, it must perform the operation first and then use **CopyBits**() with the "srcCopy" mode to copy the result into the desired destination pixel image.

How SearchProcs Influence Drawing. QuickDraw uses the following Pascal interface to call a search procedure:

```
pascal Boolean SearchProc(
  RGBColor *theColor,
  long     *position );
```

Note ▶

Search procedures are really functions since they return a result. They are often called *searchProcs*, so the term *search procedure* stuck even though it isn't technically accurate. It just sounds weird to say *searchFunc*.

The RGBColor argument passed into the searchProc is one that can be modified and returned to the calling trap, and it describes the color that the Color QuickDraw trap is attempting to map into the current GDevice. If the search procedure returns a result of TRUE, the result argument is used for the destination color. The QuickDraw trap attempting to map an RGBColor will call each installed search procedure on the current GDevice until one of them returns a TRUE match value.

If the destination is an indexed GDevice, the result is expected to be a pixel index value. If the destination is a direct GDevice, the result is expected to be a direct color at the GDevice's pixel depth.

If the search procedure returns a result of FALSE, QuickDraw passes the returned RGBColor to the next search procedure (if another exists on this device), or performs the default color mapping using an inverse table for indexed devices or truncation for direct devices. Search procedures are installed on the current device using the **AddSearch**() trap, which has the following interface:

```
void AddSearch(
  ProcPtr  searchProcAddr );
```

The search procedure will be called before any subsequent drawing occurs to that GDevice. To remove an installed search procedure from the current GDevice, use the **DelSearch**() trap, which has the following interface:

```
void DelSearch(
  ProcPtr  searchProcAddr );
```

Some SearchProc Techniques. Search procedures can be used to perform useful image processing operations. *Inside Macintosh,* Volume V, omits the fact that the RGB color passed to a search procedure is a "Var" argument; that is, one that can be modified and returned to the calling routine or trap. This property of the search procedure is very useful, as you can simply modify the RGBColor argument and return FALSE. Certain useful image processing operations, such as darkening or lightening an image, can be performed in this manner. Listing 3-9 shows an example of a search procedure which darkens all drawing that occurs on the GDevice where the search procedure is installed.

Listing 3-9. Example of a search procedure

```
pascal Boolean Darken(
  RGBColor *targetRGB,
  long     *result )
{
  targetRGB->red >>= 1;
  targetRGB->green >>= 1;
  targetRGB->blue >>= 1;

  return FALSE;
}
```

This search procedure darkens each color component by a factor of 2. Notice that if you wanted to darken drawing by a factor of 4, you could install this same procedure to the GDevice twice. A search procedure that lightens drawing would be similar.

▶ Color GrafPorts

In order to make Classic QuickDraw color-capable, a new version GrafPort structure had to be designed. The CGrafPort, or Color GrafPort, provides the informational capacity required by the new color traps while maintaining compatibility backward (old code runs on new machines) and forward (new code runs on old machines — if written well). These aims were realized by observing the following design rules:

- **Color QuickDraw must be "transparently aware" of GrafPorts** — The Color QuickDraw versions of the Classic QuickDraw recognize "black-and-white only" applications; they execute GrafPort-based code in a manner that is indistinguishable to both the user and the application. Simply put, black-and-white applications are still black and white to the user, and, presuming that the application in question meets Apple's published compatibility guidelines, it won't crash.
- **The expansion of the GrafPort to the CGrafPort must maintain the field types and ordering** — The exceptions occur in fields that were previously considered "private" to QuickDraw traps, that is, they were not to be accessed directly by application code.

For the sake of comparison, the GrafPort and CGrafPort data structures are shown in Listing 3-10.

Listing 3-10. GrafPort and CGrafPort data structures

```
typedef struct
{
   short       device;
   BitMap      portBits;
   Rect        portRect;
   RgnHandle   visRgn;
   RgnHandle   clipRgn;
   Pattern     bkPat;
   Pattern     fillPat;
```

```
  Point       pnLoc;
  Point       pnSize;
  short       pnMode;
  Pattern     pnPat;
  short       pnVis;
  short       txFont;
  Style       txFace;
  short       txMode;
  short       txSize;
  long        spExtra;
  long        fgColor;
  long        bkColor;
  short       colrBit;
  short       patStretch;
  QDHandle    picSave;
  QDHandle    rgnSave;
  QDHandle    polySave;
  QDProcsPtr  grafProcs;
}
GrafPort, *GrafPtr;

typedef struct
{
  short         device;
  PixMapHandle  portPixMap;
  short         portVersion;
  Handle        grafVars;
  short         chExtra;
  short         pnLocHFrac;
  Rect          portRect;
  RgnHandle     visRgn;
  RgnHandle     clipRgn;
  PixPatHandle  bkPixPat;
  RGBColor      rgbFgColor;
  RGBColor      rgbBkColor;
  Point         pnLoc;
  Point         pnSize;
  short         pnMode;
  PixPatHandle  pnPixPat;
  PixPatHandle  fillPixPat;
  short         pnVis;
  short         txFont;
  Style         txFace;
  short         txMode;
  short         txSize;
```

```
  Fixed          spExtra;
  long           fgColor;
  long           bkColor;
  short          colrBit;
  short          patStretch;
  QDHandle       picSave;
  QDHandle       rgnSave;
  QDHandle       polySave;
  CQDProcsPtr    grafProcs;
}
CGrafPort, *CGrafPtr;
```

Although it may seem as though much has changed from the GrafPort to the CGrafPort, in actuality they are the same size. The reference to the GrafPort's embedded BitMap has been replaced by a Handle to a PixMap, a substantially extended version of a BitMap. The size difference between a BitMap and a PixMapHandle is 14 bytes; not coincidentally, the new CGrafPort fields are exactly 14 bytes long and follow the PixMapHandle field. The references to the various Patterns have been replaced with Handles to PixPats (color pixel patterns). These are described later in this chapter.

▶ Pen Colors

Of all functional enhancements afforded by the CGrafPort, true pen colors are probably the easiest to use, and they create the most dramatic impact. The Classic color model limited pen color choices to eight basic colors, including black and white, all of which were assigned using the **ForeColor()** and **BackColor()** traps. Even this limited repertoire was academic; unless you had an ImageWriter with a color ribbon and software that printed in multiple passes, you weren't likely to see any colors other than black and white.

Under the Classic color model, pen color usage is performed implicitly: Drawing takes place using the foreground color; erasing is simply drawing in the background color. The GrafPort's default foreground and background colors were black and white, and few applications had much need to change them.

Color QuickDraw provides two new traps, **RGBForeColor()** and **RGBBackColor()**, to assign a "true" pen color. Using them is surprisingly simple, as shown in Listings 3-11 and 3-12.

Listing 3-11. Simple drawing example

```
void DrawBoxes( void )
{
Rect boxRect;

   /* set old-style pattern*/
   PenPat( black );

   /* paint one big rectangle */
   SetRect( &boxRect, 0, 0, 100, 100 );
   OffsetRect( &boxRect, 20, 20 );
   PaintRect( &boxRect );

   /* erase four small rectangles */
   SetRect( &boxRect, 0, 0, 20, 20 );
   OffsetRect( &boxRect, 40, 40 );
   EraseRect( &boxRect );

   OffsetRect( &boxRect, 40, 0 );
   EraseRect( &boxRect );

   OffsetRect( &boxRect, -40, 40 );
   EraseRect( &boxRect );

   OffsetRect( &boxRect, 40, 0 );
   EraseRect( &boxRect );
}
```

The graphic result of calling DrawBoxes() is shown in Figure 3-12.

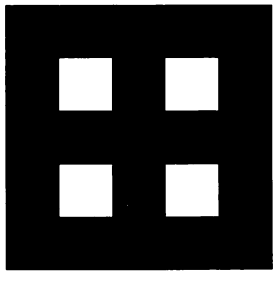


Figure 3-12. Results of DrawBoxes()

Listing 3-12 shows DrawColorBoxes(), a version of DrawBoxes() with the addition of RGB-based pen color control. The results would be red rectangles inside of a larger green rectangle.

Listing 3-12. Simple drawing example with pen color control

```
void DrawColorBoxes( void )
{
RGBColor curPenRGB;
Rect     boxRect;

  /* set the foreground pen color to brightest green */
  curPenRGB.red = 0x0000;
  curPenRGB.green = 0xFFFF;
  curPenRGB.blue = 0x0000;
  RGBForeColor( &curPenRGB );

  /* set the background pen color to brightest red */
  curPenRGB.red = 0xFFFF;
  curPenRGB.green = 0x0000;
  curPenRGB.blue = 0x0000;
  RGBBackColor( &curPenRGB );
  PenPat( black );

  /* paint one big rectangle */
  SetRect( &boxRect, 0, 0, 100, 100 );
  OffsetRect( &boxRect, 20, 20 );
  PaintRect( &boxRect );

  /* erase four small rectangles */
  SetRect( &boxRect, 0, 0, 20, 20 );
  OffsetRect( &boxRect, 40, 40 );
  EraseRect( &boxRect );

  OffsetRect( &boxRect, 40, 0 );
  EraseRect( &boxRect );

  OffsetRect( &boxRect, -40, 40 );
  EraseRect( &boxRect );

  OffsetRect( &boxRect, 40, 0 );
  EraseRect( &boxRect );
}
```

Note ▶

Applications that used pen colors other than black and white in GrafPorts will show drawing results in the selected colors if running on a Macintosh with a color monitor. It must have amused the developers of those applications to see color appear almost magically on color monitors after viewing their application in black and white on pre-Macintosh II machines.

▶ Transfer Modes

Color QuickDraw introduces a new family of transfer modes, known as *arithmetic transfer modes*. These modes calculate resultant pixel values by adding, subtracting, or averaging the RGB components of the source and destination pixel values.

The logical transfer modes perform bitwise logical operations that are a function of source pixel value, the current port's foreground and background pen colors, and the corresponding destination pixel value — pixel by pixel — to compute a value for the resulting pixel that then replaces the destination pixel in the destination PixMap.

Arithmetic Modes

Some of these modes factor in a third constant RGB operand, which is stored in the CGrafPort. The constants are listed in the following sections.

Operand Color. This value is used either as a weighting or threshold value. As a weighting value, it prescribes to the amount of source and destination values in the resultant RGB value. As a threshold, it assigns the maximum or minimum value of additive and subtractive operations. The default operand color is black and can be reset on a CGrafPort basis using the **OpColor**() trap.

Highlighting Color. In the black-and-white world, highlighting a selection is simple. Inverting monochrome pixel values yields acceptable visual results: White pixels turn black and vice versa. In a color environment, simple complementation doesn't always produce an obvious visible selection, because the inversion of multibit pixel values generally yields many different colors, which is unsuitable for highlighting text and other graphic objects.

To remedy this shortcoming, Color QuickDraw allows for the specification of a highlight value that will be used to replace all selected pixels of the CGrafPort's current background color. The default highlight value is set by the user via the Control Panel and can be reset on a CGrafPort basis using the **HiliteColor**() trap.

There are ten arithmetic modes. Their effects are described as follows.

addPin / addOver. These modes add the source and destination RGB values. AddOver will "wraparound" addition overflow, whereas addPin "pins" addition to the current value of the operand color, ignoring overflow. An example illustrating the effects of these modes is shown in Table 3-3, using an operand color of RGB = (65535, 65535, 65535).

Table 3-3. Example of RGB color addition

	Source		*Destination*		*Result*
addPin:	32767	+	40959	=	65535
addOver:	32767	+	40959	=	12286

You'll probably find addPin more appropriate than addOver.

subPin / subOver. These modes subtract the source RGB values from those of the destination. Similar to their additive counterparts, subOver will "wraparound" subtraction underflow, whereas subPin pins results to the current value of the operand color. An example illustrating the effects of these modes is shown in Table 3-4, using an operand color of RGB = (0, 0, 0).

Table 3-4. Example of RGB color subtraction

	Source		*Destination*		*Result*
subPin:	32767	–	40959	=	0
subOver:	32767	–	40959	=	57344

adMax / adMin. These modes calculate the result by taking the maximum or minimum values for each source and destination RGB component. An example illustrating the effects of these modes is shown in Table 3-5.

Table 3-5. Example of RGB color maximum and minimum thresholding

adMax:	*Source*	*Destination*		*Result*
Red:	20480	40959	=	40959
Green:	36863	12287	=	36863
Blue:	0	52992	=	52992
adMin:	*Source*	*Destination*		*Result*
Red:	20480	40959	=	20480
Green:	36863	12287	=	12287
Blue:	0	52992	=	0

These modes are especially useful for image thresholding operations. For example, an application could display only the brightest colors of an image by following these steps:

1. Render the image offscreen.
2. Set the appropriate RGB threshold value as the foreground color.
3. Call **PenMode**(), passing it the "adMax" constant.
4. Call **FillRect**(), passing it the image's boundary Rect.
5. Copy the offscreen image into a window.

By the Way ▶

The name "adMax" stands for "arithmetic drawing maximum," and "adMin" is short for "arithmetic drawing minimum." The "ad-" prefix shouldn't be construed to mean "additive." At least one compiler company has made that mistake in their header file, misnaming "adMax" and "adMin" as "addMax" and "addMin," respectively.

blend. This mode "blends" source and destination RGB values, using the operand color as a weighting factor for averaging. The result is determined by the following formula as applied to each RGB component value:

$$\text{result} = \frac{\text{source} \times \text{operand}}{65535} + \frac{(65535 - \text{destination}) \times \text{operand}}{65535}$$

Pretty messy. Perhaps the example in Table 3-6 will help. It uses an operand color of RGB = (32767, 32767, 32767).

Table 3-6. Example of RGBColor blending

Blend	*Source*	*Destination*		*Result*
Red	20480	8160	=	14320
Green	36863	12287	=	24575
Blue	0	52736	=	26368

In this example, the value of 32767 ($7FFF hexadecimal) was chosen for all three operand color components to achieve an equally weighted 50-50 average of source and destination values. Although the mathematics may seem involved, the results of a blend operation should be fairly intuitive: Blending a bright red source with a bright blue destination will produce a purple that will be slightly darker than either source or destination.

hilite. This is basically a color extension of the Invert GrafVerb. The hilite transfer mode provides an aesthetic solution to the problem of highlighting by inversion: All pixel values of the current background color are replaced with the value of the highlighting color, and vice versa.

transparent. Not really a transfer mode itself, transparent is used as a modifier in conjunction with the Classic bitwise transfer modes srcCopy and patCopy. Transparency causes pixels in the destination to be replaced by pixels in the appropriate source pixel, provided that the source pixel value isn't equal to the current background color. This makes pattern-based fills "transparent" at those points in the pattern that equal the background color value.

grayishTextOr. With the introduction of System 7.0 comes a useful feature for rendering "grayed out" text to indicate disabled items. The feature is implemented in the form of a new text transfer mode: GrayishTextOr mode (31), a text mode that draws with the average of the foreground and background colors in the current port.

Traditionally, grayed text was rendered in the manner described in Listing 3-13.

Listing 3-13. "Old-style" way to draw grayed-out text

```
void DrawGrayString(
  char  *aPascalStringPtr )
{
PenState    savedPenState;
FontInfo    curFontInfo;
Rect        textBoundsR;
Point       penPosition;
short       textHorizExtent, savedTextMode;

  if ((aPascalStringPtr == NULL) || (aPascalStringPtr[0] == '\0'))
    RETURN;

  /* determine the bounding rectangle where the text */
  /* is to be rendered */
  GetFontInfo( &curFontInfo );
  penPosition = thePort->pnLoc;
  textHorizExtent = StringWidth( aPascalStringPtr );
  SetRect(
    &textBoundsR, 0, 0, textHorizExtent,
    curFontInfo.ascent + curFontInfo.descent );
  OffsetRect(
    &textBoundsR, penPosition.h,
    penPosition.v - curFontInfo.descent );

  /* be nice and save state */
  GetPenState( &savedPenState );
  savedTextMode = thePort->txMode;

  /* draw the text */
  TextMode( srcCopy );
  DrawString( aPascalStringPtr );

  /* "punch out" the text in the black portion */
  /* of the gray pattern */
  PenPat( gray );
  PenMode( patBic );
  FillRect( &textBoundsR );

  /* be even nicer and restore state */
  TextMode( savedTextMode );
  SetPenState( &savedPenState );
}
```

On 1-bit image, grayishTextOr draws text with the familiar 50 percent gray pattern.

This text transfer mode is meant to be used for user interface purposes only. Most of the grayed text used in Finder 7.0 is drawn using this mode. Currently, the information associated with this mode isn't stored in pictures and is undefined for printing. This may change in future versions of QuickDraw.

If an application draws text using grayishTextOr, the **StdText**() bottleneck procedure will receive a text mode of grayishTextOr. The System 7.0 sample WDEF from Apple uses grayishTextOr, and there is a Gestalt flag that returns whether or not the mode is available.

These arithmetic modes facilitate a number of powerful image processing operations that would be difficult or impossible with the original transfer modes. Listing 3-14 shows a simple example of tinting an image using the blend mode.

Listing 3-14. Use of OpColor() with the blend transfer mode

```
void AdjustTint(
  RgnHandle   interestRgn,
  SmallFract  tintFactor,
  Boolean     makeLighter )
{
PortColorState  savedColorInfo;
PenState        savedPenState;
RGBColor        newRGBColor, newOpColor;
short           arithPenMode;

  if (EmptyRgn( interestRgn ))
    RETURN;

  GetPenState( &savedPenState );
  CPort_GetColorState( &savedColorInfo );

  /* determine a good "tintFactor" value: 0x3FFF will tint 25% */
  newOpColor.red = newOpColor.green = newOpColor.blue =
    tintFactor;
  arithPenMode = blend;

  if (makeLighter)
    newRGBColor.red = newRGBColor.green = newRGBColor.blue =
      0xFFFF;
  else
    newRGBColor.red = newRGBColor.green = newRGBColor.blue =
      0x0000;
```

```
    OpColor( &newOpColor );
    RGBForeColor( &newRGBColor );
    PenMode( arithPenMode );
    PenPat( black );

    PaintRgn( interestRgn );

    CPort_SetColorState( &savedColorInfo );
    SetPenState( &savedPenState );
}
```

Logical Transfer Modes

The bitwise logic that is applied to the source and destination pixels works most understandably when the pixels are 1 bit deep (BitMaps), as in the days when Macintoshes had only black-and-white screens. When performed on multi-bit pixel images, the results of these bitwise operators may seem, at first blush, counterintuitive. The logical transfer modes still perform the same operations that were done on black-and-white pixels, but the results can be unexpected, particularly with indexed PixMaps. When bitwise operations are performed on indexed pixel values, the calculations are performed on the indexes themselves, not on the RGB color values they represent. Such an operation is said to take place in index space, as opposed to colorspace. Consider the partial ColorTable shown in Table 3-7.

Table 3-7. Partial ColorTable example

Entry	*R*	*G*	*B*
0	$0000	$0000	$0000
1	$FFFF	$0000	$FFFF
2	$0000	$FFFF	$0000
3	$90F0	$5AA5	$8888

Now consider an operation using srcOr mode, where source pixels of value 1 are combined with destination pixel values of 2.

In RGB color space:

CTab.rgb[1] OR CTab.rgb[2]
= ($FFFF, $0000, $FFFF) OR ($0000, $FFFF, $0000)
= ($FFFF, $FFFF, $FFFF)

In index space:

```
    1 OR 2
=   (0000 0001)  OR  (0000 0010)
=   0000 0011
=   3
```

Is the difference clear now? The important idea here is that arithmetic transfer modes operate in colorspace, and logical transfer modes operate in index space. So choose your transfer modes wisely.

Index space operations on indexed GDevices proceed according to the diagram in Figure 3-13.

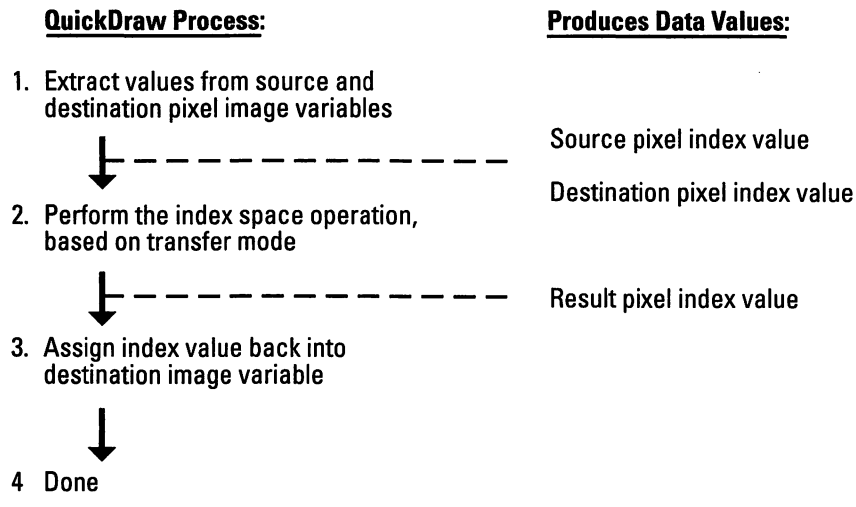


Figure 3-13. Processing index operations

Colorspace operations are shown in Figure 3-14.

The arithmetic modes are colorspace operators. Arithmetic operations usually take more time to execute than bitwise operations because arithmetic operations are more computationally intensive and because RGBColor values require more memory to store than index values.

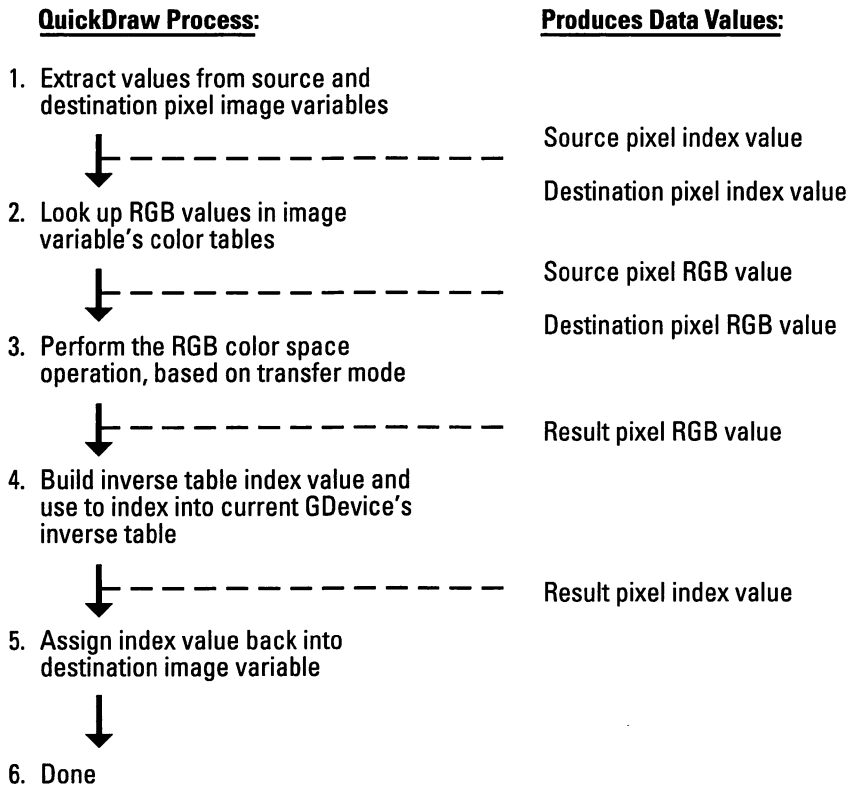


Figure 3-14. Processing index space to RGB operations

▶ Digital Images

Digital pictures can be treated as a two-dimensional array of pixel values. Each pixel has its own value as it's stored in digital memory. When a pixel value is written to a display GDevice, its digital value is converted to a voltage that finds its ultimate expression as an amount of emitted light on a raster-scan display.

Pixels in computer memory have a few nice properties by virtue of their digital manifestation. Since digital images are composed of digital values (numbers), they can be statistically manipulated, almost as if they were a spreadsheet. Pixels in one image can be subtracted from corresponding pixels in another image to produce a *difference image*.

Difference images reveal subtle changes between similar images, such as a moving planet in a fixed field of stars. A pixel value in an image can also be numerically compared to its neighbors and then given a new value based on the result of the comparison; such an operation is a type of filter function, a filter being a process that takes an image as input and produces another image as output. Digital images can be used to express concepts and phenomena visually that aren't by their nature visual, such as economic data or consumer spending trends. In other cases, digital images may be obtained from sources that don't use visual imaging techniques. Many of the different medical imaging technologies such as magnetic resonance imaging (MRI) and ultrasound can "photograph" internal organs. These images can be assigned color values to emphasize changes between the data points within the image; when they are displayed with these colors, they are called *false-color images* because the image colors aren't related to the subject's "inherent" color. An indexed pixel image's value-to-color mapping is defined by the order of ColorSpec entry values within its ColorTable: It may be either determined mathematically (as with the rainbow ColorTable in Listing 3-2) or contain arbitrarily ordered RGB values. To change the false-color mapping, an application can change color values within the image's ColorTable and redraw it to the screen or other desired destination GDevice.

▶ PixMaps

The data structure QuickDraw uses to describe pixel images is the **PixMap**, which replaces the BitMap discussed in the previous chapter. Listing 3-15 shows the PixMap data structure.

Listing 3-15. PixMap data structure

```
typedef struct
{
  Ptr         baseAddr;
  short       rowBytes;
  Rect        bounds;
  short       pmVersion;
  short       packType;
  long        packSize;
  Fixed       hRes;
  Fixed       vRes;
  short       pixelType;
  short       pixelSize;
```

```
    short       cmpCount;
    short       cmpSize;
    long        planeBytes;
    CTabHandle  pmTable;
    long        pmReserved;
}
PixMap, *PixMapPtr, **PixMapHandle;
```

The PixMap is a workhorse of a data structure that is used wherever a description of related pixels is needed. Every color window contains a CGrafPort, which in turn contains a PixMap that defines the pixel values within the window. Color versions of familiar user interface data structures, such as icons, cursors, and patterns, all contain PixMaps. Image display and output devices are partially described by the PixMap contained in the *gdPMap* field of its public GDevice. These PixMap's *baseAddr* field points to the memory whose values represent the pixels currently displayed on the associated display monitor.

▶ PixMap Data Structure Description

Here's how the fields in a PixMap describe a pixel image:

- *baseAddr* — This field serves the same purpose as it did in a BitMap; it contains a pointer to a block of memory where the pixel values are stored, which we refer to as the *pixel image buffer* or simply *image buffer*. Starting at this address, pixel values are stored sequentially in image row order, beginning with the image's upper-left pixel. The value of this field can be directly assigned at PixMap initialization; an existing PixMap's *baseAddr* value should be modified only with extreme caution. If Color QuickDraw version 1.0 is available, then reading this value should always be performed indirectly via the **GetPixBaseAddr**() trap. The reasons for this are given in the "Addressing Modes" section of Chapter 9.
- *bounds* — This field describes the PixMap's coordinate system and displayable pixel values: The top and left values are often zero, but not necessarily so.
- *rowBytes* — This field holds the number of bytes needed to store a single row of pixel data, rounded up to the nearest even number. For maximum execution speed, the value of *rowBytes* should be a multiple of 4, so that each row of pixel values begins on a long word boundary. The two highest order bits are reserved as flag

bits, which must always be set to one. These flags are used by various QuickDraw traps to distinguish between a GrafPort's BitMap and a CGrafPort's PixMap. The allowable values for *rowBytes* are $0 to $3FFE hexadecimal (16,382 decimal), excluding the flag bits. The usage of flag bits is as follows:

00xxxxxx	BitMap
01xxxxxx	(illegal)
10xxxxxx	PixMap
11xxxxxx	CGrafPort

The value of *rowBytes* may be larger than the display area indicated by the bounds rectangle. This means that the first pixel of a row doesn't always follow the last pixel on the previous row. Therefore, the address of a chunky pixel at (i, j) should be calculated according to the following formula:

```
pixel[i, j] =
   baseAddr +
   (j - bounds.top) * (rowBytes & 0x3FFF) +
   ((i - bounds.left) * pixelSize) / 8;
```

- *hRes* and *vRes* — These fields define the resolution of the PixMap in pixels per inch, expressed as a Fixed (fixed-point) value. Typically, the value of these fields is 72.0 ($4800000 hexadecimal), but can be changed by the application as needed.

Note ▶

These fields are initialized by the **NewPixMap**() trap. Before System 7.0, these fields were assigned the value of the current GDevice PixMap's *hRes* and *vRes* fields. Now these fields are initialized to $48000000, regardless of the current GDevice resolution.

- *pixelSize* — This field holds the number of bits per pixel value. The allowable pixel sizes are any power of two between 1 and 32, inclusive. Byte-sized (8-bit) pixels are by far the most common. Within each row of pixels there is no byte- or word-alignment of pixels.
- *cmpCount* and *cmpSize* — The *cmpCount* field indicates the number of components that make up each pixel value. Indexed pixels have a *cmpCount* of 1 because each pixel value is a single indivis-

ible index into the PixMap's ColorTable. Direct pixel values are actually the combination of distinct values for R, G, and B; consequently direct pixels typically have *cmpCount* values of 3.

The *cmpSize* field indicates the actual size of each pixel value. As noted in *Inside Macintosh*, Volume V, the value of *pixelSize* doesn't necessarily equal *cmpCount* × *cmpSize*. For direct pixels, pixelSize never equals *cmpCount* × *cmpSize*: 32-bit pixels contain only 24 bits of color information, and 16-bit pixels contain only 15 color information bits.

- *pixelType* — This field indicates that the *baseAddr* field points to indexed (0) or direct (16) pixel values.
- *packType* — This field indicates which of the three indexed pixel memory organizations is being used: chunky (0), planar/chunky, or planar. As of QuickDraw version 1.2, indexed PixMaps are always chunky.

```
Chunky:
   RGB pix0   RGB pix1   ...

Planar/Chunky:
   R pix0  R pix1  ...  G pix0  G pix1  ...  B pix0  B pix1  ...

Planar:
   R pix0/bit0     R pix1/bit0 ...
   R pix0/bit1     R pix1/bit1 ...
      ...
   G pix0/bit0     G pix1/bit0 ...
   G pix0/bit1     G pix1/bit1 ...
      ...
   B pix0/bit0     B pix1/bit0 ...
   B pix0/bit1     B pix1/bit1 ...
      ...
```

Indexed PixMaps are always chunky; direct PixMaps can be either one of three types, except that, as of System 7 QuickDraw version 1.3, only chunky packing is supported for any type of PixMap.

- *planeBytes* — This field indicates how many bytes are required for each color component plane, from the beginning of one color plane to the beginning of the next. This assumes that the *packType* field indicates a planar packed PixMap — if we could use planar pixel types. For now, leave *planeBytes* set to 0.

- *pmVersion* — The field should be set to 0. If the PixMap's baseAddr memory is only addressable in 32-bit mode, then Bit 2 should be set. This feature and addressing modes are discussed in detail in Chapter 9.
- *pmReserved* — This field should be set to 0 to ensure that the application will be compatible with future versions of QuickDraw that might look at that location for more data. This 32-bit quantity is the hook on which Apple will hang future enhancements to the PixMap structure.
- *pmTable* — This field contains a handle to the PixMap's ColorTable. The ColorTable may be shared by other PixMaps, or it may be NULL, in which case the color values are assumed to be linearly scaled from the background color (low pixel values) to the foreground color (high pixel values).

  In Tech Note #272, Apple advises that the *pmTable* of a direct PixMap should hold a reference to a valid ColorTable. The ColorTable for a direct PixMap should have a *ctSeed* value equal to *cmpCount* × *cmpSize*. In the case of the preceding example, the *ctSeed* value would be 3 × 8, or 24. In the same ColorTable, the *ctFlags* and *ctSize* fields should be set to 0.

▶ PixMap Management Traps

Color QuickDraw provides a number of PixMap management traps:

```
PixMapHandle NewPixMap( void );
```

This trap simply returns a Handle to a new PixMap. The PixMap's *pmTable* field contains a valid Handle to an empty ColorTable. With the exception of *pmTable*, the fields of the new PixMap are copied from the current GDevice's PixMap. Applications need to set the PixMap's fields explicitly to avoid unforeseen problems. In particular, applications need to determine the PixMap's dimensions and coordinate system and assign them to the appropriate PixMap structure fields. If the PixMap is offscreen, then it requires its own private pixel image memory, which must be allocated and a pointer to it installed in the *baseAddr* field.

```
void DisposPixMap(
  PixMapHandle    thePMHdl );
```

This trap disposes everything that was created by **NewPixMap**(): the PixMap and the CTabHandle stored in *pmTable* field.

```
void CopyPixMap(
  PixMapHandle     srcPMHdl,
  PixMapHandle     destPMHdl );
```

This trap copies the field values of "srcPMHdl" into "destPMHdl", along with the associated ColorTable. Realize that this trap copies only the references to pixel memory (*baseAddr*), not the associated pixel image data! Both PixMapHandle arguments must be valid PixMaps, preferably created with **NewPixMap**().

As you can see, creating PixMaps and determining sensible values for the PixMap fields isn't a trivial task; two routines to help with these tasks are described in Listings 3-16 and 3-17.

▶ PixMap Management Strategies

Now that we understand basic PixMap management, let's look at the ways in which PixMaps are used within generic applications and extrapolate some rules and guidelines for effective PixMap utilization.

- **A single PixMap is a memory hog** — From the perspective of memory allocation, a PixMap looks something like Figure 3-15.

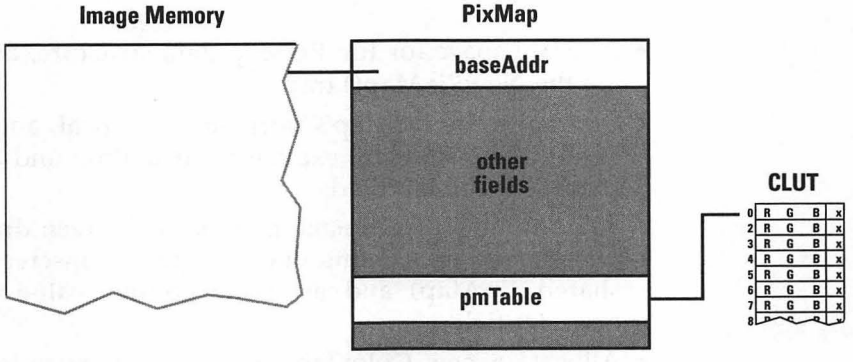


Figure 3-15. PixMap memory diagram

Deep pixel images require more memory than shallow ones. For two PixMaps of equal height and width, a 32-bit PixMap requires four times the memory of an 8-bit PixMap. In the case of a 640-pixel-by-480-row image, the difference is on the order of 900K of memory.

- **ColorTables and pixel memory buffers can be shared** — Sharing can save a lot of memory, especially when pixel image buffers are shared. The drawback of sharing is that an application must "remember" which PixMaps fields contain references to shared data structures when disposing of the entire PixMap. An 8-bit ColorTable requires 2056 bytes. For small images on the order of 50-by-50 pixels, an 8-bit ColorTable uses as much memory as the pixel values themselves. If you want to conserve memory and you're dealing with relatively small images, it might be worth the effort to have the images share the same ColorTable. But be careful when disposing of or modifying such shared data structures: Drawing with a modified ColorTable can produce unexpected results, and drawing with a disposed ColorTable usually causes swift and certain application death. Additionally, some direct PixMaps have a NULL *pmTable* fields, which should be checked accordingly before deallocating a PixMap's ColorTable.

▶ Creating Offscreen PixMaps

Creating a PixMap involves four distinct steps:

- Allocate space for the PixMap data structure, usually performed via the **NewPixMap**() trap.
- Configure the PixMap's horizontal, vertical, and depth (bits-per-pixel) dimensions by explicitly calculating and assigning the appropriate PixMap fields.
- Allocate new pixel image memory (offscreen drawing) or copy a reference to an existing one (usually an onscreen CGrafPort or a shared PixMap) and assign a pointer value to the PixMap's *baseAddr* field.
- Allocate a new ColorTable or copy reference to an existing one and install the reference in the PixMap's *pmTable* field.

These steps involve a small pile of grunt calculation. Creating an offscreen PixMap involves the allocation of an image buffer, and perhaps a private ColorTable as well. **NewPixMap**() fills in PixMap

field values with those from the current GDevice, where appropriate. Two of the routines from **PixMapsUtils** are presented in Listings 3-16 and 3-17 simply to demonstrate how to fill in the appropriate field values, allocate an image buffer, and duplicate a ColorTable. In particular, observe the following processes that go on within these routines:

- calculation of *rowBytes* from the bounds rectangle and pixel size,
- calculation of memory needed for pixel image buffer, and
- the relationship between pixel size and other field values.

Listing 3-16. PixMap_Create()

```
PixMapHandle PixMap_Create(
  Rect   *theBounds,
  short  targetDepth,
  OSErr  *errStat )
{
GDHandle      curGDev;
PixMapHandle  thePixels;
CTabHandle    theCTable;
Ptr           thePixelMem;
Rect          mapBounds, maxQDRect;
OSErr         localErrStat;
long          theRowBytes;
short         imageWide, imageTall;

  if ((theBounds == NULL) || (targetDepth <= 0))
    return NULL;

  if (errStat != NULL)
    *errStat = (OSErr)(-1);

  curGDev = GetGDevice();

  thePixels = NewPixMap();
  if (thePixels == NULL)
    return NULL;

  /* allocate pixel image memory */
  (**thePixels).bounds = *theBounds;
  (**thePixels).pixelSize = targetDepth;
  localErrStat = PixMap_AllocPixMem( thePixels );
  if (localErrStat != noErr)
  {
```

```
    PixMap_Dispose( thePixels );
    if (errStat != NULL)
      *errStat = localErrStat;
    return NULL;
  }

  /* get a brand new copy of the current GDevice's CTable */
  /* give it a new seed, a non-GD-type CTFlag, and re-index */
  if (targetDepth > 8)
  {
    theCTable = (CTabHandle)NewHandleClear( 8L );
    (**theCTable).ctSeed = targetDepth;
  }
  else
  {
    theCTable = ((**(**curGDev).gdPMap)).pmTable;
    localErrStat = HandToHand( &theCTable );
  }

  if (localErrStat == noErr)
  {
    /* perform fixup for non-GDevice-based ColorTables */
    (**theCTable).ctFlags = 0x0000;
    for (i=0; i<=(**theCTable).ctSize; i++)
      (**theCTable).ctTable[i].value = i;

    /* dispose CTable stub, left by NewPixMap(), */
    /* and install new CTable */
    DisposHandle( (**thePixels).pmTable );
    (**thePixels).pmTable = theCTable;
  }
  else
  {
    PixMap_Dispose( thePixels );
    *errStat = localErrStat;
    return NULL;
  }

  if (targetDepth <= 8)
  {
    (**thePixels).pixelType = 0;
    (**thePixels).cmpCount = 1;
    (**thePixels).cmpSize = targetDepth;
  }
  else
  {
```

```
    (**thePixels).pixelType = 16;
    (**thePixels).cmpCount = 3;
    (**thePixels).cmpSize = ((targetDepth == 32) ? 8 : 5);
  }

  (**thePixels).packType = 0;
  (**thePixels).packSize = 0;
  (**thePixels).planeBytes = 0L;

  /* at least until a new version comes out! */
  (**thePixels).pmVersion = 0;

  if (errStat != NULL)
    *errStat = localErrStat;

  return thePixels;
}
```

Listing 3-17. PixMap_AllocImageBuff()

```
OSErr PixMap_AllocPixMem(
  PixMapHandle  thePixels )
{
Rect   theBounds;
Ptr    thePixelMem;
OSErr  errStat;
long   imageWide, imageTall;
short  theRowBytes;

  if (thePixels == NULL)
    return (OSErr)(-1);

  theBounds = (**thePixels).bounds;
  imageWide = theBounds.right - theBounds.left;
  imageTall = theBounds.bottom - theBounds.top;

  /* Note: NewPtrClear() will take longer than NewPtr(), but */
  /* it initializes the pixel values nicely */
  theRowBytes = ((**thePixels).pixelSize *
    ((imageWide + 15) / 16)) * 2;
  thePixelMem = NewPtrClear( (Size)(imageTall *
    (long)theRowBytes) );
  errStat = MemError();
  if (errStat == noErr)
  {
```

```
    /* set bit for PixMaps */
    (**thePixels).rowBytes = theRowBytes | 0x8000;
    (**thePixels).baseAddr = thePixelMem;
  }
  else
  {
    (**thePixels).baseAddr = NULL;
    (**thePixels).rowBytes = 0;
  }

  return errStat;
}
```

▶ Color Patterns

Chapter 2 introduced pen patterns and the Pattern data structure as the facility by which texture is added to shape and line drawing. Color QuickDraw updates the pen pattern concept substantially with the introduction of the **PixPat**, described by the data structure shown in Listing 3-18.

Listing 3-18. PixPat data structure

```
typedef struct
{
  short         patType;
  PixMapHandle  patMap;
  Handle        patData;
  Handle        patXData;
  short         patXValid;
  Handle        patXMap;
  Pattern       pat1Data;
}
PixPat, *PixPatPtr, **PixPatHandle;
```

The PixPat is based upon the PixMap data structure, as the monochrome Pattern data structure is based on the BitMap, albeit loosely. However, a PixPat has a number of attributes that its 1-bit cousin lacks:

- **Variable size** — Patterns are limited to an 8-by-8 pixel size. PixPats can have any vertical and horizontal dimension that is an integral power of two between 2 and 128, inclusive. As a consequence of their variable size, PixPats are Handle-based.

- **Multiple drawing modes** — Patterns are always drawn *port-relative*: Black pixels are drawn in the GrafPort's foreground color; white pixels are drawn in the background color. PixPats are much more flexible: They can be relative or absolute. An absolute PixPat is one where the PixPat's PatMap ColorTable has an entry for every pixel value. Absolute PixPats aren't affected by the current pen colors. Any pixel value outside of the ColorTable is drawn relative to the current pen colors. Relative PixPats are those who have at least one pixel value outside of PixPat's ColorTable.

Additionally, PixPats can also be *device-relative*, where each pixel value is calculated to simulate a color not available on the current GDevice. Such PixPats are adjusted before they're referenced for drawing, to ensure the best possible match to the current GDevice pixel depth and color settings.

▶ Using PixPats

The **NewPixPat**(), **CopyPixPat**(), and **DisposPixPat**() traps perform the same role for PixMaps that the **NewRgn**(), **CopyRgn**(), and **DisposRgn**() traps do for regions. The most common PixPat-related crashes occur because a PixPatHandle was used before memory for it was allocated. The **GetPixPat**() trap creates a PixPat from a specified resource template. PixPats may be passed as arguments to Color GrafVerb Fill traps, whose argument lists are similar to their Classic QuickDraw counterparts. For example:

```
FillRect( &someRect, somePattern );

FillCRect( &someRect, somePixPatHdl );
```

▶ CGrafPort PixPats

The CGrafPort contains PixPats for the current, fill, and erase patterns, just as the GrafPort does. The Classic QuickDraw **PenPat**() and **BackPat**() traps function with CGrafPorts, changing the CGrafPort's PixPats to port-relative PixPats. The new **PenPixPat**() and **BackPixPat**() traps take PixPatHandle arguments. Unlike the CFill GrafVerb traps, these pen traps "borrow" the PixPat for their own use. Disposing of a PixPat currently in use by a CGrafPort as a pen pattern or background (erasing) pattern usually causes Toolbox failure and/or application death.

▶ Creating Custom PixPats

PixPats are a substantial enhancement to the monochrome Pattern. However, PixPats are somewhat difficult to create, not from a data structure perspective, but from a data perspective: There are few means from which to create them from previously existing pixel images. The disk available with this book contains PixMap2Pat() and two other related routines, which facilitate the creation of PixPat resources (type 'ppat') from memory-based pixel images.

▶ Moving Pixels Around

You've seen that a PixMap is the general-purpose way that QuickDraw keeps track of a picture and that they're used in a number of different places. So it stands to reason that QuickDraw should provide a general-purpose tool for copying data between PixMaps. The tool exists and its name is **CopyBits**(). The name is held over from the old days when monochrome pixels and bits in RAM were virtually interchangeable concepts. **CopyBits**() is an extremely powerful trap, having a number of important features:

- Transfers pixels between pixel images, regardless of location (onscreen or offscreen) or differences in bits per pixel. Operations between PixMaps and BitMaps are fully supported.
- Automatically scales pixel values to account for differences between source and destination images.
- Supports all pen transfer modes.
- Transfer operations are extremely rapid, minimizing execution time and flicker.

The number of embedded features and relatively fast performance of **CopyBits**() has made it the trap of choice of many graphics programmers. There are a few other traps that perform pixel data processing operations, but **CopyBits**() is by far the most important of them. Figure 3-16 illustrates the work performed this trap.

Note ▶

Most QuickDraw traps assume the current GrafPort. **CopyBits**() is an exception to this rule in that the destination PixMap must be passed in explicitly.

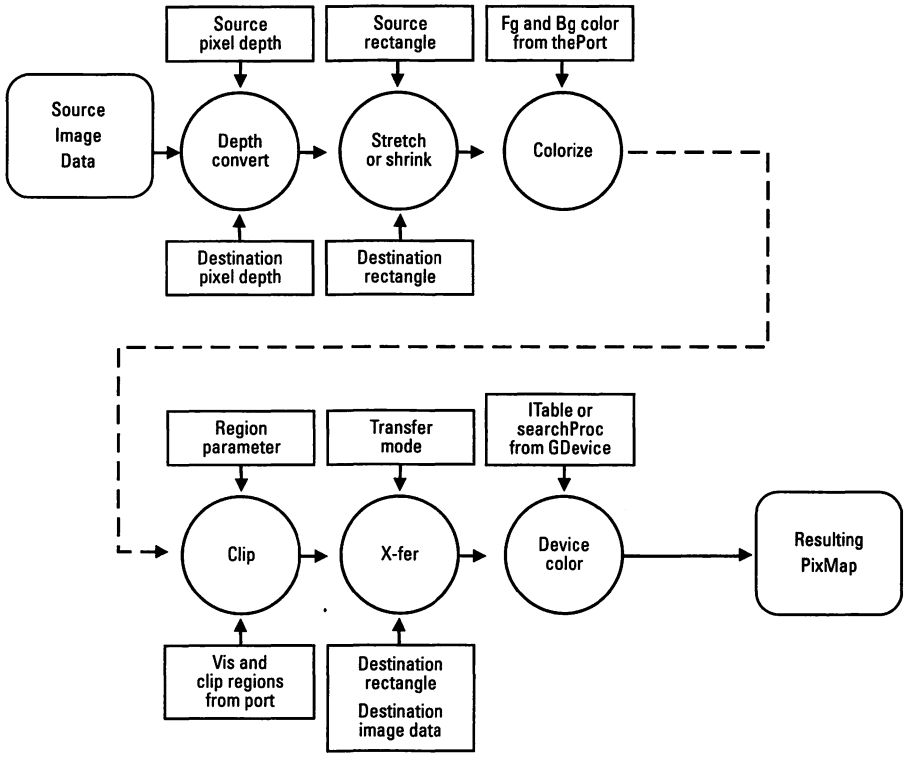


Figure 3-16. CopyBits() functional process diagram

▶ Introducing CopyBits()

In order to make good use of this trap, you have to roll up your sleeves because there's a lot to get into. The interface to **CopyBits()** is:

```
void CopyBits(
  BitMap      *srcBits,
  BitMap      *dstBits,
  Rect        *srcRect,
  Rect        *dstRect,
  short       transferMode,
  RgnHandle   maskRgn );
```

The first two arguments are pointers to the source and destination PixMaps or BitMaps. Listing 3-19 shows a routine that copies a picture from an off-screen PixMap to a window on the screen.

Listing 3-19. Simple use of the CopyBits() trap

```
void MoveSomePixels(
  PixMapHandle  offscreenPM )
{
WindowPtr   someWindow;

  if (rampPM == NULL)
    RETURN;

  someWindow = FrontWindow();
  if (someWindow == NULL)
    RETURN;

  /* CopyBits() requires these "standard" pen colors */
  /* for "basic" pixel image transfers */
  ForeColor( blackColor );
  BackColor( whiteColor );

  /* Note: PixMaps will need to be locked */

  /* blast PixMap data into a window somewhere */
  CopyBits(
    *offscreenPM, *(((CGrafPtr)someWindow)->portPixMap),
    &((**offscreenPM).bounds), &(someWindow->portRect),
    srcCopy, NULL );
}
```

Notice that the destination PixMap has been extracted from a WindowPtr and appropriately dereferenced. Realize that the window might not be a color window! Fortunately, **CopyBits**() can recognize this and convert the improper reference to a nonexistent PixMap into the correct reference to the window's GrafPort BitMap. This feature is necessary to provide compatibility with Classic QuickDraw.

The third and fourth arguments sent to **CopyBits**() are bounding rectangles. They define the subset of the pixels in the source and destination PixMaps, respectively. Recall the *bounds* field from the definition of the PixMap structure. It describes not only the PixMap's displayable area but its private coordinate system. The upper-left corner of a PixMap's bounds rectangle determines the coordinates of the first pixel in the PixMap, and the bottom-right coordinates describes the last visible pixel.

▶ Horizontal and Vertical Scaling

One of the more important features of **CopyBits**() is that it can be used to scale the source pixel values into the destination PixMap's rectangle. If the rectangles are of equal width and height, no such scaling takes place. If the source rectangle width is greater than the destination rectangle, source pixel values will be omitted according to a number of different rules in order to squeeze the source to fit inside the destination. The ratio of horizontal and vertical dimensions between the source and destination rectangles is calculated as a fixed-point decimal and appropriately rounded off for each destination pixel value calculated. For example, if the source is twice as wide and four times as tall as the destination, then the ratio is source[2 • 4] into destination[1 • 1], which means that from the source, 2 horizontal and 4 vertical pixel values will be used to compute a single destination pixel value. The nature of the computations depend on the QuickDraw version and source pixel image.

1. **1-bit source PixMaps and BitMaps** — QuickDraw will preserve black values. If there is a single pixel value in the source pixel sub-matrix, then the destination pixel value is black.
2. **2-bit and deeper source PixMaps** — The upper-left source pixel value is placed into the destination.

This is equally applicable to the relative heights of the source and destination rectangles. The scaling effects are perhaps easier to understand visually, as shown in Figure 3-17.

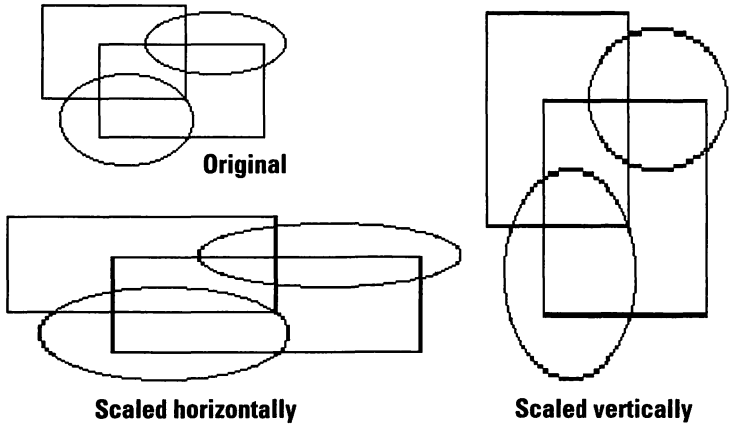


Figure 3-17. Examples of CopyBits() spatial scaling

▶ Examples of Using CopyBits()

Simple examples will help you understand the interaction between a PixMap's *bounds* rectangle and the "srcRect" argument in **CopyBits()**. Consider the code fragment in Listing 3-20, which sets up a PixMap which describes an image that is four rows by four columns of 8-bit deep pixels.

Listing 3-20. Creating another example indexed PixMap

```
void CreateIndexedPixMap( void )
{
PixMapHandle  srcPM;
Rect          pmBounds;
char          *pixBufPtr;
OSErr         errStat;
long          nPixels;
short         i;

  /* Allocate a new PixMap via our utility routines */
  SetRect( &pmBounds, 0, 0, 4, 4 );
  srcPM = PixMap_Create( &pmBounds, 8, &errStat );

  /* always check for failure when allocating large */
  /* data structures! */
  if (errStat != noErr)
    return;

  /* assign pixel values */
  /* pixel bytes = 4 pixels * 4 rows * (rowbytes / 8) */
  pixBufPtr = (char*)GetPixBaseAddr( srcPM );
  pixelBytes = 16;
  for (i=0; i<pixelBytes; i++)
    pixBufPtr[i] = (char)i;
}
```

Figure 3-18 shows what the contents of the PixMap's *baseAddr* memory looks like. Figure 3-19 shows the same PixMap as it would appear in a window.

Now let's examine what happens when the example PixMap's contents are copied using **CopyBits()**. Assume that the destination PixMap is 8 by 8 and the upper-left corner of its coordinate system is anchored at (0, 0). For the first example of a **CopyBits()** operation, let's consider the simple pixel copy operation shown in Listing 3-21.

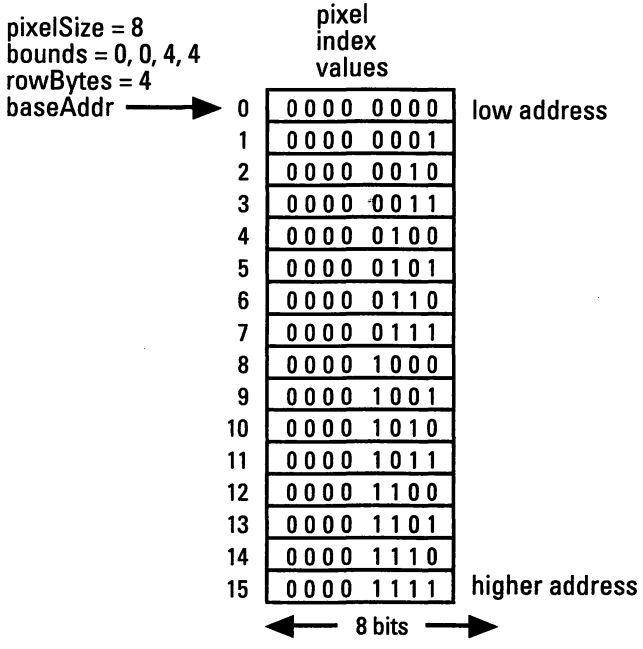


Figure 3-18. SomeRoutine() PixMap contents (memory)

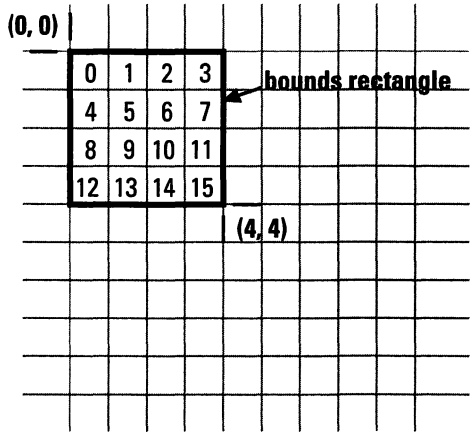


Figure 3-19. SomeRoutine() PixMap contents (in window)

Listing 3-21. Copying all pixel values

```
/* Copy all pixels */
srcRect = (**srcPM).bounds;
dstRect = srcRect;
CopyBits( *srcPM, *dstPM, &srcRect, &dstRec, srcCopy, NULL );
```

Figure 3-20 shows the destination space before the **CopyBits**(); the gray pixels represent whatever existed there before **CopyBits**() was called. Figure 3-21 shows the destination space after calling **CopyBits**().

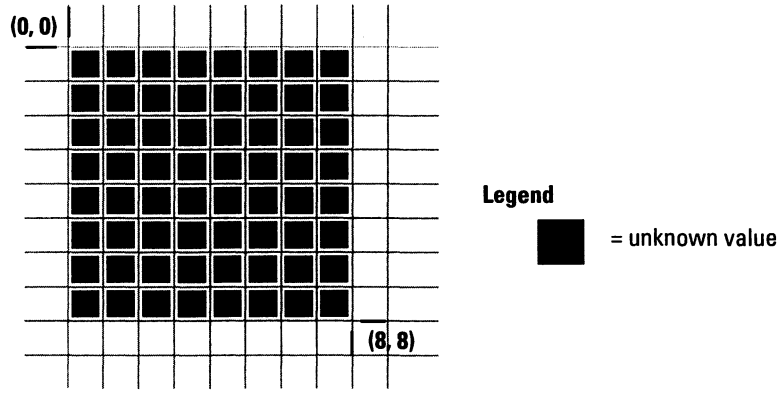


Figure 3-20. Before PixMap, CopyBits() example #1

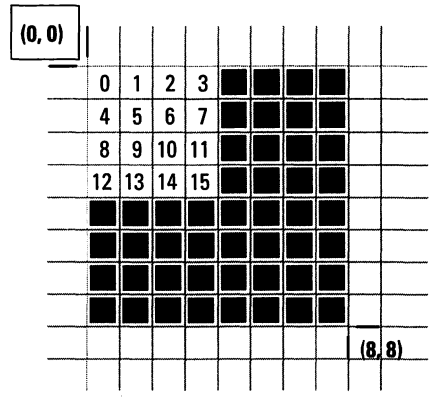


Figure 3-21. After PixMap, CopyBits() example #1

The result shouldn't be surprising, for two reasons: First, "srcRect" was set equal to (**srcPM).bounds, which directed **CopyBits**() to use all of the visible pixels in srcPM. Then "dstRect" was set equal to srcRect, which directed **CopyBits**() to copy all source pixels into the destination, without scaling.

Let's make it a little more interesting. Instead of setting srcRect equal to (**srcPM).bounds as in the last example, let's use srcRect as a window into the source PixMap as shown in Listing 3-22.

Listing 3-22. Copying only center pixel values

```
SetRect( &srcRect, 1, 1, 3, 3 );
dstRect = srcRect;
CopyBits( *srcPM, *dstPM, &srcRect, &dstRec, srcCopy, NULL );
```

Figure 3-22 shows the results of this operation.

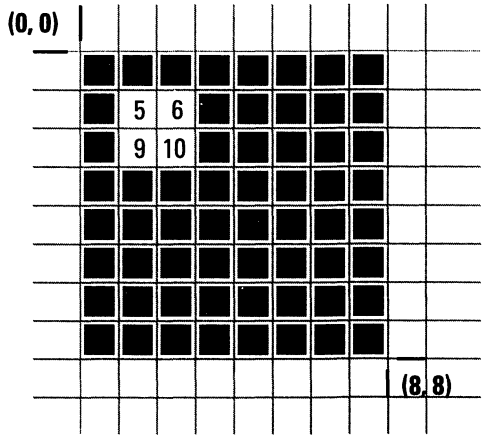


Figure 3-22. PixMap results, CopyBits() example #2

The srcRect variable is used as a 2-by-2-pixel window into the source PixMap. Setting srcRect to (1, 1, 3, 3) directed **CopyBits**() to use the center 4 pixels because — and this is important — srcRect uses the coordinate system as defined by the bounds rectangle of source PixMap; in this case, (**srcPM).bounds is (0, 0, 4, 4), a 4-by-4 square anchored at (0, 0). Figure 3-23 shows the relationship between srcRect and (**srcPM).bounds.

Let's make another minor change, as shown in Listing 3-23.

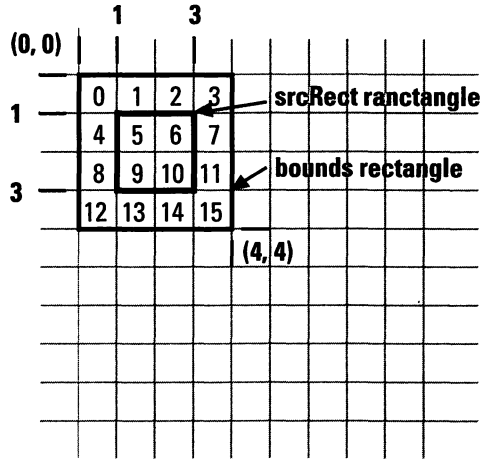


Figure 3-23. PixMap results, CopyBits() example #3

Listing 3-23. Copying the source's center into the upper-left corner

```
SetRect( &srcRect, 1, 1, 3, 3 );
dstRect = srcRect;
OffsetRect( &dstRect, 0, 3 );
CopyBits( *srcPM, *dstPM, &srcRect, &dstRec, srcCopy, NULL );
```

This time, instead of making srcRect and dstRect equal, we set dstRect to be the same size (2 by 2) as srcRect, but now it's anchored at (1, 4) instead of (1, 1). Figure 3-24 illustrates what happens.

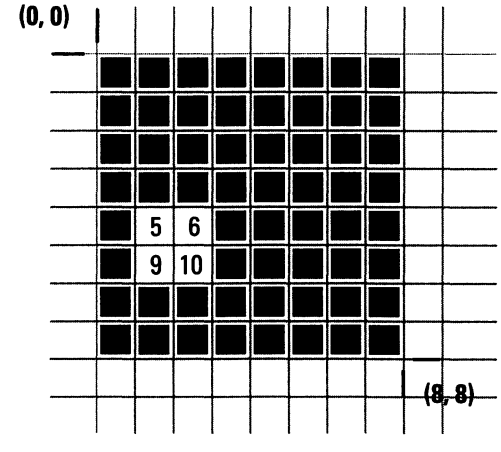


Figure 3-24. PixMap results, CopyBits() example #4

▶ Performance Considerations

These examples should give you an idea of how **CopyBits**() can be used to transfer and scale pixel images in horizontal dimensions. A pixel image can also be treated as a three-dimensional beast: The coordinates (x, y, z) map into (horizontal, vertical, pixel value). Figure 3-25 hints at this three-dimensional nature.

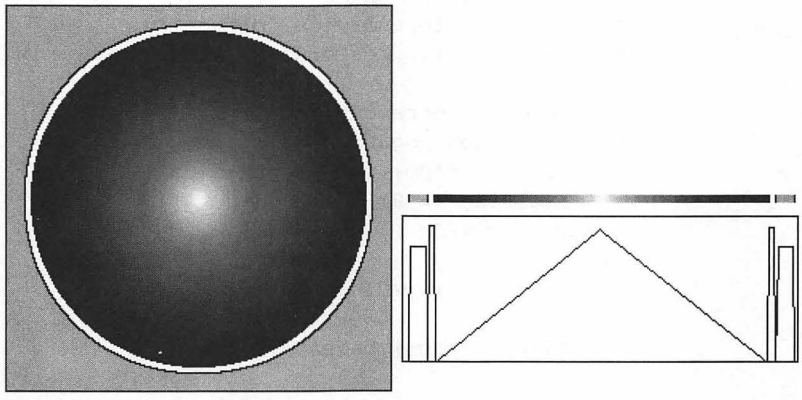


Figure 3-25. Profile of a grayscale image

In Chapter 7, we examine how **CopyBits**() can be used to transform pixel images in the z (pixel value) dimension. When **CopyBits**() calls are made using a srcCopy transfer mode, a black foreground color, and a white background color, transformations in the pixel value dimension are effectively suppressed.

The routines employed to transfer blocks of raw pixel data between video and nonvideo RAM are commonly referred to as *blitters*. A pixel-blitting routine has to be fast. The processing of pixel images naturally involves nested loops something like those shown in Listing 3-24.

Listing 3-24. Framework for a blitting routine

```
void PixBlitter( void )
{
register Ptr    curPixelPtr;  /* hopefully this group */
register short  pixelSize, i; /* gets in registers */

register Ptr    curPixelPtr;  /* if these make it, */
register long   bytesPerRow;  /* even better */
register short  j;
```

```
Rect            boundsR;

  /* determine number of bytes in each row of pixels */
  /* include pixels outside to the right of the bounds rect */
  bytesPerRow =
    (long)((**pm).bounds.right - (**pm).bounds.left) *
    (long)(pm.rowBytes & 0x7FFF);

  /* account for the number of bytes per pixel value */
  /* multiply before divide, please */
  bytesPerRow = ((bytesPerRow * 8) / (**pm).pixelDepth;

  /* determine increment for */
  /* must be non-negative */
  pixelSize = (**pm).pixelDepth;
  if (pixelSize <= 8)
    pixelSize = 1;
  else
    pixelSize = pixelSize / 8;

  startAddr = (**pm).baseAddr;

  for (j=boundsR.top; j<boundsR.bottom; j++)
  {
    curPixelPtr = startRowAddr;

    /* process a row of pixel values */
    for (i=boundsR.left; i<boundsR.right; i++)
    {
      /* ••• do interesting pixel operations here ••• */

      /* ••• if resolution is less than 8 bits per pixel ••• */
      /* ••• do multiple pixels at a time ••• */

      /* advance current pixel pointer */
      curPixelPtr = curPixelPtr + pixelSize;
    }

    /* advance the row pointer */
    startRowAddr = startRowAddr + bytesPerRow;
  }
}
```

The primary influence upon a blitter's performance is the number of CPU cycles executed in the innermost loop. This generally means writing code that generates a small number of simple instructions. For example the DIV (divide) is an order of magnitude slower than the LSR (rotate right) instruction. If a blitter's innermost loop needed to perform division by an integral power of 2, then performing the divide by right-shifting would yield much shorter execution time than using a "plain" divide operator. Adds, subtracts, and bit-shifting operations are all preferable to multiplication and division.

The blitting routine shown in Listing 3-24 is meant only to illustrate the complex nature of copying pixel images. For most applications, you shouldn't feel the need to write such a routine. But understanding the basics of blitting operations should help you when writing routines that examine the pixel image data contained within PixMaps and BitMaps.

In terms of raw execution speed, **CopyBits**() performs impressively. The performance is even more impressive when you consider the number of operations that it can perform just by varying the pen mode and rectangle arguments. As expected, the complex multi-mode functionality of **CopyBits**() consumes more CPU than using a customized blitting routine. So how does an application avoid an unnecessary large performance penalty when using **CopyBits**()?

- **Force long word alignment between source and destination images** — See the section on PixMap alignment in the next chapter.
- **Don't colorize** — Using pen colors other than black and white for the foreground and background slows **CopyBits**() down considerably.
- **Don't scale** — Scaling occurs when copying between rectangles of unequal sizes and means extra inner integer or fixed point multiplications. Use source and destination rectangles of equal dimensions.
- **Use equal resolutions** — PixMap resolutions factor into the scaling equation; differences in resolution will cause scaling and engender added calculations. **CopyBits**() is actually ignorant of resolution and effectively factors resolution against the source and destination rectangles.
- **Transfer between PixMaps of equal depth** — This should be obvious. Converting between pixel depths involves some amount of extra instructions to round off, truncate, or sign-extend pixel values.

- **Use srcCopy mode** — Since the destination pixel values can be ignored, fast move operations can be employed. The other logical pen modes are somewhat slower. Arithmetic operations are by nature computation-intensive, and they slow down **CopyBits**() and other transfer operations considerably.
- **Make source PixMap and the destination *GDevice* have identical ColorTables** — For the purposes of color mapping, all direct PixMaps have identical ColorTables. As mentioned earlier in this chapter, indexed ColorTables are equivalent if they have the same ctSeedvalue.
- **Don't clip the destination** — Pass **CopyBits**() NULL for "maskRgn" arguments. Clipping may occur when copying into a destination PixMap that happens to belong to the current GrafPort. Clipping occurs if either the mask or clip region intersects the destination rectangle.
- **Remove all installed search procedures** — Search procedures slow CopyBits() down considerably if installed on the current GDevice.

By the Way ▶

In 1987, Andy Hertzfeld demonstrated a few other techniques that ultimately became part of QuickDraw. He released QuickerGraf, an INIT that patched out **CopyBits**() and other traps with replacement routines that contained such tricks as special-cased inner loops for different pixel depths.

With the release of HyperCard, Bill Atkinson demonstrated the value of long-word alignment of source and destination pixel memory buffers.

▶ PixMap Resolution and Patching the StdBits() Bottleneck

You may recall that there are two fields, *hRes* and *vRes*, that indicate the PixMap's resolution. These hold two 32-bit fixed-point values that indicate the number of pixels per inch. These values are important, but they are particularly important when copying between PixMaps of differing resolutions. The typical resolution of a "native" Macintosh PixMap is 72 dpi; however many imported pixel images contain PixMaps of 300 dpi or greater.

In order to properly account for differences in resolution between PixMaps, an application can employ a small number of tricks. One is to scale all rectangle coordinates before passing them on to **CopyBits()**. An arbitrary sub-rectangle in a 72-dpi PixMap is about one-sixteenth as pixel-data-dense as the same-size rectangle in a 300-dpi PixMap. Therefore, a rectangle in a 300-dpi source should be copied and its lower-right coordinates scaled down by a factor of $^{72}/_{300}$ relative to the upper-left coordinates in order to use it as an appropriate destination in a 72-dpi PixMap. Alternatively, all four of the rectangle's coordinates could be inset to center-justify the source image within the destination.

Another alternative is to patch the GrafPort's **StdBits()** bottleneck procedure with one that scales the destination rectangle whenever copying between PixMaps of differing resolutions. Listing 3-25 shows an example of such a patch and its installation.

Listing 3-25. Making StdBits() account for "nonstandard" resolutions

```
pascal void StdBitsProc_ResolutionIndependent(
  BitMap      *sourceBits,
  Rect        *sourceR,
  Rect        *destR,
  short       transferMode,
  RgnHandle   maskRgn )
{
Rect      newSourceR, newDestR;
Fixed     seventyTwo, sourceRes, destRes;
long      width, height; /* long to avoid calculation overflow */
Boolean   sourceIsPM, destIsPM;

  /* modify a copy of rectangles, not the originals */
  newSourceR = *sourceR;
  newDestR = *destR;

  /* check to see if one of the BitMaps is a PixMap */
  sourceIsPM = ((sourceBits->rowBytes & 0x8000) == 0x8000);
  destIsPM = ((destBits->rowBytes & 0x8000) == 0x8000);

  if (sourceIsPM || destIsPM)
  {
    /* equal to FixRatio( 72, 1 ) */
    sourceRes = 0x48000000;
    destRes = 0x48000000;
```

```
    /* horizontally scale the destination and source rects */
    if (sourceIsPM)
      sourceRes = ((PixMap*)sourceBits)->hRes;
    if (destIsPM)
      destRes = ((PixMap*)destBits)->hRes;
    if (sourceRes != destRes)
    {
      width = sourceR->right - sourceR->left;
      width = FixRound( width * FixDiv( destRes, sourceRes ) );
      newDestR.right = newDestR.left + width;
    }

    /* vertically scale the destination and source rects */
    if (sourceIsPM)
      sourceRes = ((PixMap*)sourceBits)->vRes;
    if (destIsPM)
      destRes = ((PixMap*)destBits)->vRes;
    if (sourceRes != destRes)
    {
      height = sourceR->bottom - sourceR->top;
      height = FixRound( height * FixDiv( destRes, sourceRes ) );
      newDestR.bottom = newDestR.top + height;
    }
  }

  StdBits( sourceBits, &newSourceR, &newDestR, transferMode,
    maskRgn );
}


/*
**install custom CopyBits() proc into a CGrafPort or GrafPort,
**or standard StdBits() proc if "newBitsProc" argument is NULL.
*/
void InstallBitsProc(
  CGrafPtr theCurPort,
  Ptr      newBitsProc )
{
CQDProcs    stdCQDProcs, *newCQDProcsPtr;
QDProcs     stdQDProcs, *newQDProcsPtr;
Boolean     isColorPort;

  if (theCurPort == NULL)
    theCurPort = (CGrafPtr)thePort;
```

```
    /* check for color or b&w GrafPort */
    isColorPort = ((theCurPort->portVersion & 0xC000) == 0xC000);

    if (theCurPort->grafProcs == NULL)
    {
      if (isColorPort)
      {
        newCQDProcsPtr = (CQDProcsPtr)NewPtr( sizeof(CQDProcs) );
        SetStdCProcs( newCQDProcsPtr );
        theCurPort->grafProcs = newCQDProcsPtr;
      }
      else
      {
        newQDProcsPtr = (QDProcsPtr)NewPtr( sizeof(QDProcs) );
        SetStdProcs( newQDProcsPtr );
        theCurPort->grafProcs = newQDProcsPtr;
      }
    }

    if (newBitsProc != NULL)
      /* install custom proc */
      theCurPort->grafProcs->bitsProc = newBitsProc;
    else
    {
      /* install "standard" proc */
      if (isColorPort)
      {
        SetStdCProcs( &stdCQDProcs );
        theCurPort->grafProcs->bitsProc = stdCQDProcs.bitsProc;
      }
      else
      {
        SetStdProcs( &stdQDProcs );
        theCurPort->grafProcs->bitsProc = stdQDProcs.bitsProc;
      }
    }
}
```

▶ Summary

This chapter introduced Color QuickDraw and 32-bit QuickDraw, along with the basic color theory needed to support the context of color functionality. The salient difference between Classic QuickDraw and Color QuickDraw is the use of multi-bit pixels that allow for color programming.

The chapter also presented the two color models that Color QuickDraw supports, indexed and direct. The indexed color model is the most commonly used pixel type; it uses a ColorTable to store a list of available colors. The direct color model does not require a ColorTable because each pixel value contains color information. The chapter also presented the GDevice as the data structure for defining a color model and the resulting organization of pixel memory.

QuickDraw's major data structure for handling pixel images is the PixMap, which describes the organization of raster images in Macintosh memory. The PixMap is the extension of the Classic QuickDraw's BitMap structure. Copying images from PixMap to PixMap is accomplished by using **CopyBits()**.

The next chapter examines these concepts in greater depth; the emphasis in this chapter was on basic practical information. Chapter 4 describes how palettes work and how more control can be exerted in the color mapping process.

4 ▶ Display Control and Offscreen Drawing

In Chapters 2 and 3, we explored many of the fundamental Classic QuickDraw and Color QuickDraw concepts, data structures, and traps. This chapter focuses on integrating those concepts to produce a useful and practical foundation on which graphics can be rendered and displayed in a robust, flicker-free manner, with a minimum of programming effort.

Because Macintosh systems have a wide variety of display hardware configurations, graphics applications must strive to be as independent of the video display hardware as possible while simultaneously taking maximum advantage of the available graphics hardware resources. The chapter title reflects the relationship between these two goals.

▶ The Need for Drawing Offscreen

At a minimum, it's fair to say that the QuickDraw environment provides basic graphics functionality. But the ability to draw text and shapes just isn't enough for most commercial-grade Macintosh applications, which demand solid, unhesitating responses to mouse and keyboard actions. The ability to render complex graphics offscreen and transfer the results onscreen quickly is an essential part of computer graphics presentation and is well-supported by the Macintosh Graphics Toolbox. In order to understand the relationship between graphics software and hardware architectures, a quick video overview is in order.

▶ Video Display Overview

The hardware that converts pixel values in video memory into analog signals is called *digital-to-analog hardware* (D/A). A single video image is called a *frame*. A single line of drawn pixels is called a *scan line*. The period of time between reaching the end of a one scan line and start of drawing on the next scan line is called the *horizontal blanking interval*, or *hblank*. The period in time between reaching the end of the last scan line and start of drawing on the first scan line of the next frame is called the *vertical blanking interval*, or *vblank*. Figure 4-1 illustrates the blanking intervals relative to the scan pattern of a typical video display.

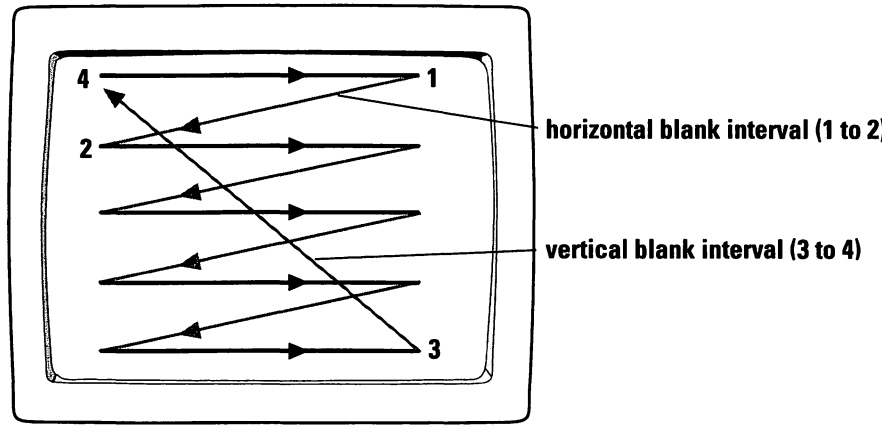


Figure 4-1. Video display scan pattern

Of particular interest is the vblank interval. Realize that CPU's operate rates (8–50 megahertz, or million cycles per second) are much faster than those of video displays, where each frame is redrawn every 1/30 of a second. It's not only possible but desirable to update display characteristics between video frames or fields to ensure smooth transition between frames.

▶ Causes of Flicker

Even though hardware performance increases are regularly doing wonders for even the most sluggish, convoluted application, the video rate bottleneck remains. The speed mismatch between CPUs and video display is the source of great difficulty. The problems start with the

fact that most video memory is accessed through shared address lines. The flicker problem manifests itself when the D/A hardware that controls a monitor also tries to read that same video memory. There's no problem if the monitor is in its vblank or hblank, because the D/A hardware isn't reading the video memory. If the D/A is reading while the bus is also trying to write or read the same memory, flicker results. The Macintosh CPU and bus hardware, through the marvels of hardware bus arbitration logic, are kept from interfering with each other when reading or writing memory at nearly the same time. The problem is resolved by delaying all bus access to video memory until vertical or horizontal blank, thereby avoiding flicker due to access during the nonblanking intervals.

The most common cause of video flicker is the double-drawing of pixel values. Applications that fail to optimize their rendering methods often flicker noticeably during window updates and mouse tracking. Many of these flicker-producing techniques are common: completely erasing a window before drawing its contents, xor-ing a shape during a sketching loop, and so on. But even the most basic sequences of graphic operations can cause flicker due to double-drawing pixels: Consider the example in Figure 4-2, which shows how flicker affects graphics that track the current mouse position.

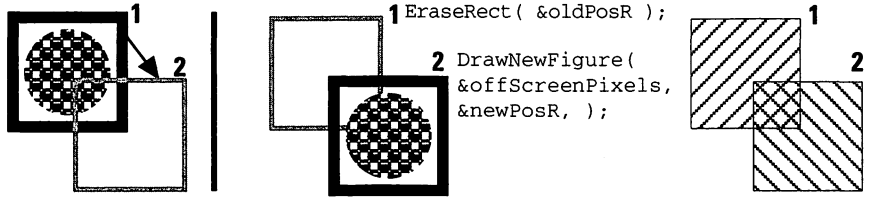


Figure 4-2. Example of flickering from mouse tracking graphics

Flicker is caused by the pixels in the intersection of the shape at the last position and the new position being erased and then redrawn with the desired graphic. It would be desirable to have operations that change onscreen pixel values more than once to buffer those writes, but few display cards have this capability. Display cards simply lack

the raw execution speed necessary to process pixel images at video rates, unless the destination image is small and whose computation doesn't involve horizontal, vertical, or pixel value scaling.

The flicker problem is inherent in most commercial microcomputer video architectures, not just the Macintosh. The preferred methodology to eliminate flicker, one favored by most graphics applications, involves performing all drawing operations to an offscreen pixel memory buffer, and then copying the buffer contents into the active video display controller's video memory in a single operation. Figure 4-3 illustrates this process.

This technique is known by a number of names; we're going to call it simply *offscreen drawing*.

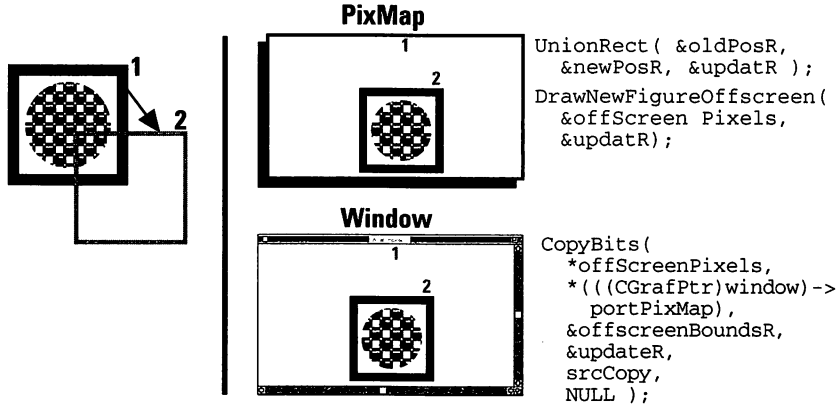


Figure 4-3. Offscreen drawing procedure

▶ Basic Offscreen Drawing Procedure

Graphics-intensive applications must adapt to a wide range of display configurations. The main tenet of offscreen drawing is the "decoupling" of the software components that perform image rendering and image display. Another important consideration is to rasterize graphic objects as close to display device itself. This technique involves the creation of an offscreen graphics software environment that models closely the ideal hardware configuration, without imposing the restraints of a physical graphics device.

The primary limitations of offscreen graphics are the availability of memory and maximum allowable execution time. Image buffers are

memory hogs, and rasterizing and pixel image transfer operations are very execution-intensive. An effective offscreen drawing procedure must deal with these two limitations in great detail and with flexibility.

The GDevice was designed to facilitate graphics software/hardware independence by placing a device driver between application software and low-level graphics hardware control. The important tasks that the GDevice driver carries out are handling the lowest levels color mapping and pixel memory addressing. The GDevice scheme is also designed to arbitrate color requests in offscreen pixel images, where hardware limitations don't apply. These functions are to be carried out with a minimum effect on execution speed, if only because the sheer magnitude of image data tends to magnify execution overhead. A device driver is also expected to simplify for application software the tasks related to hardware device I/O, which most applications depend upon heavily. For example, tasks like file and screen reads and writes.

In these respects, the GDevice driver specification is both sufficient and sensible. Most of the time, applications barely have to concern themselves with the current state of the display hardware. Unfortunately, the addition of GDevices to the support of offscreen drawing is neither simple nor transparent, and requires no small amount of careful programming. Using CGrafPorts and GDevices together with their associated data structures to create offscreen graphics buffers is one of the most confounding tasks of QuickDraw programming. Additionally, there is duplication of important information between GDevices and CGrafPorts, particularly boundary rectangles, pixel maps, and ColorTables. The steps for creating an offscreen graphics environment from scratch proceed as follows:

1. If the graphics to be rendered differ from the monitor that they will be displayed on, then a "private" GDevice is necessary. Create one using **NewGDevice**(). Save the current GDevice reference and make the newly created GDevice the current one.
2. Initialize the GDevice's PixMap. Assign the *pixelSize* field to the desired pixel depth and bounds rectangle to the desired area. Compute the *rowBytes* based on the pixel size and bounding rectangle. Allocate enough memory to accommodate the desired image (using **NewPtr**() or **NewHandle**()).
3. If the indexed color model is desired, create and install a ColorTable into the GDevice's PixMap. Then create and install an inverse table using **MakeITable**().

4. Create a CGrafPort with **OpenCPort**() and make it the current port, using **SetPort**(). Copy the current GDevice's PixMapHandle into the CGrafPort's PixMap via **CopyPixMap**().
5. Whenever drawing is to be performed offscreen, save the current GDevice and CGrafPort, make current the private GDevice and CGrafPort, draw, and restore the original GDevice and CGrafPort when finished with the drawing operations.

As you can see, this is a fairly involved procedure, given that each one of these steps involves significant calculation and error checking.

▶ GWorlds

Apple recognized the complexity of this data structure concerto and the associated programming difficulties. With the release of 32-bit QuickDraw version 1.0's Graphics World paradigm, simply known as GWorlds, Apple introduced a simple and powerful alternative to the laborious offscreen drawing process described in the previous section. A GWorld contains a CGrafPort, a PixMap, a ColorTable, and a GDevice; the details of managing them are neatly encapsulated (hidden away) in a handful of routines contained in the 32-bit QuickDraw package. It's important to understand the roles of each of the four component data structures: those roles will be discussed in greater detail later in this chapter. But for now, let's concentrate on basic GWorlds usage. It may help you to think of a GWorld as a sort of "super" CGrafPort with its own private GDevice sharing a common PixMap.

Note ▶

You may be wondering how GWorlds can work in noncolor QuickDraw environments. As of System 7 QuickDraw version 1.3, GWorlds are implemented for monochrome QuickDraw. This capability greatly reduces the need for special-case graphics code. Now, with much less effort than ever before, you can write generalized, Macintosh-model-independent graphics code.

▶ Using GWorlds

GWorlds are primarily used as a drawing area associated with a particular window. This isn't a requirement: A GWorld may be used simply as a "private" drawing area without being attached to any particular window.

A list of the most important GWorld traps, and descriptions of how to use them effectively, follows. The more esoteric GWorld facilities will be addressed in later chapters.

Creating a GWorld

```
pascal QDErr NewGWorld(
  GWorldPtr   *offscreenGWorld, /* Note: a Var param in Pascal */
  short       pixelDepth,
  Rect        *boundsRect,
  CTabHandle  cTable,
  GDHandle    aGDevice,
  long        gdFlags );
```

The **NewGWorld()** trap is used to allocate and initialize a GWorld. It returns a Ptr (not a Handle) to the new GWorld and may also be cast (or type-coerced) as a CGrafPtr and treated as such. This trap passes back a return value that indicates whether or not the requested GWorld could be created.

Changing a GWorld

```
pascal GWorldFlags UpdateGWorld(
  GWorldPtr   *offscreenGWorld, /* Note: a Var param in Pascal */
  short       pixelDepth,
  Rect        *boundsRect,
  CTabHandle  cTable,
  GDHandle    aGDevice,
  long        gdFlags );
```

This trap is used to reconfigure an existing GWorld. It's most commonly used to change either the pixel depth or ColorTable, but it can be employed to change other GWorld characteristics as well.

Handling GWorld Components

```
pascal void GetGWorld(
  CGrafPtr *port,    /* Note: a Var param in Pascal */
  GDHandle *gdh );   /* Note: another Var param in Pascal */

pascal void SetGWorld(
  CGrafPtr port,
  GDHandle gdh );
```

As with GrafPorts, QuickDraw maintains a current GWorld, which is the combination of the current CGrafPort and GDevice. These two

traps perform the appropriate **Get/SetGDevice**() and **Get/SetPort**() calls. If the CGrafPtr argument passed to **SetGWorld**() is in fact a GWorldPtr, then NULL can be passed for the GDHandle argument: **SetGWorld**() will obtain the GDevice reference from the GWorld itself.

```
pascal PixMap Handle GetGWorldPixMap(
  GWorldPtr   offscreenGWorld );

pascal GDHandle GetGWorldDevice(
  GWorldPtr   offscreenGWorld );
```

These traps return Handles to the GWorld's associated PixMap and GDevice, respectively. The PixMapHandle is commonly required for the **CopyBits**() call that transfers the GWorld contents to the display in response to an update event.

Note ▶

GetGWorldPixMap() has been completely dysfunctional in Color QuickDraw versions 1.2 and older. Fortunately, the PixMap reference can be obtained directly from a GWorldPtr by type-coercing it to a CGrafPort, as shown in Listing 4-1.

The System Software 6.0.7 release has actually fixed this release; unfortunately, the QuickDraw version number is still 1.2. In other words, some version 1.2s have a working version of this trap and some do not, so it's best to assume that all Color QuickDraw version 1.2s are broken in this respect.

Listing 4-1. Obtaining a GWorld's PixMapHandle

```
PixMapHandle BetterGetGWorldPixMap(
  GWorldPtr   aGWorld )
  short       qdVersionID
{
PixMapHandle  theGWPixMapHdl;
Boolean       isTrapBroken;

  isTrapBroken = (qdVersionID <= 0x0102);
  if (isTrapBroken)
    /* prefer using GetGWorldPixMap(), but it's broken, so… */
    theGWPixMapHdl = ((CGrafPort)aGWorld)->portPixMap;
  else
    /* in System 7.0 QuickDraw versions, we can do this instead */
```

```
    theGWPixMapHdl = GetGWorldPixMap( aGWorld );

  return theGWPixMapHdl;
}
```

Disposing of a GWorld

```
pascal void DisposeGWorld(
  GWorldPtr  offscreenGWorld );
```

Use the **DisposeGWorld()** trap to dispose of all memory allocated by **NewGWorld()**.

▶ More GWorld Techniques

The traps just presented were described as briefly as possible in order to convey the essence of GWorlds as concisely as possible. GWorlds have a number of other important features that will become important to you once you've started implementing your own GWorld-based applications.

Handling GWorld Errors

An application must be prepared to deal with the failure of most GWorld traps. It's not that GWorlds are unreliable; in fact, the traps are quite robust. However, GWorlds are quite hardware-resource-hungry and any code that relies on the availability of memory, disk space, or other such facilities absolutely must be prepared to handle a lack of them, both gracefully and clearly. To be more specific, an application that relies on GWorlds to perform double-buffered updates from offscreen should be prepared to draw directly to the screen should the necessary GWorld be unavailable. The Menu Manager uses offscreen buffering to save and restore the pixel values under a menu. When the Menu Manager cannot geet enough memory for the offscreen image buffer, it invalidates the area behind the menu to cause the update events necessary to redraw the screen, one window at a time.

GWorlds and their associated offscreen PixMaps usually require huge gobs of memory. The return values from **NewGWorld()** and **UpdateGWorld()** should always be checked for success. To give you an idea of how likely a **NewGWorld()** failure is, ponder this tidbit: A 512-by-512-pixel by 32-bits-per-pixel PixMap requires a one megabyte RAM allocation. How many GWorld-based, deep-pixel windows can your Macintosh handle?

So how should an application check the return values? The return values from GWorld traps indicate more than just execution failure or success. Under successful circumstances, the values should be interpreted as a bit-mapped flag value, with individual bits indicating actions that were undertaken to fulfill the request implied by the incoming trap arguments. The error bit is one of these bits: if set, the rest of the flag value can be interpreted as a bona fide error value. Listing 4-2 shows an example of how to process these return values.

Listing 4-2. Example of testing for GWorld errors

```
OSErr ChangeAGWorld(
  GWorldPtr   someGW,
  short       newPixDepth )
{
Rect          newBoundsR;
OSErr         errStat;
GWorldFlags   newGWFlags, curGWFlags;

  SetRect( &newBoundsR, 0, 0, 1024, 1024 );
  curGWFlags = alignPix | stretchPix | newDepth;

  /* go off and (try to) update the GWorld */
  newGWFlags =
    UpdateGWorld(
      &someGW, newPixDepth, &newBoundsR,
      (CTabHandle)NULL, (GDHandle)NULL, curGWFlags );

  /* determine if the last trap call failed */
  if ((newGWFlags & gwFlagErr) == 0L)
    /* success */
    errStat = noErr;
  else
    /* failure: mask out the error bit to get */
    /* actual error value */
    errStat = newGWFlags & ~gwFlagErr;

  return errStat;
}
```

This is certainly more involved than, say, checking a file system error.

Using NewGWorld() and UpdateGWorld()

The **NewGWorld**() and **UpdateGWorld**() traps each have six arguments, with certain combinations having special meanings. Therefore, the assignment and usage of the following arguments must be carefully understood before you use the **NewGWorld**() and **UpdateGWorld**() traps:

- "**offscreenGWorld**" — There's no mystery here, just pass in the address of an unallocated or allocated existing GWorldPtr variable to **NewGWorld**() and **UpdateGWorld**(), respectively.
- "**pixelDepth**" — This value needs to be a valid pixel depth. Zero is a special value that can be passed to **NewGWorld**(). It directs **NewGWorld**() to use the depth of the deepest display GDevice intersecting the "boundsRect" argument. The treatment of the "boundsRect" argument is the same as performed by the **GetMaxDevice**() trap.
- "**boundsRect**" — This value specifies the height, width, and coordinate system of the GWorld, just as every other QuickDraw bounds rect does.
- "**cTable**" **and** "**aGDevice**" — Most of the time a NULL value should be passed for these arguments. The primary exception to this guideline is when an application needs to change the colors available inside an indexed GWorld, that is, a GWorld with a pixel depth of 8 or less. It's perfectly allowable to allocate a private GDevice and assign it to the GWorld, but the creation and maintenance of a GDevice is one of the more important labor-saving features of a GWorld. Of course, you're (fairly) free to do what you want in the context of your application.
- "**gdFlags**" — This flag value should be composed by or-ing together one or more of the defined GWorldFlag constants. These should specify the desired GWorld attributes, as documented in *Inside Macintosh*, Volume VI.

Make sure that there is enough heap memory to hold an entire duplicate of a GWorld before passing it to **UpdateGWorld**(). This trap calls **NewGWorld**() to duplicate the GWorld itself, calls **CopyBits**() to transfer the pixel values, and then disposes the original GWorld.

Important ▶

There are problems associated with checking some of these flag bits in Color QuickDraw versions 1.2 and earlier. However, the error bit is always set appropriately.

Additionally, early versions of **UpdateGWorld**() have been known to corrupt the GWorld's clipping region and ColorTable.

Stretch, Clip, and Dither

These flag bits specify how **UpdateGWorld**() will change the pixel image when converting between ColorTable, geometry (horizontal and vertical scaling), and pixel size. The effects of such operations are discussed at length in Chapter 7.

PixMap Alignment

The concept of memory alignment was mentioned briefly in the previous chapter. Now we will discuss it more fully.

The architecture of the 68000 family of processors (as well as most others) facilitates the fastest transfer of memory values when the source and destination addresses are aligned on 16-bit (even byte) or 32-bit (multiple of 4 byte) addresses. On Macintoshes, memory and bus transfers operate the fastest if both addresses are even multiples of 4 bytes, that is, if neither of the two low address bits are ones. Figure 4-4 shows examples of aligned and non-aligned addresses.

The notion of memory alignment is of particular importance to graphics processing because of the large number of pixel-blitting operations and the vast amount of data associated with each operation.

GWorlds have a built-in provision to realign their image buffers to their display destination. The "whichFlags" argument to **UpdateGWorld**() has an *alignPix* bit, which, if set, will cause the trap to realign the image buffer via the following mechanism: A little-known detail about GWorld PixMaps is that each row of pixel values is actually 4 bytes longer than the allocation request called for and is accounted for in the PixMap's *rowBytes* value. This leeway allows **UpdateGWorld**() to shift around the bits of each row without having to completely reallocate the PixMap.

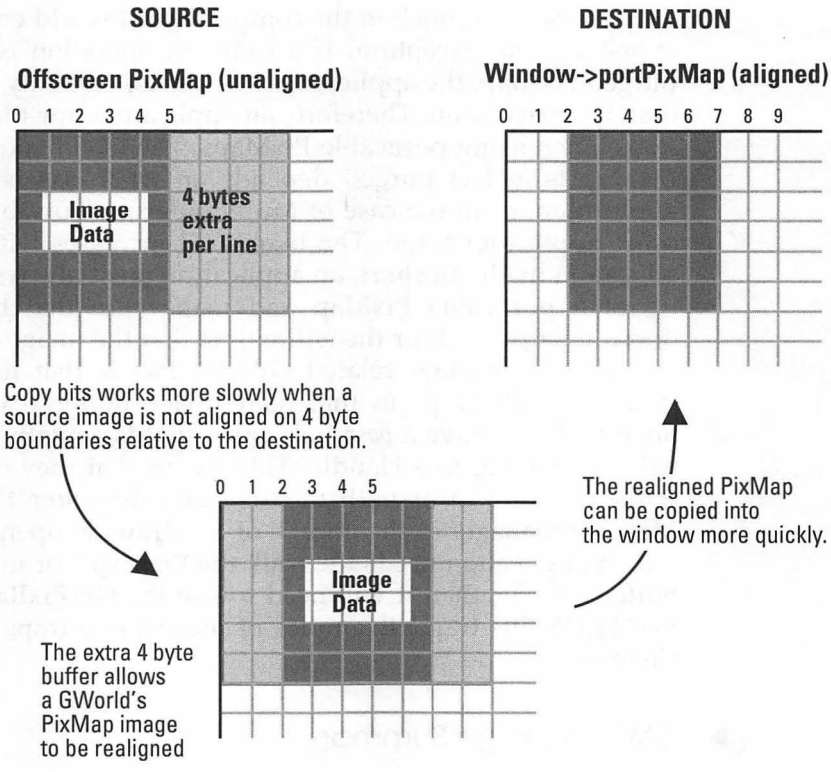


Figure 4-4. Address alignment

Purgeable and Locked PixMaps

During the execution of an image processing application it often becomes necessary to use temporary image buffers. That's all fine and dandy, but we already know what memory hogs pixel images can be, so the use of temporary image buffers can be somewhat dicey: They must be disposed of as soon as they're no longer needed — you just can't leave them around.

This is the same problem encountered with the Resource Manager, and like resources, GWorld PixMaps can be made purgeable. In fact, an application can allocate temporary PixMaps "permanently" by simply making them purgeable. Additionally, they can be made to live in MultiFinder memory.

Of course, not much in the computational world comes for free, and memory is no exception. If a drawing operation is attempted on a purged PixMap, the application will almost certainly die, either immediately or real soon. Therefore, an application must take extra precautions when using purgeable PixMaps. The likelihood that a purgeable PixMap is in fact purged depends on what else is going on in the application, or in the case of MultiFinder memory-based PixMaps, in other applications, too. The likelihood increases with every memory allocation made. In short, an application must always check the validity of a purgeable PixMap and make sure that the image buffer doesn't get purged for the entire time that the image itself is required.

Another important related GWorld fact is that its PixMaps differ from other PixMaps in that their image buffers can be relocatable. Such PixMaps have a *baseAddr* value that is actually a master pointer value belonging to a Handle. This means that they need to be locked while drawing in or with them so that they aren't relocated by the Memory Manager in the middle of the drawing operation itself.

If an application needs to "walk the PixMap," or to access the image buffer directly, then it will need to use the **GetPixBaseAddress**() and **PixMap32Bit**() traps. Examples of using these traps are presented in Chapter 9.

▶ GWorld Usage Summary

With a slight excursion into the world of GWorld programming behind you, it should be fairly obvious that GWorlds offer a masterful combination of functional richness and flexibility in a relatively simple and compact framework. There are few areas of the Macintosh Toolbox that provide such comprehensive functionality without burdening you with needing to become acquainted with numerous data structures and traps.

▶ Application Display Control

▶ Rules about Changing GDevice Settings

Version 1.2 of 32-bit QuickDraw added the ability of applications to control the pixel depth and color/grayscale characteristics of screen (public) GDevices. But just because a monitor can be reconfigured on the fly doesn't mean that applications should do so indiscriminately. The purpose of this facility is to allow applications to provide an

alternative means of control for the user (not the application), rather than force them to use the Monitors Control Panel Device. There should be no need for applications to make these changes without explicit instruction from the user.

▶ Using HasDepth() and SetDepth()

32-bit QuickDraw provides two traps for controlling display characteristics:

```
pascal short HasDepth(
  GDHandle    gdh,
  short       pixelDepth,
  short       attributeFlags,
  long        modeFlags );

pascal OSErr SetDepth(
  GDHandle    gdh,
  short       pixelDepth,
  short       attributeFlags,
  long        modeFlags );
```

These traps facilitate the testing and setting of a GDevice's pixel depth and color/grayscale characteristics. The arguments hold the same meaning for both traps. The "gdh" is the display GDevice to reconfigure; private GDevices can be set to any valid pixel depth or color map. The "pixelDepth" should be an appropriate pixel depth. The "attributeFlags" are the same as those passed to a NewGDevice() call. As of Color QuickDraw version 1.2, only the "gdDevType" flag bit has a defined meaning, so until future functionality is added to these traps, pass only that flag bit for this argument. The "modeFlags" should be zero if grayscale, or 1 if color.

When **HasDepth**() is called, it queries the GDevice's driver to determine whether the depth/color request can be handled, returning the corresponding mode value if the request is valid, or zero if the request cannot be handled. The mode value is insignificant, as **SetDepth**() accepts the same depth and color mode arguments.

When **SetDepth**() is called, it attempts to perform the depth and/or color mode change. It returns "noErr" (zero) if successful, or an appropriate error value, typically a "cDepthErr" (–157) or "cParmErr" (–158). If the pixel depth and color modes are already in effect, no action is taken and "noErr" is returned.

The basic calling sequence is shown in Listing 4-3.

Listing 4-3. Example of using HasDepth() and SetDepth()

```
void DepthCharger( void )
{
GDHandle    theMainGD;
short       newPixDepth, newColorMode, depthModeID;
OSErr       errStat;

  /* let's try to set the main screen to 8-bit color */
  theMainGD = GetMainDevice();
  newPixDepth = 8;
  newColorMode = 1;

  depthModeID = HasDepth(
    theMainGD, newPixDepth, 1 << gdDevType, newColorMode );

  /* if HasDepth() says OK, then update the device as requested */
  /* make sure that menu highlighting is off */
  if (depthModeID != 0)
    errStat = SetDepth(
      theMainGD, newPixDepth, 1 << gdDevType, newColorMode );
}
```

There are few video cards that can handle all valid pixel depths and color/grayscale combinations, making the error checking all the more important.

A quirky side effect of **SetDepth()** is that any menu bar highlighting will get messed up. If the depth change is the result of a menu command, then the application should unhighlight the menu bar via a **HiliteMenu**(0) call.

▶ The Palette Manager

The indexed display model presents a number of problems when taken in context to the Desktop's multi-window, multi-application environment. The most important issue to resolve is the assignment of displayable colors to windows. Suppose that we have four applications running under MultiFinder, with each application displaying four windows, each of which contains a picture with 64 unique colors, as shown in Figure 4-5. Any time the user has more than one window open that contains graphics, chances are that the total number of colors requested by those windows will exceed the 256 possible in an 8-bit system (by far the most common color environment).

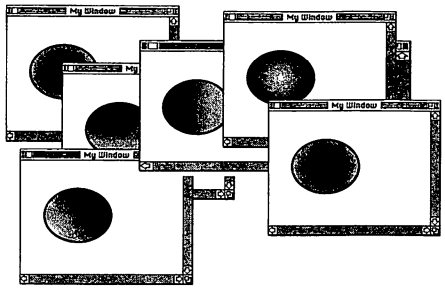


Figure 4-5. Windows with too many color requests

Using simple arithmetic, we can see that 4 × 4 × 64 equals 1024 colors, which is 768 more than 256, or four times the number of available colors on any indexed GDevice. This is called *color contention*, which is a classic example of computer processes (windows) battling over limited hardware resources (hardware color entries). So which windows get which colors? The issues related to this dilemma are solved through palettes, which are managed through the Palette Manager.

The Palette Manager is a layer of software that lies between applications and GDevice ColorTables. Its primary job is to resolve color contention with a minimum of changes in the GDevice's ColorTable. Palettes are constructed similarly to CTables, with the addition of usage data for each color entry. Applications interact with the Palette Manager simply by determining the desired colors for a window or group of windows, creating an appropriate palette, and attaching it to that window or windows. From that point on, the Window and Palette Managers take responsibility for all management of the GDevice's color entries. The Palette Manager resolves contention for display GDevice's ColorTable entries through a set of well-defined entry usage and color matching algorithms.

The Palette Manager's role in the arbitration of color requests and resolution of color contention is like that of a librarian: Color entries can be checked out, used as needed, and returned to the shelves for other windows to use.

▶ Basic Palette Concepts

A palette's header are pmEntries, which simply indicate how many color entries the palette contains, and pmDataFields, which are reserved for future use by system software. Some definitions of the Palette

structure divide *pmDataFields* into *pmWindow, pmPrivate, pmDevices,* and *pmSeeds*. In either case, those 12 bytes should be left alone.

The *pmInfo* entries (the color specifications) differentiate a palette from a ColorTable. Each *pmInfo* entry describes not only the RGB color but the closeness to which that color needs to be matched when the palette is activated.

Windows and Color Requests

A palette is the means by which a window expresses the color that it wishes to have available to draw its contents accurately. The Palette Manager treats a palette as a list of color requests. Each window in the system has an associated palette. If the application fails to assign one, then the Palette Manager uses a default palette.

The requests presented by the frontmost window's palette of the foreground application takes precedence over all other color requests. Every time a new window or application is brought to the front or foreground, the Window Manager informs the Palette Manager. The Palette Manager, in turn, determines whether a change in the GDevice's ColorTable is necessary to fulfill the color request made by that window's palette. The Palette Manager makes any necessary changes in the appropriate GDevice's ColorTable to accommodate the window's palette so that the window's contents appear to the user as the application intends.

Think of a palette as a list of requests; each entry in a palette is a color that your window wants to be able to use. When a window's palette is installed in the ColorTable, each entry in the palette will occupy, or be accounted for by, an entry in the ColorTable *within the rules of that palette entry's usage attribute*. See the "Entry usage" section on the Palette Manager that follows for more details. When another window's palette is installed in the ColorTable, if any entries were changed to accommodate the new palette, then the Color Manager assumes that entries that the window is using may have changed and it tells the application so by giving your window an update event. When, in response to the update event, you redraw the contents of your window color, matching takes place with the current (changed) ColorTable and new entries are used where appropriate.

Here's an example to illustrate what we mean by color environment changes causing update events. Suppose an application has a window which is drawn with a red background. The window's palette contains an entry with values RGB = (65535, 0, 0) and before drawing into the window the application calls **RGBBackColor**(&myRedBack-

ground) where myRedBackground.RGB = (65535, 0, 0). Let's say that when the application's window came to the front (and its palette was installed in the GDevice's ColorTable), the background red color ended up in the ColorTable at index 12. Suppose that the application is also sharing the system with another application whose window has a palette that installs a pastel pink in the ColorTable at index 125 and the rest of the ColorTable is filled with shades of gray. When the second application's window comes to the front, as it probably would if it got clicked on by the user, its palette is installed in the GDevice's ColorTable, which, because at least one entry was changed, causes the application to receive an update event for its window. After the second application's palette was installed, the red background color was replaced in the ColorTable by some shade of gray and at that instant the background flashed from red to gray. The update event is the application's opportunity to adjust to the changing color environment. When the window's update routine tries to set its background color to red, Color QuickDraw can't oblige; the red color is no longer in the GDevice's ColorTable. So the Color Manager maps the request to the pastel pink that it finds at index 125 and the window's update routine drawing with a pink background. The justification, of course, is that pink is a better substitute for red than just trusting to luck and using whatever color replaced the color of choice.

The need for the Palette Manager grew out of the problems that arise when more than one window with different color needs try to display on the same indexed GDevice. The result is contention for the GDevice's ColorTable, and the Palette Manager was created to arbitrate the conflicts. The Palette Manager is most interesting when windows are being displayed on an indexed monitor so the discussion that follows concentrates on indexed pixels being displayed on indexed GDevices. The use of direct pixels is mentioned here and there to keep the rules and facts complete and accurate.

The Palette Manager is started up the first time that **InitWindows()** is called after the Macintosh is turned on. It runs independently of windows, which can come and go, and also independently of applications, which can start and quit while the Palette Manager remains active. Contention for GDevice CLUT entries comes from windows in different applications as well as from windows of the same application.

Entry Stealing

If the GDevice has enough unused color entries, or if the current entry values meet the palette's requirements, then the Palette Manager changes entries in the GDevice's ColorTable. When there are not

enough available unused entries, the Palette Manager changes entries that belong to another window's palette. This process is known as *entry stealing*.

Color Updates. A window whose palette has entries stolen is said to experience a color environment change; if the window's palette was set with a **SetPalette**() or **NSetPalette**() call whose "cUpdates" argument requested updates, then the Palette Manager will invalidate the contents of that window. The ensuing update event is called a *color update*. Color updates involve the redrawing of entire windows and may happen to many windows simultaneously. The resultant flash associated with numerous windows redrawing themselves can be quite disconcerting. The frequency with which color updates occur is determined by four factors:

- the number of different palettes across all applications;
- palette size, or the number of colors;
- palette "flexibility," implied by the palette entry usages; and
- the number of front window changes between windows of differing palettes.

It should be evident that the user has the most control over the color environment. Running applications that demand large numbers of displayable colors tends to increase color updates for all windows in all applications. The applications that are the most "color-aggressive" are games and other graphics-intensive programs, particularly paint programs. Users who frequently run these kinds of applications in addition to other applications will probably experience both eyestrain and more than an average number of color updates.

Assignment of Color Updates. The Apple documentation implies that the Palette Manager is making informed decisions about which windows do and do not need color updates. But, in fact, any window on a GDevice whose CLUT has been changed gets an update event. Even windows with explicit palettes — windows that, by design, don't care which color values are at which entries in the GDevice's ColorTable — get notified of color environment changes. The other unfortunate thing about color updates is that a color update cannot be distinguished from a "normal" (window order change) update.

Important ▶

Tech Note #120, Building an Offscreen PixMap, suggests that color updates can be recognized by keeping track of the GDevice ColorTable's *ctSeed* value. Unfortunately, this is an unreliable indicator, as a number of other processes cause the seed value to change without causing a "real" color update.

Palette Data Structure

Listing 4.4 shows the definition of the Palette data structure.

Listing 4.4. Palette data structure

```
typedef struct
{
   RGBColor ciRGB;
   short    ciUsage;
   short    ciTolerance;
   short    ciFlags;
   long     ciPrivate;
}
ColorInfo;

typedef struct
{
   short       pmEntries;
   short       pmDataFields[6];
   ColorInfo   pmInfo[1];
}
Palette, *PalettePtr, **PaletteHandle;
```

This data structure is presented for informational purposes only. All palette data access and modification should be performed using the appropriate Palette Manager traps rather than accessing a palette variable directly. The exception to this rule is that the *pmEntries* field may be read in order to determine the number of color entries. The other fields are used by the Palette Manager to maintain private status information particular to each palette.

Entry Color

Recall the ColorTable structure from Chapter 3. It consists of a simple header followed by an array of color definitions. A palette structure is

very similar to a ColorTable with the addition of fields in each color specification to describe the intended usage of the entry and its associated color value. Each palette entry has an associated RGB value.

The Palette Manager considers white-and-black entries to be special, provided they are the first and last palette entries, respectively. These entries are fixed and cannot be stolen. The Palette Manager expects to see those entry colors in their rightful places. A palette may have any number of additional black and/or white entries. ColorTables should also be built according to this rule.

Entry Usage

Every palette entry has a usage value that defines the degree to which that color request should be satisfied:

- **Courteous** — Courteous colors simply reserve a place in the palette for a particular color. When drawing with courteous colors, the best available color match is used, regardless of how close the available color matches it.
- **Explicit** — An explicit usage directs the Palette Manager to provide the color currently available at that index, regardless of what that color may be; the entry's color is ignored. Explicit colors are as innocuous as courteous color in that neither usage will force the Palette Manager to cause a color update.
- **Tolerant** — Tolerant colors are perhaps the most useful to the most applications. When tolerant colors are added to the ColorTable, the available entries are first checked to see if there is a color already in the table that is a close enough match to the requested color. The difference between two colors is defined as the maximum of the three differences between the colors' respective components. The tolerance value specified in the palette entry for a tolerant color tells the Palette Manager how close is close enough for a match. A tolerance of zero means that the entry's color must match exactly. Such entries may cause color environment changes.
- **Animated** — An animating palette entry reserves an index in the GDevice's ColorTable for exclusive use of all windows associated with that palette. This allows applications to make frequent, rapid color changes to the contents of their windows, without having to redraw them. This technique is often referred to as *color animation*, and is employed by a number of game applications and "lava-light" screen savers.

The animated entry usage is the most demanding of the color environments; animated entries are reserved by that palette for as long as it exists.

- **Grayscale-inhibited and color-inhibited** — The inhibited usage allows selective control of which display types attempt color request matching. A palette containing color entry values by definition cannot match exactly or even very closely on a grayscale display. Such palettes are good candidates for inhibited entries.

Entry Tolerance

Every palette entry has an associated tolerance value. However, this value is only pertinent to tolerant entries. It defines the allowable distance between an entry's color value and a prospective color value match.

▶ Important Palette Manager Traps

The following traps make up the core of the Palette Manager application interface.

Creating and Disposing Palettes

```
PaletteHandle NewPalette(
  short         numEntries,
  CTabHandle    srcColors,
  short         srcUsage,
  short         srcTolerance );

PaletteHandle GetNewPalette(
  short         paletteID );
```

These two traps create palettes. **NewPalette**() builds a palette from the first "numEntries" entries of an existing ColorTable "srcColors," setting each entry's usage and tolerance value to "srcUsage" and "srcTolerance," respectively.

GetNewPalette() creates a palette from a 'pltt' resource with an ID "paletteID." Because palettes must be "registered" and initialized by the Palette Manager, the application must load resource-based palettes using this trap rather than **GetResource**(). Palettes are automatically created and assigned to windows created by **GetNewCWindow**() if both a 'pltt' and 'WIND' resource exist with the specified resource IDs.

```
void DisposePalette(
  PaletteHandle thePalette );
```

This trap disposes of all memory associated with a palette and all references to it, regardless of which palette trap it was originally created with. Disposing of a palette must be performed by this trap rather than **DisposHandle()**, otherwise the Palette Manager gets confused when activating the window that it used to belong to.

Palette Assignment

```
void SetPalette(
  WindowPtr       destWindow,
  PaletteHandle   srcPalette,
  Boolean         cUpdates );

void NSetPalette(
  WindowPtr       destWindow,
  PaletteHandle   srcPalette,
  short           pmUpdateFlags );
```

These traps associate "srcPalette" with "destWindow." Actually, these two traps are the same traps with an enhanced meaning for the last argument, which indicates under what circumstances "destWindow" will receive color updates. **NSetPalette()** is a slightly more recent version of **SetPalette()**; its "pmUpdates" argument takes one of the following constant values: "front window only," "background window only," "always," or "never." **SetPalette()**'s "cUpdate" argument allows only for always (TRUE) or never (FALSE). A "destWindow" value of –1 directs the Palette Manager to make "srcPalette" the default palette.

Pen Color Assignment

```
void PmForeColor(
  short    palIndex );

void PmBackColor(
  short    palIndex );
```

These traps operate similarly to the their absolute color counterparts **RGBForeColor()** and **RGBBackColor()**, except that they take palette indexes rather than RGB values as arguments. The index values are forced within the range of actual palette entries via modulo math:

index = index modulo pmEntries

Palette Activation

```
void ActivatePalette(
  PaletteHandle   thePalette );
```

This trap instructs the Palette Manager to re-register "thePalette." This trap is called by the Window Manager whenever a new window is made frontmost so that applications won't have to call **ActivatePalette()** in response to activate events. This trap should be called after changes to one or more of a palette's entries (color or usage) are performed.

Palette and CTable Conversions

```
void CTab2Palette(
  CTabHandle      srcCTab,
  PaletteHandle   destPalette,
  short           usage,
  short           tolerance );

void Palette2CTab(
  PaletteHandle   srcPalette,
  CTabHandle      destCTab );
```

These traps allow simple conversion between palettes and ColorTables. Each Handle-based argument must be a valid Handle.

Palette Entry Control

```
void GetEntryColor(
  PaletteHandle   thePalette,
  short           destEntry,
  RGBColor        *srcRGB );

void SetEntryColor(
  PaletteHandle   thePalette,
  short           destEntry,
  RGBColor        *srcRGB );
```

These traps allow read/write access to palette entry RGB color values and should be used in lieu of direct references to palette data. If the entry is out of range, then no action is taken and/or no arguments are modified.

```
void GetEntryUsage(
  PaletteHandle   thePalette,
```

```
  short           destEntry,
  short           *srcUsage );

void SetEntryUsage(
  PaletteHandle   thePalette,
  short           destEntry,
  short           srcUsage );

void GetEntryTolerance(
  PaletteHandle   thePalette,
  short           destEntry,
  short           *srcUsage );

void SetEntryTolerance(
  PaletteHandle   thePalette,
  short           destEntry,
  short           srcUsage );
```

These traps allow read/write access to palette entry usage and tolerance values and should be used in lieu of direct references to Palette data. If the entry is out of range, then no action is taken and no arguments are modified.

▶ The Dark Past of the Palette Manager

The Palette Manager has been with Macintosh since the introduction of Color QuickDraw; however, earlier versions had a number of critical limitations:

- Palettes couldn't assign a color to a specific entry, making it difficult to display and manipulate pixel data whose relative values were significant, such as false color images.
- Color animation was limited. It wasn't possible to animate the contents of more than one window at a time, making it difficult to perform operations such as contrast adjustment.
- It was unable to define default palettes.
- It couldn't share palettes — one palette per window, one window per palette.
- It couldn't use **CopyBits**() to transfer animating pixel values.

These limitations induced many applications to program around the Palette Manager by calling the Color Manager to change display

GDevice ColorTables directly. The Palette Manager has been enhanced considerably since its inception; all the limitations listed here have been substantially addressed.

Important ▶

If you're developing an application that requires unblemished palette functionality, then it will require version 1.2 of Color QuickDraw, which contains the earliest version of the Palette Manager with the enhancements mentioned here.

The Palette Manager provides an application with enough color environment control, but it doesn't burden the application with the task of maintaining the color states of windows in other applications over multiple public GDevices. The Palette Manager sits above the Color Manager and the Graphic Devices Manager, making calls to each to manage palettes across multiple GDevices.

▶ Using Palettes

Palettes are extremely crucial to effective indexed color environment management. The basic palette techniques are quite simple: An application can be ecologically courteous and still appear in its requested colors, provided that all executing applications follow Apple's palette guidelines. With a bit of care, applications can exercise a high degree of sophisticated color display control without causing frequent color updates for other applications.

Associating Palettes with Windows

To associate a palette with a window, simply call **NSetPalette**() with the window, the palette, and the constant indicating which type of color updates the window should receive. Most applications will call for the value "pmAllUpdates" for the last argument.

If a window is created via **GetNewCWindow**() and there is an available, corresponding 'pltt' resource with the same ID as the window, then that palette will be allocated and assigned to that window automatically.

There is also a default palette, which is used for all color windows that don't have a palette. This palette is the system palette on application startup, but it can be changed by calling **SetPalette**() or **NSetPalette**() and passing –1 for the WindowPtr argument. Ironically, the default

palette cannot be obtained with **GetPalette**(); calling this trap with a WindowPtr argument that references a window that has no palette simply returns NULL.

Associating Palettes with CGrafPorts

Earlier in the chapter we stated that palettes were only associated with windows. In reality, palettes can be associated with any CGrafPort. Associating a palette with an offscreen CGrafPort has no effect on its appearance because it is offscreen. However, it does allow pen color assignments via the **PmForeColor**() and **PmBackColor**() traps. Additionally, setting the second-highest bit of the *ctFlags* field of a CGrafPort's ColorTable directs QuickDraw to treat indexed pixel values as palette indexes rather than as ColorTable indexes. This allows animating pixel values to be transferred via **CopyBits**() unchanged, an otherwise impossible task due to the fact that animating entries are reserved and otherwise not matchable by **CopyBits**().

Choosing Usages and Tolerances

Generally speaking, it's best to keep palettes as flexible as possible. Use high-tolerance values for tolerant entries whenever possible. Minimize the number of animated and low-tolerance entries. Use courteous entries whenever possible. Palettes structured around these guidelines allow the Palette Manager to effectively distribute and share requested colors to the maximum number of applications with a minimum of entry stealing and color updates.

Compound Usages. Under Color QuickDraw, (versions 1.0 and later) the Palette Manager supports certain combinations of usage values. These combinations allow the exact specification of which color gets assigned to which entry, an important feature necessary for the accurate display of scans and other false-color images. Additionally, the inhibitory usages may be added to any usage value in order to prevent color requests from being generated on indexed GDevices of specific pixel depths and color capabilities.

Tolerant/explicit usage values direct the Palette Manager to install the requested color at the requested entry. This is particularly useful for displaying false-color images. The recipe for displaying false-color images while avoiding undesired color matching is as follows:

1. Construct a palette to match the image's desired colors and install it into the destination window using **NSetPalette**().

2. Set bit 14 ($4000 hexadecimal) in the *ctFlags* field of the image's offscreen PixMap's ColorTable. (See the "Associating Palettes with GrafPorts" section.)
3. Call **CopyBits**() to update the window as needed.

Setting the *ctFlags* bit directs **CopyBits**() to skip the color matching step that it would otherwise perform. The rates achievable by this method allow for series of images to be displayed close enough to the 30 images per second video rate, depending on the width and height of the displayed images.

Explicit/animated usage values are the most inflexible and demanding of all usage combinations, translating into color requests at specific entries that cannot be matched by any other palette. This usage can be very useful when a large number of related pixel values are to be animated all at once. However, it's best to limit this usage for only as long as it's absolutely needed because it shuts out all other applications from using those affected entries and related indexed pixel values.

As shown in Table 4-1, the request inhibitors are useful for suppressing color requests that are inappropriate for particular GDevice settings.

Table 4-1. Palette entry inhibitory constants

Usage flag	*Inhibit requests on GDevices*
pmInhibitG2	2-bit grayscale
pmInhibitC2	2-bit color
pmInhibitG4	4-bit grayscale
pmInhibitC4	4-bit color
pmInhibitG8	8-bit grayscale
pmInhibitC8	8-bit color

Palettes and Direct GDevices

As established in the previous chapter, color arbitration is non-existent on direct GDevices. In general terms this implies a relaxation of rules and restrictions on palettes. The effect on palette entry usages and tolerances is as follows:

- **Courteous, tolerant** — Equivalent to intolerant (zero tolerance value) entries. An entry's color request is always fulfilled.

- **Explicit, animated, inhibited** — Has no meaning, since direct GDevices don't have entries. GDevice entries and color values cannot be reserved by applications.

The **PmForeColor()** and **PmBackColor()** traps become functionally equivalent to **RGBForeColor()** and **RGBBackColor()** calls, using the palette entry's color value, as shown in the following example:

```
PmForeColor( 4 ); ==> RGBForeColor( &((**pltt).pmInfo[4].rgb) );
```

Creating and Using 'pltt' Resources

Like all other user interface objects, palettes can be resource-based. Like windows and dialogs, palette resources are template-based: Multiple copies of a palette can be generated from a single palette resource. Palettes may be saved as resources simply by writing the contents of a PaletteHandle to a resource file as a resource of type 'pltt'.

Using 'pltt' resources has some interesting implications, particularly regarding entry usage and tolerance. Generally speaking, an application decides on entry tolerance and usage values based on the state of the application's windows and the hardware limitations of the available display GDevices. This being the case, the usage and tolerance values found in the resource are best ignored, except when the application can be sure of the resource's origin, for example, when it created the palette. A palette stripped of usage and tolerance is a ColorTable, plain and simple. An application can easily load a ColorTable from a 'clut' resource, convert it into a palette, and assign new tolerance and usage values, as shown in Listing 4-5.

Listing 4-5. Building a palette from a CTabHandle

```
PaletteHandle BuildClutPalette(
  short  clutResIndex )
{
PaletteHandle   curPalette;
CTabHandle      curClut;

  /* load a 'clut' resource and copy its data into a */
  /* newly allocated CTabHandle */
  curClut = GetCTable( clutResIndex );
  if (curClut == NULL)
    return NULL;
```

```
  /* build a new palette, assigning usage and tolerance */
  /* values for all entries */
  curPalette =
    NewPalette( curClut,
                (**curClut).ctSize + 1,
                pmTolerant, 0x2000 );

  /* don't need this anymore */
  DisposCTable( curClut );

  return curPalette;
}
```

Palette Animation Example

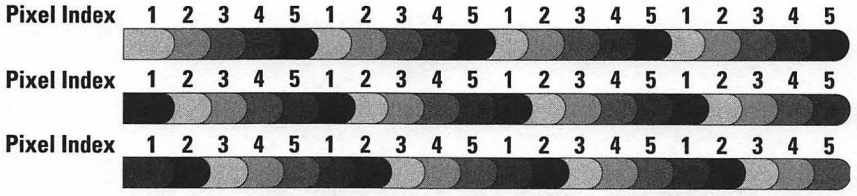
Palette animation is a wonderful way to generate dynamic, kinetic graphic effects with a minimum of programming fuss while incurring the execution workload of "real" drawn animation.

The main issues to consider when animating a palette are as follows:

- **Which entries should you animate?** — The most effective palette animation techniques involve reserving a small number of contiguous entries at one end or the other of the GDevice's ColorTable. The number of entries required is equal to the number of different colors for each animated graphic times the number of animation phases to cycle through. Figure 4-6 shows a possible animation sequence.

  The animated graphic has five colors and cycles through four phases, one for each drawing; it therefore would require 20 animating entries.
- **How should the animating colors be maintained?** — The **AnimateEntry**() trap takes a single RGB color. For animating subranges of entries, the **AnimatePalette**() trap is more appropriate: It takes a ColorTable and a number of offset value arguments, allowing a whole range of palette entries to be animated at once. This readily facilitates *color cycling*, the process of rotating through a small fixed set of RGBColor values.
- **How often should animating take place?** — That depends. The cycle time is usually regulated by using the **TickCount**() trap to divide animation intervals into ⅙₀ of a second chunks. A palette animation cycle can shift a small number of entries frequently to effect smooth color transitions, or it can shift a large number of entries less often to produce dramatic changes. In Figure 4-6, the

drawing could be made to move by cycling through a ColorTable that contained five unique colors in adjacent entries, with the remaining entry colors matching the background. If the palette was animated by cycling it in increments of five entries, then the drawing would move left to right on each animation phase.



Motion is achieved by animating five consecutive entries.

Figure 4-6. Drawing with animated entries

- **When is animation permissible?** — The only windows that can animate their palette are those that share their palette with the frontmost window in the current foreground application. All other window animation requests will be ignored. However, the available entries requested will be reserved by that window and palette and will become unavailable to other windows. A courteous animating window will give up some of its entries, especially if it reserves a large number of entries, say, more than 64.

In Listing 4-6, the CColorCycler class is presented. It attempts to address the concerns just described as simply as possible.

Listing 4-6. CColorCycler class

```
typedef enum
{
  COLORANIMOPT_none       = 0,
  COLORANIMOPT_some       = 1
}
ColorAnimation_OptFlags;


class ColorCycler : indirect
{
  Ptr             nextObject;
  long            objectID;
```

```
  WindowPtr       plttWindow;
  CTabHandle      animCTabHdl;     /* colors used in animation */
  Point           cycleRange;      /* index range for animation */
  short           cycleIncrement;  /* # of indexes to skip each */
                                   /* time */
  short           animationMode;   /* special animation tricks */
                                   /* enabled? */
  short           curPlttEntry;    /* palette entry being */
                                   /* animated */
  unsigned long   lastAnimTicks;   /* time of last palette */
                                   /* shift */
  unsigned long   animTickDelay;   /* # of ticks to wait */
                                   /* between shifts */


  void Init( void );

  void Dispose( void );

  void SetWindow(                  /* associate window with object */
    WindowPtr   thePlttWindow );

  OSErr PaletteChanged( void ); /* update the internal ColorTable */

  Boolean IsActive( void );        /* check if cycling */

  void SetCycling(                 /* turn cycling on or off */
    Boolean   makeActive );

  void Advance( void );            /* advance one cycle */
};



void ColorCycler::Init( void )
{
  this->nextObject = NULL;
  this->objectID = 0L;

  this->plttWindow = NULL;
  this->animCTabHdl = NULL;
  SetPt( &(this->cycleRange), 1, 254 );
  this->cycleIncrement = 1;
  this->animationMode = COLORANIMOPT_none;
  this->curPlttEntry = 1;
```

```
    this->lastAnimTicks = 0L;
    this->animTickDelay = 1L;
}


void ColorCycler::Dispose( void )
{
    if (this->animCTabHdl != NULL)
        DisposCTable( this->animCTabHdl );
}


void ColorCycler::SetWindow(
    WindowPtr   thePlttWindow )
{
    this->plttWindow = thePlttWindow;
    (void)this->PaletteChanged();
}


OSErr ColorCycler::PaletteChanged( void )
{
PaletteHandle   curPlttHdl;
CTabHandle      newCTabH;
Size            newCTSize;
OSErr           errStat;

    if (this->plttWindow == NULL)
        return (OSErr)(-1);

    curPlttHdl = GetPalette( this->plttWindow );
    if (curPlttHdl == NULL)
        return (OSErr)(-1);

    newCTabH = this->animCTabHdl;
    if (newCTabH == NULL)
    {
        /* allocate a new empty ColorTable */
        newCTSize = sizeof(ColorTable) - sizeof(CSpecArray);
        newCTabH = (CTabHandle)NewHandleClear( newCTSize );
        errStat = MemError();
        if (errStat != noErr)
            return errStat;
    }
    Palette2CTab( curPlttHdl, newCTabH );
    this->animCTabHdl = newCTabH;
```

```
  return errStat;
}


Boolean ColorCycler::IsActive( void )
{
  return (this->animationMode != COLORANIMOPT_none);
}


/*
** "unrotate" the palette and reset cycling variables
*/
void ColorCycler::SetCycling(
  Boolean  makeActive )
{
  if (makeActive == this->IsActive())
    RETURN;

  this->curPlttEntry = this->cycleRange.h;
  if (makeActive)
    this->animationMode = COLORANIMOPT_some;
  else
  {
    this->animationMode = COLORANIMOPT_none;
    AnimatePalette(
      this->plttWindow, this->animCTabHdl, 1, 1,
      (**(this->animCTabHdl)).ctSize - 1 );
  }
}


void ColorCycler::Advance( void )
{
PaletteHandle thePlttHdl;
RGBColor      newRGBvalue;
WindowPtr     thePaletteWindow;
short         ctEntries, firstEntry, lastEntry, animatingEntries;

  if ((this->animationMode <= COLORANIMOPT_none) ||
      (this->animCTabHdl == NULL))
    RETURN;

  if (this->animTickDelay > 0)
    if (this->lastAnimTicks + this->animTickDelay > TickCount())
      RETURN;
```

```
  if (CheckKeyDown( KEYBIT_caps_lock ))
    RETURN;

  if (this->plttWindow == NULL)
    RETURN;

  thePlttHdl = GetPalette( this->plttWindow );
  if (thePlttHdl == NULL)
    RETURN;

  /* calculate the appropriate maximum palette animation range */
  ctEntries = (**(this->animCTabHdl)).ctSize + 1;
#ifdef _PALETTE_BW_PROTECT_
  firstEntry = 1;
  lastEntry = ctEntries - 1;
#else
  firstEntry = 0;
  lastEntry = ctEntries;
#endif _PALETTE_BW_PROTECT_

  /* restrict it further by the user's ramp range preferences */
  if ((firstEntry < this->cycleRange.h) &&
       (this->cycleRange.h <= lastEntry))
    firstEntry = this->cycleRange.h;
  if ((lastEntry > this->cycleRange.v) &&
       (this->cycleRange.v >= firstEntry))
    lastEntry = this->cycleRange.v;
  animatingEntries = lastEntry - firstEntry;
  if (animatingEntries <= 0)
    RETURN;

  if (this->curPlttEntry < this->cycleRange.h)
    this->curPlttEntry = this->cycleRange.h;
  else if (this->curPlttEntry > this->cycleRange.v)
    this->curPlttEntry = this->cycleRange.v;

  /* perform palette animation, in two pieces */
  AnimatePalette(
    this->plttWindow, this->animCTabHdl,
    this->curPlttEntry, firstEntry,
    (lastEntry + 1) - this->curPlttEntry );
  AnimatePalette(
    this->plttWindow, this->animCTabHdl, firstEntry,
    firstEntry + (lastEntry + 1 - this->curPlttEntry),
    this->curPlttEntry - firstEntry );
```

```
    /* advance the point of rotation */
    this->curPlttEntry += this->cycleIncrement;
    while (this->curPlttEntry < firstEntry)
      this->curPlttEntry =
        lastEntry - ((firstEntry - this->curPlttEntry) %
        animatingEntries);
    if (this->curPlttEntry > lastEntry)
      this->curPlttEntry =
        ((this->curPlttEntry - lastEntry) % animatingEntries) +
        firstEntry;

    /* update the animation counter */
    this->lastAnimTicks = TickCount();
}
```

▶ Palette Review

Palettes are an extremely clever and useful addition to the graphics programmer's tool kit. They provide a simple and reliable way for windows across all "courteous" applications to painlessly share GDevice color entries with minimal disruption to the display and execution processes of other windows and applications. Palettes can be harnessed for precise control of image display characteristics and "sprite" animation graphics at the expense of other applications that get their entries stolen when these operations are invoked in the frontmost window.

▶ Specialized ColorTables

As noted in Chapter 3, every ColorTable has an ID number that is stored in its *ctSeed* field. The significance of this value depends on the range of numbers that it falls within:

- **0 to 127** — These values are reserved by the Graphics Toolbox for ColorTables that describe standard ColorTables.
- **1 to 24** — These IDs are reserved for ColorTables for the default color monitor display at that pixel depth. For example, the default ColorTable ID for an 8-bit color display is 8.
- **33 to 63** — These IDs are reserved for ColorTables for the default grayscale monitor display at that pixel depth, plus 32. For ex-

ample, the default ColorTable for an 8-bit grayscale display is 8 + 32 = 40.

- **64 to 88** — These IDs are reserved for ColorTables for the default color monitor display, including the current highlight color, at that pixel depth, plus 64. For example, the default ColorTable for a 4-bit color display, including highlight color, is 4 + 64 = 68. The highlight color is discussed later in this chapter.
- **127** — This is the ID of the Finder's Classic ColorTable, which is visible as items in its Colors menu.
- **128 to 1023** — These IDs are reserved for application ColorTables.
- **1023 and greater** — These IDs are reserved by the Color Manager. The trap **GetCTSeed**() returns a unique value in this range. In order to guarantee the uniqueness of seed values, applications should refrain from using these values for their own ColorTable IDs.

ColorTables can be created by calling the Color Manager's **GetCTable**() trap with an integer value designating a valid ColorTable. If the ID indicates a valid default ColorTable, then the ColorTable is allocated and the appropriate values are generated by the Color Manager. If the ID is in the application ID range, then the trap looks for a 'clut' resource with that ID. If it finds one, then it fills the entry values from the resource. If the ID doesn't indicate a valid ColorTable, or there isn't enough free heap space to create one, NULL is returned.

To create a CTable from scratch, the following steps should be taken:

1. Call **NewHandle**() to allocate enough memory to hold the ColorTable and all entry data.

   size (colortable)
   = size(ctSeed + ctSize + ctFlags) + # entries × size(ColorSpec)
   = 8 + (# of entries × 8) bytes

2. Assign values for the *ctFlags, ctSeed, and ctSize.*

   ctFlags =
   $0000 for CTables in PixMaps
   $4000 for CTables in CGrafPort PixMaps with a palette
   $8000 for CTables in GDevices

   ctSeed =
   value from **GetCTSeed**()

ctSize =
number of entries minus one

3. Assign color values to ColorTable entries.

There are routines to perform simple ColorTable creation and entry assignment in CTabUtils, which can be found in the Source Code Appendix.

▶ GDevice ColorTables

We are already familiar with this ColorTable, and with the fact that it's managed by Color Manager traps rather than direct modification. An indexed GDevice ColorTable requires as many entries as there are possible color values; more entries are insignificant, fewer entries cause the color value of higher pixel index values to be calculated as a weighted average of the foreground and background pen colors. Each entry contains the RGB color value to display for the corresponding pixel index value. The entry's index value is inferred by its position in the ColorTable rather than the *value* field's value! The entry's *value* field is used by the Color Manager to hold color matching information.

▶ User Interface ColorTables

In a Color QuickDraw machine, all of the familiar user interface objects may have associated ColorTables — menus, controls, dialogs, and windows. These ColorTables are used to hold color values for parts of the particular object, each entry position being associated with a specific part. For example, a control's ColorTable is defined as follows:

Entry 0: text color
Entry 1: frame color
Entry 2: background fill color

Each object's ColorTable has a unique index format, still each is a ColorTable with the same format as a PixMap's ColorTable.

As with palettes and windows, there are ColorTable resource types that correspond to the various user interface resource types. For example, the 'cctb' type is a control ColorTable, and a 'cctb' of ID 55 would be associated with a 'CNTL' resource of ID 55. Calling **GetNewControl**(55, someWindow) would create a control from the 'CNTL' resource #55 and then attach a ColorTable created from the 'cctb' resource

#55. Certain user interface ColorTables have one or two traps that are used to assign entry color values and associate ColorTables with interface objects, in very much the same fashion as palettes are bound to windows.

Additionally, some of these user interface objects allow for a default ColorTable; usually this is the one with a resource ID of zero.

Important ▶

An annoying quirk involving dialogs and palettes demands a mention here: If a dialog created via **GetNewDialog**() uses a palette, then that dialog also requires a corresponding 'dctb' (dialog ColorTable) resource of the same resource ID as the dialog itself, or the dialog will be created with a standard GrafPort and the palette and all related operations will have no effect. This resource may have zero or more entries.

▶ Summary

This chapter presented two display control paradigms: offscreen drawing and palette-based display control. The chapter presented a brief discussion of video hardware characteristics, along with the concept of flicker and some of its causes. The chapter introduced GWorld as the preferred vehicle of offscreen drawing, providing comprehensive functionality with reduced complexity over the methods introduced in Chapter 3. The chapter also covered palettes and their component concepts, along with some examples and strategies for their effective employment in applications. Augmenting the discussion in Chapter 3, this chapter discussed the use of special-purpose ColorTables.

5 ▶ Applied Graphics Techniques

Up to this point, we've presented the Macintosh Graphics Toolbox as merely a large collection of features and functionality. You have learned of many of the tools that QuickDraw provides; this chapter focuses on the application of the tools presented in the previous chapters. The chapter begins by discussing the foundations and implementation of an example "paintbox" application, ArtToy. The discussion moves into windows and the constraints they impose on graphics presentation, along with guidelines and techniques for using specific Window Manager and QuickDraw traps to effectively manage windows and their contents within the desktop environment.

The chapter concludes with the development and discussion of two groups of window support source code, the first a generalized window class, the other a GWorld-buffered window class, ideal for a raster-oriented paint application. These classes are the heart of ArtToy's window and paint document support code.

▶ Building a Paint Application — ArtToy

Many programmers regard access to concise and pertinent source code examples as invaluable to the development of their own software development efforts. In this chapter and the next, we present ArtToy, an application that provides both reusable source code and clear methods for organizing software components. The most illustrative pieces of code are included in these chapters; the complete ArtToy source code is on the disk only.

▶ Objectives and Goals

In the process of building a paint application, we hope that a large number of software architecture issues can be both illuminated and resolved. Most of these issues involve a choice between two or more legitimate solutions, each of which is appropriate under a particular set of circumstances. The cliché "There's more than one way to skin a cat" holds true for Macintosh graphics programming, since a number of seemingly similar tools are available for a specific purpose. For example, there are three different Toolbox traps that can be used to set the foreground pen color. Which one is right? It depends on the situation.

A number of Macintosh applications exist that have presented and defined certain features so successfully that they have become a requirement for all applications of that type. The paint bucket and the spray can are two examples of tools required for commercially viable paint applications. Many popular features are task specific; a few are truly general purpose. A magnifying glass could probably be useful in any graphics application. The ability to sketch an area of interest is equally important. ArtToy contains implementations of many popular paint tools, along with sketching routines that can be reused outside of the painting environment.

Another important aspect of successful applications development is modularity. ArtToy contains a number of "elemental" utility routines that not only augment the final application's reliability but also support the ongoing development of any complex application. We understand that you may not be particularly interested in writing a paint application, but it's both useful and important to understand Macintosh graphics within some sort of source code framework. We chose a paint application primarily because the implementation covers a large portion of Macintosh Graphics Toolbox functionality and technique.

▶ Software Tools Needed

The finished ArtToy application contains these functional components:

- document windows
- tool selection via tool windows
- sketch tool mouse tracking and document modification
- animated selection tools (lasso and marquee)
- Cut-Copy-Paste support
- printing support

- PICT file reading and writing routines
- menus and associated command handlers

We will develop ArtToy from the bottom up in terms of functionality. The following features will be elaborated on in this chapter:

- **Basic window handling** — Support for handling window creation and disposal, along with standard update and activate event handling.
- **Document window handling** — The combination of a window with a GWorld to present a pixel image. The GWorld shall be readily accessible to other functional software components, particularly image modification tools and file I/O operations.

Chapter 6 presents the development of these tool features:

- **Sketch tools** — A sketch tool consists of three main components: an interface selector, such as a tool palette window or menu option; the effect itself, involving a transformation of the data structures that make up the pixel image; and an interactive sketching procedure, which provides the user dynamic visual feedback in response to mouse actions within a document window.
- **Selection tools** — A selection tool has the same three components as a sketch tool, but is used to merely select an area within a document window, rather than to actually change the content of the document.

The other application components are outside of this book's scope, but can be found on the Source Code disk.

▶ Window Fundamentals

Windows are the foundation on which most, if not all, graphic user interfaces are built. It's assumed that you understand the basics of supporting windows within a Macintosh application, so there's no need to review those techniques. However, the Toolbox's window support strongly affects the visual display of graphics and the handling of user input. The management of window-based color requests was covered in the previous chapter; we continue to delve into other graphics display techniques, particularly those involving clipped graphics operations, coordinate systems conversions, and update event handling, as well as clipboard (cut-copy-paste) and printing techniques.

Windows are the primary vehicle for data display (output) and user-application interactions (input). Regarding the display aspects, a window is the point at which an application's graphics capabilities meet the display hardware and the constraints imposed by the limitations of the hardware's capabilities. For example, an application may be able to maintain a data model of a simple three-dimensional object such as a sphere, but in order to display it on a screen, the application must somehow "flatten" the sphere into a 2-D representation, hopefully, one that maintains some appearance of depth, either by clever shading or animation effects.

As discussed earlier, these constraints are imposed and regulated by the Color, Palette, Graphics Device, and Window Managers. A window's input aspects are regulated by the Event and Process Managers, which route keyboard and mouse input to applications and their windows via events.

The burden placed upon the application by the importance of windows should not be taken lightly. Developers who account for the dual input/output nature of windows early in the development cycle are usually rewarded with applications that are both maintainable and extensible; the failure to develop a solid foundation for handling the display- and event-processing issues generally leads to undesirable software complexity and the proliferation of special-case handling.

Many development systems and software libraries exist that are intended to organize and simplify this substantial task. We will attempt to derive the more important concepts and methods put forth by these packages.

▶ Graphic Foundations of Windows

A window is basically an on-screen GrafPort. The WindowRecord and CWindowRecord are the underlying data structures used for all windows, dialogs, and alerts, as shown in Listing 5-1.

Listing 5-1. CWindowRecord data structure

```
typedef struct
{
  CGrafPort   port;       /* for WindowRecords: type GrafPort */
  short       windowKind;
  char        visible;
  char        hilited;
  char        goAwayFlag;
  char        spareFlag;
```

```
    RgnHandle       strucRgn;
    RgnHandle       contRgn;
    RgnHandle       updateRgn;
    Handle          windowDefProc;
    Handle          dataHandle;
    StringHandle    titleHandle;
    short           titleWidth;
    ControlHandle   controlList;
    WindowRecord    *nextWindow;
    PicHandle       windowPic;
    long            refCon;
}
CWindowRecord, *CWindowPeek;

typedef             CGrafPtr CWindowPtr;
```

The functional features associated with a window are spatially organized, and described by a handful of regions contained within the WindowRecord and its GrafPort.

▶ Window Regions

The regions used to describe and maintain a window's appearance and functionality are described in the following sections.

Structure Region

The structure region (strucRgn) is the area of the window that "belongs" to the window. More often than not, windows are adorned with small controls placed along the window's perimeter within the strucRgn. Some of the more familiar controls are

- **Go away box** — For closing the window
- **Drag region** — For moving the window to another screen location
- **Grow icon** — For "fine-tuned" resizing
- **Zoom / unzoom box** — For "coarse" resizing
- **Title** — For identifying windows to the user (located inside the drag region)

This region is maintained jointly by the Window Manager and the WDEF (window definition proc) routine that renders the window, and should never be directly altered by an application. In actuality, the grow box resides within the content region.

Content Region

The content region is the maximum area of the window available for application's graphic presentation. This region should never be directly altered by an application. However, the content region may be altered by the Window Manager when applications call the following traps:

- **ShowWindow()**
- **HideWindow()**
- **ShowHide()**
- **SizeWindow()**
- **ZoomWindow()**

Visible Region

The visible region describes the area within the GrafPort available for drawing. All displayable graphic operations occur within the intersection of the current GrafPort's *visRgn* and *clipRgn*. An offscreen GrafPort's *visRgn* is typically assigned shortly after creating the port itself, and is used to describe the *maximum* drawing limits within that port. It may strike you as rather odd that an offscreen port would have any *visRgn* at all, given that it *is* offscreen. In contrast to offscreen ports, a visible window's *visRgn* is changed by the Window Manager in order to accurately describe a window's visible area. In the majority of circumstances, these changes are either the result of resizing the window (via grow or zoom actions), or changing the window ordering and covering or exposing all or part of a given window. This region should never be directly altered by an application. However, the visible region may be altered by the Window Manager when applications call the following traps:

- **ShowWindow()**
- **HideWindow()**
- **ShowHide()**
- **SizeWindow()**
- **ZoomWindow()**
- **MoveWindow()**
- **SelectWindow()**

A window's "visRgn" serves an additional purpose to the one belonging to an offscreen GrafPort. It defines the current visible limits, rather than simply the drawing limit, which will change if the user clicks in the window's grow or zoom controls.

Clipping Region

As discussed in Chapter 2, the clipping region describes the area that the application has deemed fit to draw within. This works in the same way as masking tape allows house painters to mask off, or protect, specific areas from being painted over when covering large areas with paint.

Update Region

The update region describes the area that has been determined to be invalid and that requires redrawing by the application. Invalidating a window's contents is usually the result of changing the window order or uncovering some or all of an obscured window. It may also be the result of an explicit call to the **InvalRect**() or **InvalRgn**() traps by application code, or a change in displayable color availability (as discussed in Chapter 4).

The relationship of these regions to a window is shown in Figures 5-1 and 5-2.

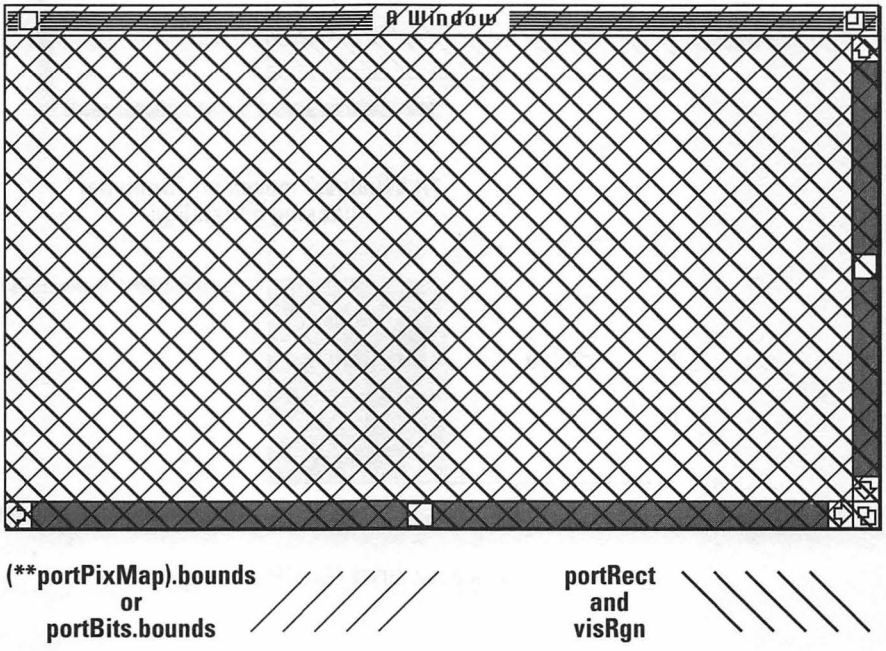


Figure 5-1. Window rectangles

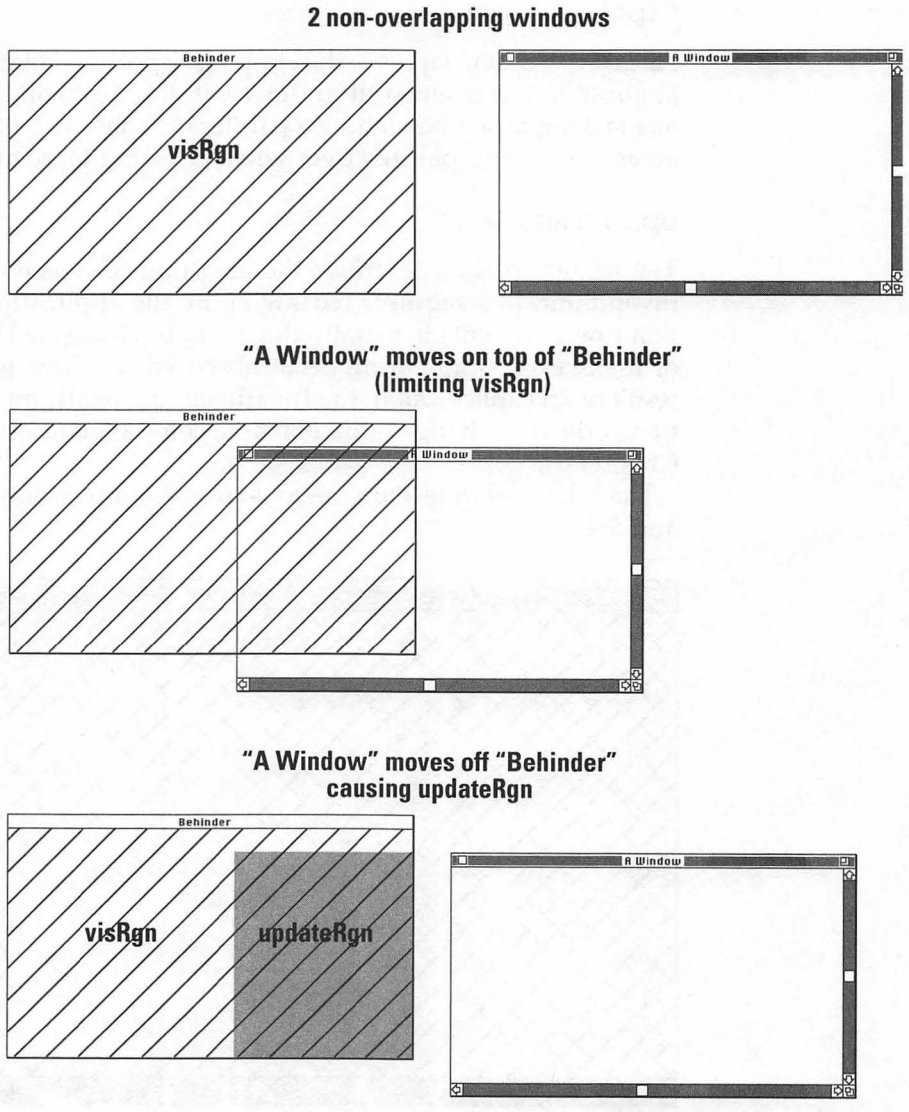


Figure 5-2. Window and GrafPort regions

Important ▶

All displayable graphic operations occur within the intersection of the clipping, content, and visible regions. The content region is the maximum area of the window available for an application's graphic presentation.

▶ Window Types and Presentation Considerations

The architecture of the software that supports window-based graphics display is largely determined by the purpose of the graphic. In particular, does the graphic reflect some important aspect of the user's data, or is the graphic being used to implement a user interface control or define the effects of a data editing "tool"?

- **Document windows** — The primary object of computation: the user's data set. The term "document" is often used to describe this data set, which is often loaded from or saved to disk. Within most useful or interesting applications, the largest portion of programming effort is made on behalf of supporting document windows.
- **Tool palettes** — Tool palette windows are often used to define the mouse's "operational mode." The functionality of tool palettes is quite similar to a set of menu options or a group of radio buttons; however, the graphical nature of a tool palette can convey to graphically inclined users more meaning and purpose regarding available functionality than text in menu items or radio buttons.
- **Simple dialogs** — Dialogs are frequently used to manipulate one feature or a few related features of document presentation. Because these dialogs are usually modal, the rendering and mouse-tracking tasks can be handled through one or two dedicated routines, without having to account for update-oriented rendering (discussed later in this chapter.) Alerts rarely if ever have significant graphics associated with their display, and as such don't require any special graphics support.

Are there any other distinct ways to utilize windows? It certainly depends on your own judgment, and this classification scheme may not accommodate your development strategies adequately. In any case, we can use this scheme to assess the general implementation requirements for the graphics programming effort required of particular application features.

▶ Views

The applications developed for the early Macintosh models tended to use windows simply. Windows often contained no more than a couple of application-specific data types and a slightly greater number of different interface object types: only one or two. This organizational discipline accounted for the following functional window types:

- simple documents
- user alerts
- modeless control panels and tool palettes
- modal control panels

For the first few years of Macintosh computing, the majority of windows implemented for applications fit somewhere in this classification scheme. Figure 5-3 shows a number of familiar window examples, grouped by functional type.

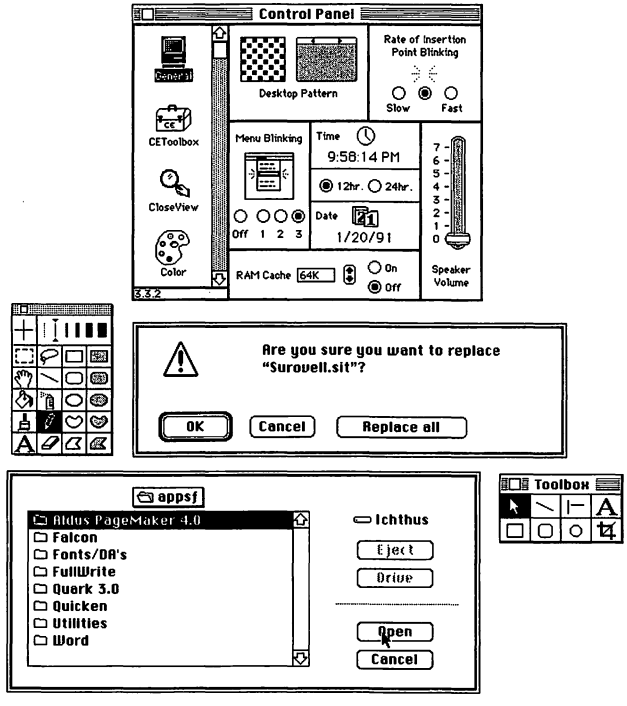


Figure 5-3. Familiar window examples

As the Macintosh toolbox grew and developers became more facile with Macintosh programming techniques, greater amounts of functionality and features were crammed into a smaller number of windows. Dialog boxes were loaded with controls and combined with other dialogs. On a few occasions, they even became the entire user interface. Document windows became increasingly adorned with miniature controls, split views, and menubars. The distinction between document windows and control panels, not particularly clear to begin with, became even more blurred.

By the Way ▶

Perhaps Apple's Control Panel started this functional explosion. This window is the prime example of features jam-packed into a too-tiny window: the General 'cdev' (Control Panel device) contains three custom controls and sports two main subareas, one of which is again divided seven more times.

For the sake of discussion, we'll call these "subwindow" areas *views*. Other names, such as panes, panels, display partitions, have been used for this conceptual domain. As far as Toolbox Managers are concerned, the management of views inside of a window isn't supported to any great degree:

- QuickDraw deals with drawing areas on a GrafPort basis. The ability to set a clipping region (via the **GetClip**(), **SetClip**(), and **ClipRect**() traps) and redefine a GrafPort's coordinate system (via the **SetOrigin**() trap) is useful in view management, but it alone doesn't constitute a view management package.
- The Control Manager doesn't quite go as far as we might like; the control implementation is adequate for simple buttons, scrollbars, and sliders, but doesn't go much beyond that. Implementing a custom control definition procedure, or CDEF, is elaborate, precluding streamlined development and testing. Additionally, it's difficult to direct keystroke events to specific controls, which might be useful for text entry, for example.
- The Dialog Manager, while a great solution for simple modal data entry, is sluggish and nearly unusable for applications that require multiple instances of modeless windows.

The harsh reality is that the tasks required of view support have to be tackled by application engineers on a window-by-window, view-

by-view basis. The lack of a bonafide "View Manager" forces developers to either write their own or to make a special case of each view by writing a separate controlling routine called from the main event loop.

Note ▶

If you're familiar with MacApp, you probably know that it uses and implements views and even calls them "views." Software developers and marketers who implement or reverse-engineer an existing good product or idea tend to rush to their thesauruses for a "better" name. We didn't want to rename the view concept, so we chose the term "view" for its descriptive value rather than its "MacAppness."

The notion of a view is still vague, so let's tighten it up a bit. Some schools of software development consider a view a window; others insist that each and every discrete object in a window is, in fact, a view. There are some good reasons behind both "viewpoints." For the sake of discussion, here's a working definition that lies somewhere in between these extremes:

- **Single data display** — Each view is dedicated to the display and control of a single functional aspect. A view can contain more than one object, such as a group of controls or other indicators, but all objects in the group should be closely related.
- **One coordinate system** — A view should display information and handle events in view-local coordinates. These are calculated relative to the view boundaries rather than those of the owning window.
- **One purpose** — The role of any view should be able to be described in one or two sentences.

Additionally, views can be nested, that is, a view can contain other views or subviews. Ideally, view nesting should be limited to one level; deeper nesting tends to obscure the purpose of the window and confuse the user. Analyzing the Control Panel in Figure 5-4 by these criteria, we can conclude that there are seven views.

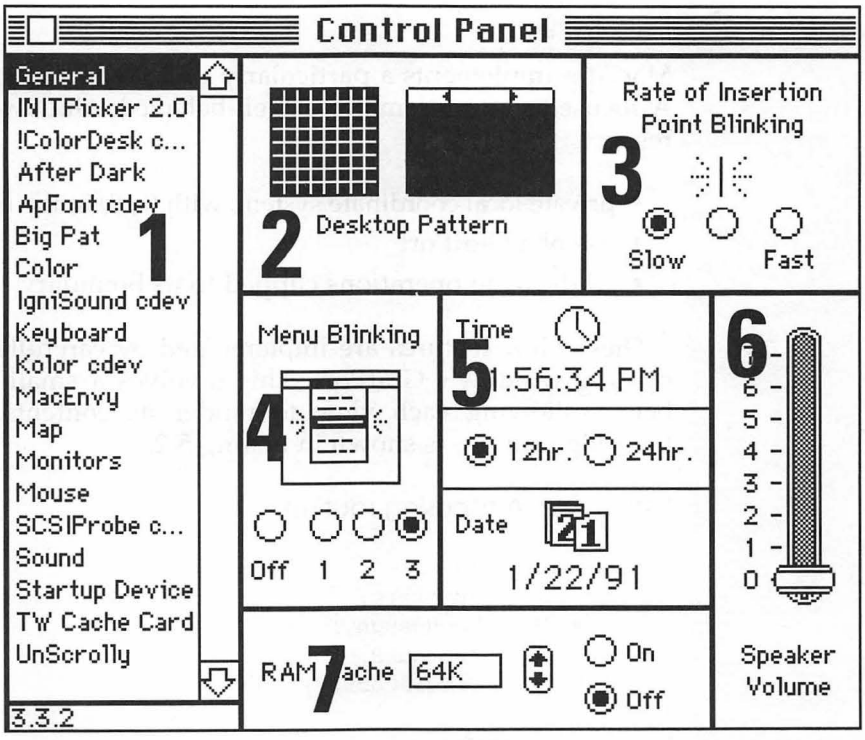


Figure 5-4. The Control Panel and its views

This definition is meant to be nothing more than a guideline to clarify and simplify the construction of complex windows. You can cram as many items into a view as you want and give each one its own coordinate system. However, a confused user probably won't become a frequent user, so strive to keep your windows as simple and concise as possible. The ideal application consists of a relatively small set of simple, powerful tools, each with a minimum of functional modes and variations. Windows and menus are devoid of rarely used options and capabilities.

An in-depth discussion of views and view handling is outside of the scope of this book. However, there are some components of view handling which present fundamental techniques for onscreen graphic display, particularly in the management of window regions.

▶ Focusing

MacApp implements a particularly useful concept known as *focusing*. A focused view resembles a well-behaved window in the following respects:

- private local coordinate system, with (0, 0) in the upper-left corner
- use of a GrafPort
- all drawing operations clipped to its boundary

These view features are implemented by carefully maintaining the owning window's GrafPort. This involves a small amount of setup before allowing each view to render its contents. An example of focusing routines is shown in Listing 5-2.

Listing 5-2: A focusing routine

```
void Port_Focus(
  void        *srcPort,
  RgnHandle   boundsRgn,
  Rect        *boundsR,
  Point       originCoord )
{
GrafPtr   savedPort;
Rect      portBoundsR;

  /* determine and set port to work upon */
  if (srcPort == NULL)
    srcPort = thePort;

  /* make the target port current */
  GetPort( &savedPort );
  SetPort( (GrafPtr)srcPort );

  /* determine boundary to use: region or rect */
  if ((boundsRgn == NULL) && (boundsR == NULL))
  {
    portBoundsR = ((GrafPtr)srcPort)->portRect;
    boundsR = &portBoundsR;
  }

  SetOrigin( -originCoord.h, -originCoord.v );
```

```
  /* set the clipRgn to the view's boundary */
  if (boundsRgn != NULL)
    SetClip( boundsRgn );
  else
    ClipRect( boundsR );

  SetPort( savedPort );
}
```

This routine allows for focusing to either a region or a rectangle. Listing 5-3 shows how a window might use focusing for rendering a list of views.

Listing 5-3. Rendering views, with focusing

```
void Window_RenderViews(
  WindowPtr   theWindowP )
{
ViewListHdl     theViewList;
ViewListEntry   *curViewP;
Rect            originR;
Point           originPt;

  theViewList = (ViewListHdl)GetWRefCon( theWindowP );
  if (theViewList != NULL)
  {
    curViewP = *theViewList;
    while (curViewP != NULL)
    {
      /* determine the view origin */
      if (curViewP->boundsRgn != NULL)
        originR = (**(curViewP->boundsRgn)).rgnBBox;
      else
        originR = curViewP->boundsR;
      SetPt( &originPt, originR.left, originR.top );

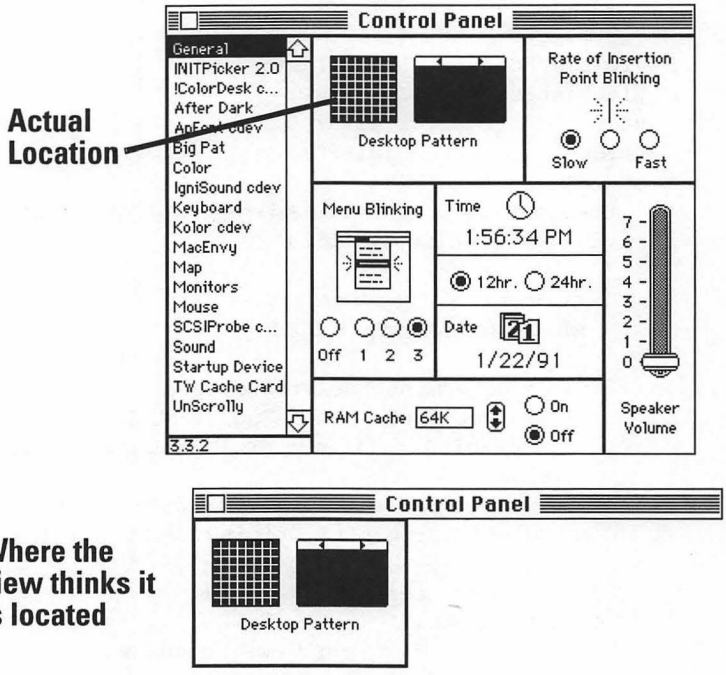
      /* focus the view */
      Port_Focus(
        theWindowP, curViewP->boundsRgn,
        &(curViewP->boundsR), originPt );

      /* invoke the view render procedure */
      /* to draw the view's contents */
      RenderView( curViewP );
```

```
        /* go to the next view */
        curViewP = curViewP->nextEntry;
      }
    }

    /* restore the window to "normal" clipping and origin values */
    SetOrigin( 0, 0 );
    ClipRect( &(theWindowP->portRect) );
}
```

The main objective of focusing is to set up the GrafPort in such a way that a view "thinks" it owns the GrafPort at the same time it's being prevented from drawing outside of its borders. Figure 5-5 illustrates the difference between focused and unfocused views, as they appear to a view-rendering routine.



When a view is focused, it acts as though it were in the upper-left corner of the window regardless of its actual location.

Figure 5-5. Focused and unfocused views

Focusing eliminates the need for view rendering routines that concern themselves with their absolute location within a window, as well as paying attention to their boundaries when rendering. When a view is focused, it acts as though it were in the upper-left corner of the window regardless of its actual location. As far as the routine is concerned, the upper-left corner of the view is (0, 0), and clipping is set to permit the entire view to be drawn into, while protecting everything outside of the view's boundaries from being drawn over.

▶ Basic Window Event-Handling Techniques

The interactive communication between the user and application is carried out through the transmission of events. Events are usually tied to a single window and typically tied to a particular region within that window.

Activate Events

Responding graphically to active events is generally fairly simple. If a document window becomes active (that is, is the new front window) it makes sense to make its GrafPort the current port via a **SetPort**() call. Any graphic object that has an associated selection range should be made visible on activation, and hidden upon window deactivation. No explicit drawing should take place; if some window object needs to be redrawn, then it should be invalidated and redrawn within the routine or method that handles the ensuing update event.

In the case of other window types, handling an activate event is even less involved. Modal windows are presumably only going to exist as front windows; hence the deactivated state can and should be ignored.

For tool palette windows, the issues are somewhat more complex. A tool window should appear to float above document windows, and its appearance shouldn't change in response to activation, nor should the appearance of *any other windows* change when a tool window is selected. Unfortunately, the Window Manager doesn't fully support the "floating" aspect of tool palettes. Calling **SelectWindow**() will cause the current front window to lose its highlighting and receive a deactivate event. The **BringToFront**() trap can be used instead, provided that the application can discriminate between its document and tool windows. The example in Listing 5-4 illustrates these issues.

Listing 5-4. Handling mouse clicks in a non-front window

```
if (theMousedWindow != FrontWindow())
{
  if (Application_IsToolWindow( theMousedWindow ))
    BringToFront( theMousedWindow );
  else
    SelectWindow( theMousedWindow );
}
```

The Application_IsToolWindow() routine would determine whether the "theMousedWindow" is a tool window or not, possibly by comparing the window's pointer or refCon values against a list of such values maintained by the application itself.

Update Events

The update event is the point at which an application makes the results of graphic operations visible in a window. This presentation typically involves drawing directly into a window, as shown in Listing 5-5.

Listing 5-5. Basic update event routine

```
void DoUpdateEvent(
  WindowPtr   theWindowP )
{
GrafPtr   savedPort;

  if (theWindowP == NULL)
    RETURN;

  /* save current GrafPort */
  GetPort( &savedPort );
  SetPort( theWindowP );

  BeginUpdate( theWindowP );
    /* render the contents of the window */
    ForeColor( redColor );
    TextFont( newYork );
    TextSize( 48 );
    MoveTo( 10, 100 );
    DrawString( "\pThis space for rent." );
    ForeColor( blackColor );
```

```
    /* draw all controls and grow icon */
    DrawControls( theWindowP );
    DrawGrowIcon( theWindowP );
  EndUpdate( theWindowP );

  /* restore original GrafPort */
  SetPort( savedPort );
}
```

Listing 5-5 illustrates the following important points:

- The GrafPort of the updating window might not be the current GrafPort, probably because that window isn't front-most.
- The use of the **BeginUpdate()** and **EndUpdate()** traps to restrict the performed drawing to only those window areas that need to be redrawn. **BeginUpdate()** restricts the performed drawing to only those window areas that need to be redrawn, by temporarily setting the window's *visRgn* equal to the intersection of the *clipRgn*, *visRgn*, and *updateRgn*. The *updateRgn* is then disposed of. **EndUpdate()** merely restores the *visRgn* to the appropriate area. This determination is made by the Window Manager, which takes the union of the window's structRgn and contRgn and subtracts the regions defined by overlapping portions of all windows in front of the updating window.
- The updating routine is responsible for all graphics that reside in the content region, including all controls. The routine is also responsible for drawing the grow region. **DrawGrowIcon()** calls the window's definition proc, or **WDEF**, which renders the grow icon only if the window is frontmost and its type is a resizable one (documentProc or zoomBoxProc). **DrawGrowIcon()** will also draw two lines 15 pixels away from the right side and bottom of the window, where it expects the application has placed two scrollbars. The updating routine is not responsible for the window's frame controls; those are handled directly by the WDEF.

This covers the basics of handling update events by drawing directly into a window. If an application's graphic requirements can be met with a small number of graphic commands, then this technique is sufficient. Most applications fall into this category. For graphics-intensive applications where the displayed graphics aren't easily rendered, an offscreen technique is preferable. The GWorld-buffered window approach is appropriate for applications that handle CAD, drafting, or paint documents. This technique involves the following steps:

- When creating a window, allocate a GWorld equal in size to the largest piece of the document to be rendered. If the document is small, then the GWorld's size can be equal to that of the document. However, if the document is very large (for example, a 32-bit, 4000-by-4000-pixel document), then the document may have to be rendered in pieces, as the memory requirements of the GWorld may be significantly greater than that which is available.
- Draw the window's contents, excluding controls and growboxes, into the GWorld rather than into the window.
- When an update event occurs, use **CopyBits**() to transfer the GWorld's graphic contents into the window.

This methodology generally yields the best performance with the most optimal color mapping available. Unfortunately, it places a potentially substantial demand for additional heap memory, and requires the presence of Color QuickDraw version 1.0. The decision whether to rely on offscreen or onscreen drawing techniques belongs to the developer and is one that should be made carefully.

▶ Desktop Interactions

Windows exist within the context of the Desktop. The Desktop is the "global" GrafPort or CGrafPort and is equal to the union of all visible areas on every active display card. There are a small number of situations in which applications need to be concerned with the interactions between its window and the Desktop.

A pointer to the Desktop's port can be obtained with either the **GetWMgrPort**() or **GetCWMgrPort**() traps. This port should be treated as a read-only entity, as modifications to it will affect all executing applications.

Display Boundaries

Prior to the introduction of the Macintosh II line, the screen of a Classic Macintosh was described by "screenBits", a BitMap in the application's global memory. This variable was initialized by the application's call to the **InitGraf**() trap and could be used to determine the screen's boundary. This was often used as shown in Listing 5-6.

Listing 5-6. Old-style usage of screenBits

```
void InitDesktopBoundsRects( void )
{
Rect dragLimitR, growLimitR;

   dragLimitR = screenBits.bounds;
   dragLimitR.top += GetMBarHeight();
   InsetRect( &dragLimitR, 2, 2 );

   /* set grow limit rect maximum and minimum size limits. */
   growLimitR.left = 70;
   growLimitR.top = 50;

   /* Note: maximums are arbitrary; make as large as necessary */
   growLimitR.right = screenBits.bounds.right;
   growLimitR.bottom = screenBits.bounds.bottom;
}
```

With the handling of multiple display monitors, this technique became inadequate. It's quite possible to configure a multiple monitor desktop so that the area isn't even rectangular. The solution can be found in the **GetGrayRgn**() trap, which returns a reference to the Desktop's visible region. This region's bounding rectangle can be used in place of "screenBits.bounds" to define common window limits. Recognize that this rectangle may contain areas that aren't visible, as shown in Figure 5-6.

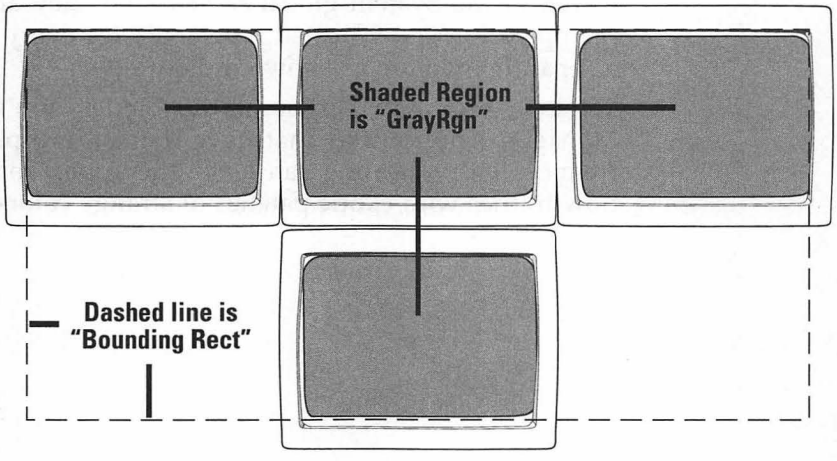


Figure 5-6. Non-rectangular desktop

Since the region returned by **GetGrayRgn**() belongs to the System, it should be treated as read-only and copied if the application needs to modify it.

Windows, Palettes, and Multiple Monitors

A number of peculiar situations can arise when working with multiple display of different capabilities for which applications may need to be accountable. The most important conditions are the following:

- A family of windows that extends across multiple devices *and* shares a common palette may appear differently, even when one of them becomes the frontmost window. Changes to the window order will only cause entry stealing for those windows that are all or partially on the same indexed display as the new front window. Windows that are entirely on other displays will be unaffected, even if other windows in their palette family experience entry stealing. A crude but effective solution to this problem is to temporarily make a window on the other devices the frontmost window via **SelectWindow**(), restoring the original frontmost window afterward.
- Since a window can extend across displays of differing capabilities, the window's *portPixMap* (or *portBits*) cannot be used to determine information about the window's display state. **CopyBits**() and other raster-oriented traps will recognize PixMaps and BitMaps as belonging to a window by comparing their *baseAddr* field's value to that of the Main GDevice PixMap (this value is also stored in the System global *scrnBase*); if they are the same, then the operation is performed repeatedly for every area that falls on a separate monitor, as shown in Figure 5-7.
- Versions 6.1.7 and earlier of the Finder don't use the Palette Manager. Applications that have demanding palettes that cause frequent entry stealing can leave the appearance of Finder window frames with exotic patches of strange colors.

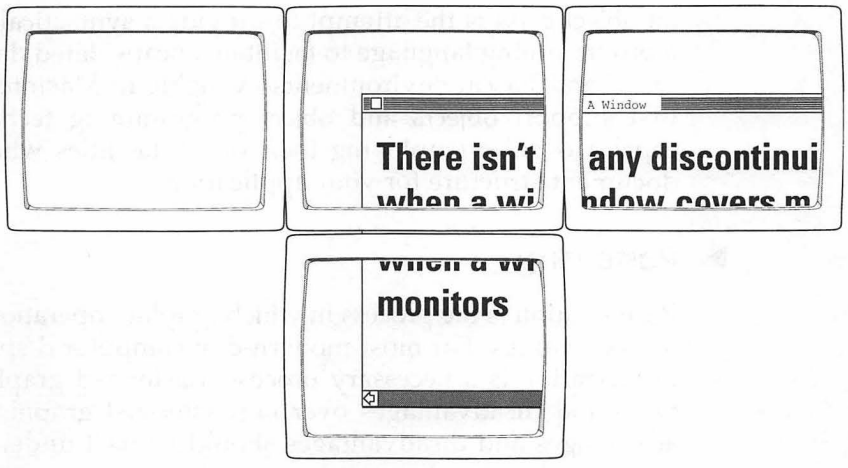


Figure 5-7. Drawing into window covering three displays

▶ Introduction to Document Organization

By this point, we've pretty much covered the general theories of offscreen and onscreen drawing, while detailing some of the important constraints and requirements inherent to both methods. Now we're going to look at specific techniques for maintaining a model of some "user-presentable" graphics and getting a representation of that model out to the user quickly and accurately.

Applications maintain a model of the graphics that they render. Whether that model is explicitly understood by the programmers who wrote the application is of no consequence: The model *does* exist. It may be expressed simply as the state of a few application variables, or it may be a sprawling database of point values whose rendering involves hundreds of routines external to the application itself. The degree to which an application is aware of its intended presentation model, and to which it organizes and manages those data structures that affect it, defines the flexibility of the graphics model and the ease with which it can be accommodated into the Desktop environment.

The applicable object programming term here is *encapsulation*. Encapsulating a programming solution means tightly defining the inputs, outputs, and operations that can be performed on a collection of related data objects — the essence of a graphics model. Developing a graphics model that relies heavily on global variables or a wide range of unrelated utility routines is difficult to encapsulate. The concept of

an object class is the attempt to provide a syntactical structure within a programming language to facilitate encapsulated data models. There are C and Pascal environments available to Macintosh programmers that support objects and object programming techniques and concepts. Consider employing their object facilities when designing the document structure for your applications.

▶ Rasterizing

Rasterization is the process in which graphics operations are committed to pixel values. For most modern-day computer display technologies, rasterization is a necessary process. Rasterized graphics have advantages and disadvantages over nonrasterized graphics. These relative advantages and disadvantages should be well understood by anyone attempting to develop a serious graphics-intensive application.

QuickDraw's GrafVerb-based drawing traps are inherently nonraster objects: The tokenized results of GrafVerb traps stored in pictures are also not rasterized, nor are PostScript commands. Within QuickDraw, rasterization occurs whenever drawing commands take place within a regular or color GrafPort (or CGrafPort) in such a way that the port's pixel values are altered. The main differences between raster and nonraster graphic objects are the following:

- **Raster is faster or slower** — Scaling calculations involving horizontal, vertical, and pixel depth must be made for every rasterized graphic operation. The transfer of one raster buffer (or PixMap) to another can be considered a single graphics operation. In most cases it's preferable to render complex graphics composed of numerous command-oriented operations into an offscreen GWorld and use **CopyBits**() to copy it to the destination GDevice. Even though this may take more execution cycles, the result will usually *appear* to be faster.
- **Raster is bigger** — A filled rectangle can be described by a Rect structure along with the pen state variables that affect it. The pixel representation of a filled rectangle will take orders of magnitude more memory.
- **Raster is dumber** — Unfortunately, rasterization loses most of the information associated with the rasterized graphic object. It's simple enough for an application to remember that it drew a circle into a PixMap; to find the pixels affected by the rasterization in the PixMap itself and then determine that those pixels indeed compose a circle is a nontrivial exercise in pattern recognition.

Rasterizing graphic objects improperly is the primary cause of aliasing. In order to prevent aliasing, the original, nonrasterized graphics commands should be preserved whenever possible. Scaling operations, which cause rasterized objects to develop those ugly jagged edges familiar to most computer users, generally do not affect nonraster graphic objects.

When to Rasterize?

Rasterizing constrains graphic objects according to the rules of the GDevice and PixMap into which the object was rendered, particularly the horizontal and vertical resolution, and the color space resolution. Preferably, rasterization should be performed on a "demand" basis only when displaying graphics on a raster display.

▶ Document Rendering Techniques

Now that the importance of a graphics presentation model is understood, let's fit it into the Desktop environment.

What and When to Render?

Within the Macintosh environment, there are three distinct conditions under which an application is directed to render its graphic model:

- the update event
- a print command
- a Clipboard command (Cut or Copy)

With the advent of System 7.0, the Clipboard command set has been extended. A subscriber can request a publishing application to render a graphic object in a manner similar to the more familiar Copy command.

Essentially, the request implied by all of these situations is the same: draw the graphic-requested objects. Unfortunately, the constraints imposed by each rendering situation can preclude the use of a single, generalized rendering routine. With care, the same rendering routine can be used, assuming the proper precautions are taken.

Update Events

As discussed earlier in this chapter, update event processing will configure the window's GrafPort or CGrafPort so that drawing cannot occur outside of the area to be redrawn. This process is intended to be

transparent to the application, and the rendering routine is supposed to pretend that it should draw the entire graphic, even if that graphic is substantially larger that the invalid area itself. For update routines that don't use any raster onscreen techniques, some performance improvement can be realized by examining the window's visRgn and explicitly clipping its drawing operations to that region by ignoring every graphic object that doesn't intersect it, rather than relying on QuickDraw to perform all of the clipping internally at the pixel level.

Clipboard Operations

Compared to update events, Clipboard operations often require the entire graphic to be rendered. Although any subselection rendering is up the the user, the tools for making such selections must be implemented and provided to the user by the application itself. The rendering should be unaffected by window or display limitation.

Printing the Rendered Document

When an application receives a printing request, it must undertake a fairly involved process beginning with the optional presentation of printing dialogs and culminating in the page-by-page rendering of the graphic into a GrafPort specially configured by the Printing Manager and the associated print driver. The application is responsible for determining the number of pages required to print the entire graphic and calculating the rectangles for each page. The application must not change the port or device while the printing port is open.

With all of these constraints well understood, let's dive into a couple of Window classes that address these concerns in a direct, concise fashion.

▶ Printing

As mentioned briefly in the previous section, Macintosh printing is essentially a page-oriented rendering operation that occurs within a specially prepared GrafPort or CGrafPort rather than a window. Most programmers are also aware of the fact that many of the popular Apple-compatible printing devices render graphics in a PostScript rather than a QuickDraw environment. These two factors, along with the handling of the standard Print and Page Setup dialogs, are the three most important issues facing developers who require hardcopy output from their applications.

▶ Overview of the Printing Process

The printing process is typically initiated by the user, and its implementation involves a joint effort between the application, the Printing Manager, print drivers, and, ultimately, the printer and its page description language. This process proceeds as follows:

1. The user selects Print from the application's or the Finder's File menu. The document (or documents) to be printed is usually represented by the icons selected in the Finder, or the one associated with the frontmost window of the application.
2. If printing from the Finder, the application is launched.
3. The application opens the current print driver, prepares a TPrint record, and passes it to **PrJobDlog**(), which presents the Standard Print dialog. The application may also present its own print dialog, if needed. The user either cancels the dialog, aborting the process, or OKs it.
4. The application opens the printer port via the **PrOpenDoc**() trap, which creates and configures the port and returns a reference to it back to the application.
5. The application, using the *rInfo* field within the TPrint record's TPrInfo substructure, determines the available printing area. Using this rectangle as a "window" into the document, the application opens a print page via the **PrOpenPage**() trap, renders that portion of the document's contents, and closes the page via the **PrClosePage**() trap. The process is repeated for all pages within the range specified in the TPrint record.
6. The application closes the printer port via the **PrCloseDoc**() trap. If the user selected draft mode printing, the application must then call the **PrPicFile**() trap to print the rasterized print image. The application's printing work is now finished.
7. If draft mode printing wasn't selected, then the printer driver goes to work. If the printer uses a page description language other than QuickDraw, then the driver converts all QuickDraw tokens into the appropriate command language program.
8. The resulting page description language command stream is executed by the printer device, which hopefully prints the document in short order.

The process described here doesn't account for print spooling. An example of handling application print duties is contained in the CWindo object described later in this chapter.

▶ Print Data Structures

Some of the important printing data structures are listed in the following sections.

TPrint, TPrInfo

As described previously, the TPrint record contains user-specified information as set by its actions within the standard Print dialog. Of particular interest to the application are the page range, print mode, and type of printer port to render into (color/grayscale or black/white). This data type is usually referenced through a THPrint, a type defined as a Handle to a TPrint record. A TPrint record contains a TPrInfo record as one of its fields, which contains available print rectangle and resolution information (see Listing 5-7).

Listing 5-7. TPrint and TPrInfo

```
typedef struct TPrInfo
{
  short     iDev;
  short     iVRes;
  short     iHRes;
  Rect      rPage;
};

typedef struct TPrint
{
  short       iPrVersion;
  TPrInfo     prInfo;
  Rect        rPaper;
  TPrStl      prStl;
  TPrInfo     prInfoPT;
  TPrXInfo    prXInfo;
  TPrJob:     prJob;
  short       printX[19];
}
TPrint, **THPrint;
```

TPrPort

This data structure is essentially a GrafPort or CGrafPort whose bottleneck procedures (see Chapter 2) have been set to handle the specialized requirements of the print driver and the destination printer's page description language.

▶ Important Print Traps

The Printing Manager is composed of a relatively small number of traps and data types. The more important ones are described here.

PrDefault(), PrValidate()

These traps initialize a TPrint record through a THPrint reference according to the requirements of the current print driver.

PrStlDlog()

This trap takes a THPrint argument and displays the standard Page Setup dialog. If the user selects OK, the trap configures the associated TPrint record according to the setting of the dialog. The printer's page rectangle can be adjusted here. This trap returns TRUE if the user selected the dialog's OK button, and FALSE otherwise.

PrJobDlog()

This trap takes a THPrint argument and displays the standard Print dialog. If the user selects OK, the trap configures the associated TPrint record according to the settings of the dialog. The printer port's type (GrafPort or CGrafPort) can be set, along with the print mode, printed page range, and further adjustments to the printed page's rectangle. This trap returns TRUE if the user selected the dialog's OK button, and FALSE otherwise.

PrOpenDoc(), PrCloseDoc()

PrOpenDoc() allocates and initializes the appropriate type of printing port (CGrafPort or GrafPort), installs the necessary bottleneck procedure pointers, and returns a reference to it back to the application. **PrCloseDoc**() deallocates the printer port, writes a temporary print data file, and invokes the printer driver to process it.

PrOpenPage(), PrClosePage()

These two traps open and close a single printer page. The application is responsible for calculating the source rectangle within its representation model; however, the destination rectangle is always the same and can be retrieved from the THPrint reference.

PrGeneral()

This trap allows the application to access and modify additional features specific to the current printer driver. Among other things, it can be used to determine the selected page orientation (portrait or landscape) and set the printer's horizontal and vertical resolution values.

▶ A Basic Window Object

The CWindo class is what's known as an *abstract superclass*. This means that CWindo objects generally don't exist by themselves, rather that CWindo is meant to be used as a foundation for other classes that will derive much of their behavior from it.

▶ Objectives

The CWindo class provides a sensible rendering scheme for all drawing situations, as discussed earlier in the "Document Rendering Techniques" section. Updating, printing, and clipboard operations all reference a single Render method.

Avoiding crashes due to referencing nonexistent System facilities is a very important CWindo design objective. The creation of windows may be undertaken by any one of four Window Manager traps: **NewWindow**(), **GetNewWindow**(), **NewCWindow**(), and **GetNewCWindow**(). The latter two are only available with Color QuickDraw. The two methods in the CWindo class that create windows check to see which one is appropriate before making the trap call. All other methods that utilize QuickDraw use only Classic QuickDraw trap calls.

Another important aspect of the CWindo class is its ability to link CWindo objects together and search that linked list for individual members. This feature allows simple event handling and is particularly useful when keeping track of even a small number of windows.

▶ Class Definition

The full definition of the CWindo class is shown in Listing 5-8.

Listing 5-8. CWindo class definition

```
struct Windo: indirect
{
  long            objectID;     /* housekeeping aid */
  Windo           *nextObject;  /* next article in the list */

  WindowPtr       theWindowP;   /* the genuine window article */
  short           windResID;    /* window resource ID, */
                                /* if applicable */

  Rect            growBoundsR;  /* resizing limits */
  Rect            dragBoundsR;  /* drag movement limits */

  Boolean         doBackClicks; /* handle mouse click if not */
                                /* front window? */
  EventRecord     curEvent;     /* event to consider */
  EventRecord     lastEvent;    /* last event processed */

  THPrint         printingInfo; /* print and page setup */
                                /* information */

  void            *extraData;   /* object's private data */
  unsigned long   extraDataLen; /* sizeof( object's private */
                                /* data ) */


  void Init( void );            /* set to known empty state */

  void Dispose( void );         /* close window; delete object */

  /* methods to handle linked lists of CWindo objects */
  void AddIndObject(            /* link another CWindo to */
    Windo *addedObject );       /* this one */

  Windo* KillIndObject(         /* remove linked CWindo from */
    Windo *dyingObject );       /* this */

  short CountObjects( void );   /* count CWindos linked to */
                                /* this one */
```

```
Windo* GetIndObject(                /* get nth linked CWindo */
  short objIndex );

long GetObjectID( void );           /* get objectID value */

void SetObjectID(                   /* set new objectID value */
  long newObjectID );

Windo* GetObjectByID(               /* find linked CWindo, */
  long  targetObjectID );           /* by ID value */

WindowPtr GetWindow( void );        /* obtain owned window ptr */

void SetWindow(
  WindowPtr theWindowP );           /* set new window ptr */

OSErr Create(                       /* analogous to NewWindow() */
  void      *wStorage,              /* trap */
  Rect      *boundsunsigned,
  short     wProcID,
  char      *wTitle,
  short     isVisible,
  Boolean   hasGoAway );

OSErr CreateRes(                    /* analogous to GetNewWindow() */
  void      *wStorage,              /* trap */
  short     theWindID );

void GetDocSize(                    /* get current maximum window */
  short     *width,                 /* dimensions */
  short     *height );

void SetDocSize(                    /* set maximum window dimensions */
  short     width,
  short     height );

OSErr Render(       /* •• VIRTUAL •• */
  void      *desiredArea,           /* draw window contents */
  short     boundsType,
  short     situation );

Boolean GetRenderBounds(            /* determine bounds within */
  void        *desiredArea,         /* which to draw */
  short       boundsType,
  Rect        *boundsR,
  RgnHandle   *boundsRgn );
```

```
  OSErr DoPageSetup( void );  /* handle and display "Page */
                              /* Setup..." dialog */

  OSErr DoPrint( void );      /* print entire document in */
                              /* pages */

  OSErr DoClipCopy( void );   /* copy to the Clipboard */

  Boolean IsWindowEvent(      /* does this event pertain to */
    EventRecord *theEvent ); /* this window? */

  short DoWindowEvent(        /* process the event */
    EventRecord *theEvent );

  Windo* FindWindowEvent(     /* determine which linked */
    EventRecord *theEvent ); /* window "owns" the event */


  /* specific event handlers called by DoWindowEvent() method */
  void DoActivate( void );    /* handle activate event */

  void DoUpdate( void );      /* handle update event */

  Boolean DoKeyDown( void );  /* handle key down event */

  Boolean DoMouseDown(        /* handle mouse down event */
    short theWindowPart );

  Boolean DoDrag( void );     /* handle mouse down in */
                              /* drag event */

  Boolean DoGoAway( void );   /* handle mouse down in go-away */
                              /* box event */

  Boolean DoZoom(             /* handle mouse down in zoom */
    short theWindowPart );    /* box event */

  Boolean DoGrow( void );     /* handle mouse down in grow */
                              /* region event */

  void ResizeView(            /* resize a window */
    Point newDimension );

  void InvalidateView(        /* invalidate an entire window */
    Boolean doEraseAlso );
};
```

▶ Public Methods

The CWindo class may seem quite large, but in fact there are only a small number of methods that programmers need to be concerned with. These methods are called *public* methods. The other methods and instance variables are considered *private* and are intended for internal use only by other CWindo methods. The public methods are described in the following sections.

You may also note in Listing 5-8 that some of the method declarations are commented as being "virtual." A *virtual* method is one that is intended to serve as a placeholder or stub. Such methods should be *overridden*, or replaced, by classes which are derived from CWindo. Since Think C 4.0 doesn't support virtual methods explicitly, those CWindo methods which are considered virtual are so noted with comments.

Note ▶

Complete source code for the CWindo class can be found on the Source Code Disk.

Init(), Dispose()

Init() initializes the object's private variables, or instance variables. **Dispose**() deallocates any dynamic memory allocation referenced by the instance variables.

Create(), CreateRes()

These methods create a window and initialize the object's fields concerned with growing and dragging the window. Create() creates a window from scratch, whereas CreateRes() builds one from an available 'WIND' resource. The source to Create() is shown in Listing 5-9.

Listing 5-9. CWindo::Create() method

```
OSErr CWindo::Create(
  void            *wStorage,
  Rect            *boundsR,
  short           wProcID,
  unsigned char   *wTitle,
  short           isVisible,
  Boolean         hasGoAway )
{
```

```
RgnHandle          screenBoundsRgn;
SysEnvRec          curSysInfo;
Rect               localBoundsR;

  if (wTitle == NULL)
    wTitle = "\p";
  if (boundsR == NULL)
  {
    SetRect( &localBoundsR, 0, 0, 100, 100 );
    boundsR = &localBoundsR;
  }

  /* create the window */
  (void)SysEnvirons( curSysEnvVers, &curSysInfo );
  if (curSysInfo.hasColorQD)
    this->theWindowP =
      NewCWindow(
        (Ptr)wStorage, boundsR, (char*)wTitle, isVisible,
        wProcID, (WindowPtr)(-1), hasGoAway, 0L );
  else
    this->theWindowP =
      NewWindow(
        (Ptr)wStorage, boundsR, (char*)wTitle, isVisible,
        wProcID, (WindowPtr)(-1), hasGoAway, 0L );

  if (->theWindowP ! = NULL)
    return (OSErr) (-1);

  screenBoundsRgn = GetGrayRgn();
  this-> dragBoundsR = (**screenBoundsRgn).rgnBBox;
  this-> dragBoundsR.top += GetMBarHeight();

  SetRect(
    &(this->growBoundsR), 30, 30,
    (**screenBoundsRgn).rgnBBox.right,
    (**screenBoundsRgn).rgnBBox.bottom );

  SetPort( this->theWindowP );
  return noErr;
}
```

Render(), GetRenderBounds()

This method is a "virtual" method — one that serves as a placeholder for one implemented by those subclasses derived from it. The Render() method of the CWindo class simply erases the window. The "situa-

tion" argument indicates whether the "desiredArea" argument is a RgnHandle or a Rect. GetRenderBounds() is used to make further determinations about the area to be rendered. The "situation" argument indicates which of the three rendering cases is requested.

Subclasses of CWindo should implement a Render() method that meets the following criteria: no extra redrawing of pixels onscreen to avoid flicker, and an ability to handle either color or monochrome GrafPorts.

The source to Render() is shown in Listing 5-10.

Listing 5-10. CWindo::Render method

```
OSErr CWindo::Render(     /* •• VIRTUAL •• */
  void   *desiredArea,    /* area to be drawn */
  short  boundsType,      /* Rect, Rgn or ??? */
  short  situation )      /* update, copy, print or ??? */
{
RgnHandle   boundsRgn;
Rect        boundsR;

  if (!this->GetRenderBounds(
       desiredArea, boundsType, &boundsR, &boundsRgn ))
    return (OSErr)(-1);

  if (boundsRgn != NULL)
    EraseRgn( (RgnHandle)desiredArea );
  else
    EraseRect( &boundsR );

  return noErr;
}
<CWindo::Render method>
```

DoUpdate(), DoClipCopy(), DoPrint()

These methods prepare the port to be rendered into, determine the area to be rendered, and call the Render() method to draw the graphics within the specified area.

The source to DoUpdate() is shown in Listing 5-11.

Listing 5-11. CWindo::DoUpdate method

```
void CWindo::DoUpdate( void )
{
GrafPtr     savedPort;
```

```
RgnHandle   updateBounds;
Point       globalOffset;
OSErr       errStat;

  if (this->theWindowP == NULL)
    RETURN;

  /* save current GrafPort */
  GetPort( &savedPort );
  SetPort( this->theWindowP );

  updateBounds = NewRgn();
  if (updateBounds != NULL)
  {
    CopyRgn(
      ((CWindowPeek)(this->theWindowP))->updateRgn,
      updateBounds );
    globalOffset.h = globalOffset.v = 0;
    LocalToGlobal( &globalOffset );
    OffsetRgn( updateBounds, -globalOffset.h, -globalOffset.v );
  }

  BeginUpdate( this->theWindowP );
    errStat = this->Render(
      updateBounds, SHAPE_Rgn, DOCREND_update );

    DrawControls( this->theWindowP );
    DrawGrowIcon( this->theWindowP );
  EndUpdate( this->theWindowP );

  if (updateBounds != NULL)
    DisposeRgn( updateBounds );

  /* restore original GrafPort */
  SetPort( savedPort );
}
<CWindo::DoUpdate method>
```

IsWindowEvent(), FindWindowEvent(), DoEvent()

These three methods are used to perform the routing and handling of events associated with an object's windows. FindWindowEvent() will start with the original object and chain through the object's linked list pointers until either NULL is encountered or IsWindowEvent() matches the event argument to the owning CWindo object.

The source to DoUpdate() is shown in Listing 5-12.

Listing 5-12. CWindo::IsWindowEvent method

```
Boolean CWindo::IsWindowEvent(
  EventRecord *theEvent )
{
WindowPtr   theWindow;
short       pressedWindowPart;

  if ((theEvent == NULL) || (this->theWindowP == NULL))
    return FALSE;

  switch (theEvent->what)
  {
    case mouseDown:
    case mouseUp:
      pressedWindowPart =
        FindWindow( theEvent->where, &theWindow );
      return (theWindow == this->theWindowP);

    case activateEvt:
    case updateEvt:
      theWindow = (WindowPtr)(theEvent->message);
      return (theWindow == this->theWindowP);

    case keyDown:
    case autoKey:
      if (theEvent->modifiers & cmdKey)
        return FALSE;
      else
        return (this->theWindowP == FrontWindow());

    case diskEvt:
    case app4Evt:
    default:
      return FALSE;
  }
}
<CWindo::IsWindowEvent method>
```

▶ A GWorld Window Object

▶ Objectives

CPixBufWindo, an object that implements PixMap-buffered window, uses the GWorld functionality presented in Chapter 4. Its objectives are essentially a superset of the CWindo objectives, which make it an ideal subclass of CWindo. In object programming terms, the CPixBuff class *inherits* from CWindo. The inheritance is substantial and results in a compact object with a few powerful methods that replace, or *override*, those identically named in the CWindo class, particularly the virtually defined Render() method.

As you are now well aware, GWorlds are huge heap monsters. The chances are low that enough heap memory exists to allocate a GWorld of the desired pixel depth and planar dimensions, and these chances decrease with every one allocated. Apple's GWorld usage guidelines suggest that window update routines that rely on GWorlds should be prepared to draw directly into the window's port should the offscreen GWorld be deallocated. The CPixBuff class accommodates this by allowing for routines that use these objects to install a rendering procedure into the objects themselves. This allows the window to be supported whether or not the GWorld is allocated.

▶ Class Definition

The full definition of the CPixBuffWindo class is shown in Listing 5-13.

Listing 5-13. CPixBuffWindo class definition

```
struct CPixBuffWindo: CWindo
{
  GWorldPtr   offWorld;          /* The off-screen pixels */
  GDHandle    screenGDev;        /* window's GrafDevice */
  ProcPtr     altRenderProc;     /* alternate rendering */
                                 /* routine ptr */
  Boolean     canGWorldPixMap;   /* working GetGWorldPixMap() */
                                 /* trap present? */

  Point       scrollOrigin;      /* current scrolled offset */
  long        docSize;           /* ?? */
  Ptr         docData;           /* ?? */

  void Init( void );             /* •• OVERRIDE •• */
```

```
    void Dispose( void );          /* •• OVERRIDE •• */

    Boolean AugmentWindow(
      Rect   *boundsR,
      short  pixDepth );

    void RenderWindow( void );

    OSErr Render(                  /* •• OVERRIDE •• */
      void   *desiredArea,
      short  boundsType,
      short  situation );

    void InstallAltRenderProc(
      ProcPtr newAltRenderProc );

    void EraseOffWorld( void );

    void GetScreenWorld( void );

    void SetScreenWorld( void );

    void SetOffWorld( void );

    void DisposeOffWorld( void );
};
```

You may notice that this class has many fewer methods than the CWindo class from which it's derived. This is because much of the behavior inherited from the CWindo class can be used without modification of any sort. However, some of the CWindo methods are insufficient for the CPixBuffWindo class, and will need to be partially or wholly overridden. In Think C 4.0, overriding isn't explicitly accounted for in the language, so those CPixBuffWindo methods that override CWindo methods of the same name are noted with comments.

▶ Public Methods

As with the CWindo class, there are a number of methods which should be considered public, and the rest are intended for internal use by other CPixBuffWindo methods. These public methods are described in the following sections.

AugmentWindow()

This method creates an offscreen GWorld to attach to the window. It accepts arguments for the planar and pixel depth dimensions. If the GWorld is successfully allocated, the inherited document size is adjusted and TRUE is returned. FALSE is returned on any failure to allocate the GWorld.

Render()

The Render() method is fairly involved. It handles the rendering of window contents for the update, clipboard copy, and printing cases, and for both color and monochrome GrafPorts.

The source to Render() is shown in Listing 5-14.

Listing 5-14. CPixBuffWindo::Render method

```
OSErr PixBuffWindo::Render( /* •• OVERRIDE •• */
  void   *desiredArea,      /* area to be drawn */
  short  boundsType,        /* Rect, Rgn or ??? */
  short  situation )        /* update, copy, print or ??? */
{
CGrafPtr      savedPort;
GDHandle      savedGDev;
PixMapHandle  srcPM;
PixMapPtr     destPMPtr;
RgnHandle     boundsRgn;
Rect          sourceR, destinationR;
Boolean       isGWorldOK;

  /* determine validity of offscreen World */
  if (this->offWorld == NULL)
    isGWorldOK = FALSE;
  else
  {
    if (this->canGWorldPixMap)
      srcPM = GetGWorldPixMap( this->offWorld );
    else
      srcPM = ((CGrafPtr)(this->offWorld))->portPixMap;
    isGWorldOK = LockPixels( srcPM );
  }

  /* if offscreen world isn't available, try to use the */
  /* alternate or inherited drawing procs */
  if (!isGWorldOK)
  {
```

```
    if (this->altRenderProc != NULL)
      (*(this->altRenderProc))
        ( desiredArea, boundsType, situation );
    else
      inherited::Render( desiredArea, boundsType, situation );
    return (OSErr)(-1);
  }

  /* determine boundaries within which to draw */
  if (!this->GetRenderBounds(
       desiredArea, boundsType, &sourceR, &boundsRgn ))
    return (OSErr)(-1);

  destinationR = sourceR;
  OffsetRect(
    &sourceR, this->scrollOrigin.h, this->scrollOrigin.v );

  /* if a clipboard render, draw based on the onscreen world */
  if (situation == DOCREND_copy)
  {
    GetGWorld( &savedPort, &savedGDev );
    this->SetScreenWorld();
    if (this->theWindowP != NULL)
      SetPort( this->theWindowP );
  }

  /* determine destination PixMap or BitMap reference */
  if (PORT_IS_COLOR( thePort ))
    destPMPtr = *((CGrafPtr)thePort)->portPixMap;
  else
    destPMPtr = (PixMapPtr)&(thePort->portBits);

  /* finally! blast the pixels out! */
  /* BWCopyBits() is simply a covering routine to CopyBits() which */
  /* ensures that the current port's pen colors are black and white */
  BWCopyBits(
    *srcPM, destPMPtr,
    &sourceR, &destinationR,
    srcCopy, boundsRgn );

  if (situation == DOCREND_copy)
    SetGWorld( savedPort, savedGDev );
  UnlockPixels( srcPM );

  return noErr;
}
```

GetGWWorld()

This method gives the consumer access to the object's offscreen GWorld.

SetOffGWorld(), SetScreenGWorld()

This makes current the appropriate GWorld. SetOffGWorld() makes current the offscreen GWorld's port and GDevice; SetScreenGWorld() makes current the window and its associated GDevice.

▶ Summary

This chapter established the relationship between Macintosh graphics and windows. A discussion of the treatment of windows as graphics entities focused on the windows' regions and their management as handled by the application and the Window Manager, rendering and rendering requests, and window-based documents and data organization of graphics presentation models. Printing and the interrelationship between user, application, and system software were also briefly discussed. Finally, the windowing foundation of ArtToy, an example painting application, was discussed and presented in window object classes. The paint tools discussed in Chapter 6 will continue building the ArtToy application with the addition of painting tools. The full source code for the CWindo and CPixBufWindo() classes can be found on the optional source code disk.

6 ▶ Mouse-Based Graphics

When a user of the ArtToy paint application selects the pencil from the tool palette, moves the cursor over the content region of the current document, and presses the mouse button, he or she reasonably expects that the mouse will act like a pencil and draw a line in the current foreground color wherever the mouse is moved until the mouse button is released. The user has similar expectations for all of the tools in the tool palette. Turning a user's mouse into a drawing tool is what this chapter is about.

We'll continue to employ the ArtToy paint application as the basis for discussing the topic of mouse-based drawing. Although the applications you write won't use all of the techniques described in this chapter (the world doesn't need very many more paint applications), most of your applications will use some of them. And as you'll read in the sections ahead, mouse-based drawing techniques are not cut and dried — there is a lot of room for putting your own spin on the basic techniques to get just the tool your application needs.

We'll also continue to use the object-oriented approach to programming the techniques we discuss. The discussion naturally starts with a general description of the things that all mouse-based drawing techniques have in common and proceeds to describe the specific differences required by each individual tool. Such a progression from the general to the specific mirrors the relationship between high-level, abstract object classes as ancestors to a whole family of specific, utilitarian objects.

The object definitions given here use the Think C language syntax. If Think C isn't your native tongue, or if you're unfamiliar with the object-related syntax, don't worry. You'll still be able to read and understand the sense of what's being defined. And the objects should translate easily into any other object-oriented language.

▶ The Two Basic Types of Mouse-Based Tools

All of the wonderful things the user can do to a document with a mouse can be divided into two basic kinds of activities: drawing and selecting.

Drawing tools include the pencil and paintbrush tools as well as the geometric shapes like rectangles and ovals. Perhaps less-obvious drawing tools are the paint can, spray can, and eraser. All of them add pixel color to the "canvas" in response to mouse movement.

The selecting tools, on the other hand, let the user designate areas of the document as the object of some future activity such as deletion, rotation, or duplication. Like the drawing tools, selecting tools come in a variety of shapes and actions. And, as you'll see in the sections that follow, the selecting tools are largely based on the drawing tools.

▶ Sketching

To understand how the selecting tools are related to the drawing tools, consider the four user actions that are the basis of all ArtToy tools: point, click, drag, and release. When the cursor is in the document's content region, the cursor reminds the user what the mouse can do. The user points the cursor to a place in the window, clicks down the mouse button, drags the mouse to somewhere else in the window, and releases the button.

An application should present the user with the appropriate cursor at all times and respond correctly when the button goes down. Before the mouse button is pressed (during "pointing") there isn't much tool-specific processing to be done, just basic cursor maintenance. When the button goes down, the user is activating the tool that the mouse is understood to represent, whether it's a pencil, a spray can, a lasso, or whatever.

Most of what you do to effect a tool for the user is to watch the mouse button and do your tool-like thing while the user holds the button down (the "click" part). If the user is drawing a pencil line (the "dragging" part), the application repeatedly reads the mouse location as the mouse moves around on the screen, drawing line after line connecting successive points until the user lets up on the button.

While the user holds the mouse button down, the application's execution is very narrowly focused — it isn't interested in any other events until the mouse button is released.

The ToolBox provides two traps for the purpose of waiting for the mouse button to be let up.

```
Boolean StillDown( void );
Boolean WaitMouseUp( void );
```

These traps return TRUE if the mouse button has not been let up since the last mouseDown event. The **WaitMouseUp()** trap removes the mouseUp event from the event queue and is therefore preferred to the **StillDown()** trap. Listing 6-1 is a simplified but typical example of a pencil drawing tool.

Listing 6-1. A WaitMouseUp() loop

```
/* application-dependent external references */
enum
{
  PencilToolID  = 1,
  RectToolID    = 2,
  MarqueeToolID = 3
};

extern short  gCurrentToolID;

void TrackingSomeRoutine(
  EventRecord   *curEvent,
  short         curToolIndex )
{
EventRecord curEvent;
Point  curMouse, lastMouse;

  if ((curEvent->what != mouseDown) || (gCurrentToolID !=
                                        PencilToolID))
    return;

  curMouse = curEvent->where;
  GlobalToLocal( &curMouse );
  lastMouse = curMouse;

  MoveTo( curMouse.h, curMouse.v );
  while (WaitMouseUp())
```

```
    {
      GetMouse( &curMouse );
      if (!EqualPt( curMouse, lastMouse ))
      {
        LineTo( curMouse.h, curMouse.v );
        lastMouse = curMouse;
      }
    }
}
```

The **MoveTo**() and **LineTo**() trap calls are what make this code a pencil drawing tool. Without them the code routine becomes a generic template for mouse tracking.

The code inside the **WaitMouseUp**() loop implements the dragging part of point, click, drag, release — what we'll call *sketching*. It repeatedly reads the mouse and draws a shape on the screen in response to the user's mouse movements. The shape that is drawn during sketching depends on the tool being used. ArtToy exploits the fact that sketching functionality is shared by both selecting and drawing tools. Consider the pencil drawing tool and the lasso selecting tool. The pencil drawing tool is used to draw an irregular line while the lasso selecting tool is used to select an irregular area. Although the end result is very different, both the lasso selection and the pencil line are sketched in the same way. What differentiates the lasso and the pencil is what happens when the button is released. The important thing to remember is that whether users are drawing or selecting, first they have to sketch in order to define the area to be drawn or selected.

The same kind of relationship exists between the rectangle drawing tool and the rectangular marquee selecting tool. Both are sketched by showing the rectangle defined by the initial "click" point and the current cursor position. But when the button is released the marquee leaves running ants around a selected area of the screen while the rectangle tool leaves a rectangle in the current foreground color and pen shape.

The heart of every mouse-based tool is its sketching method — the QuickDraw traps called inside the **WaitMouseUp**() loop. The selecting tools use the drawing tools in that they use the same sketching methods to let the user define a shape or area. When the button is released the selecting tools and the drawing tools go their separate ways.

With the understanding that selection tools are just like drawing tools until the button is released, let's define the drawing tools first. After the sketching methods are defined, we'll come back to the selecting tools to describe their special behavior.

▶ What All Mouse-Based Objects Have in Common

The mouse-based tools, drawing and selecting, for ArtToy will require a good number of different objects and methods, but let's start with what all of the objects have in common. We need to define the granddaddy of all mouse-based objects, which contains the instance variables and methods common to all. The primary ancestor of all of the objects we'll declare to support mouse-based tools is called BasicMouseTool. Described in Listing 6-2, BasicMouseTool defines what all mouse-based drawing objects have in common.

Listing 6-2. BasicMouseTool class definition

```
/*
** Something a user draws onto the screen.
*/
struct BasicMouseTool : indirect
{
  Rect limitRect;           /* Drawing limits */

  void Init( void );        /* Initialization method */
  void ObjKill( void );     /* Detypedef struction method */

  void SetLimitRect(        /* access limiting rectangle */
    Rect *curLimitR );

  void GetLimitRect(
    Rect *curLimitR );

  void Draw(                /* ••• VIRTUAL ••• */
    short transferMode );   /* Overridden by subclasses */

  void Sketch(              /* ••• VIRTUAL ••• */
    Point anchorPt );       /* Overridden by subclasses */

  RgnHandle MakeRgn( void );/* Region made drawing the object */
};
```

The only instance variable described for the basic object is a rectangle called limitRect, which defines, in local window coordinates, the portion of the window in which the object can be drawn. The limitRect instance variable is accessed by two methods, SetLimitRect() and GetLimitRect().

The definition of the BasicMouseTool class includes two methods that should be included in every class definition: an Init() method to initialize a newly created object, and an ObjKill() method to prepare an existing object for deletion. Both the Init() and ObjKill() methods perform simple jobs. Init() sets each of the object's instance variables to known values; it would, for example, set Handles and pointers to NULL. ObjKill(), on the other hand, gets the object ready for deletion; it releases memory that the object allocated. [ObjKill() is called just before an object is deleted to avoid having memory rendered unavailable because they are allocated to a non–existent object.]

The Sketch() and Draw() methods are what give the objects their personalities. Sketch() handles the user interaction; its code contains the WaitMouseUp() loop for mouse tracking. The Draw() method renders the finished object in the current port.

The rules governing all Sketch() methods are the following:

- Use the patXor copy mode for drawing to the screen.
- Track the mouse while the user sketches the design.
- Save a description of the sketched design.
- Leave the screen the way you found it, with no trace of sketching.

The Draw() method uses the description of the sketched object to render the design in the current port.

The last method, MakeRgn(), creates a region in the shape of the design.

These are the instance variables (one) and methods inherited by all of the mouse-based drawing tools that we're going to define. Some methods, like SetLimitRect() and GetLimitRect(), are well defined in the basic object and are not overridden (modified) by the other classes. Other methods, like **Sketch**() and **Draw**(), are only "stubs" in the basic class and require definitions specific to each class you define. BasicMouseTool is an *abstract superclass* in that ArtToy never creates an instance of a BasicMouseTool. Instead, it serves as a foundation on which the actual tools that ArtToy uses for its mouse-based drawing can be built.

While still in the realm of the abstract, and before pressing on with further definitions of the subclasses of objects that constitute ArtToy's drawing tools, let's describe the basic algorithm in which ArtToy uses its drawing tools. [The code in Listing 6-3 takes place in the context of a mouseDown event in the content region of the front window.]

Listing 6-3. General drawing tool use algorithm

```
void PerformUserSketching( void )
{
WindowPtr   curFrontWindow;

  /* prepare for sketching into the front window */
  curFrontWindow = FrontWindow();
  if (curFrontWindow == NULL)
    RETURN;

  SetGWorld( curFrontWindow, GetGDevice() );

  /* Create new drawing object and make it initialize itself */
  myDrawTool = new( currentToolType );
  myDrawTool->Init();

  /* invoke tool's sketch method, which should allow the user */
  /* to sketch to his/her satisfaction */
  myDrawTool->SetLimitRect( &(thePort->portRect) );
  myDrawTool->Sketch();

  /* finished sketching, so now render the sketched graphic */
  /* into the document's offscreen GWorld */
  SetGWorld( frontWindowOffGWorld, NULL );
  myDrawTool->Draw();

  /* the tool object's job is done, so dispose it */
  myDrawTool->ObjKill();
  delete( myDrawTool );

  /* force an update event and let the window's */
  /* update event handling routine redraw it */
  SetGWorld( curFrontWindow, GetGDevice() );
  InvalRect( &(thePort>portRect) );
}
```

The drawing tool objects are created, used, and deleted within the time that the mouse button is pressed and released — they don't exist between mouseDown events. While the mouse button is down the drawing tool objects facilitate sketching. After the button is released the result of the sketching is drawn offscreen.

Consider what's *not* part of the basic tool definition. The tool objects have no notion of which port they're drawing in. They draw as easily to the screen as they do offscreen. It's up to code that utilizes the objects to set the correct drawing environment before invoking their sketching and drawing methods. Similarly, the drawing objects use whatever pen characteristics and colors happen to be in force when they're called (with the exception of the **Sketch**() method, which works in patXor mode). Such is the scope of the mouse-based drawing tools.

▶ Drawing Tools and Abstract Differentiation

Of all the tools available to the user of ArtToy, it is the drawing tools that let the user sketch a shape on the screen. Let's look at what drawing tools have in common and the purposes served by their differences.

The designs that ArtToy lets the user draw on the screen can be subdivided into two types: regular and irregular. For the purpose of building ArtToy, a regular shape is one that can be described by two points, while an irregular shape is described by a collection of a variable number of points.

▶ Regular Shapes

Two points describe a rectangle and there is a family of shapes that QuickDraw makes available that can be defined in terms of a rectangle. There are framed rectangles, rounded rectangles, and filled rectangles. These are all shapes that QuickDraw inscribes in a defining rectangle. See Figure 6-1 for some examples of QuickDraw's rectangular shapes.

Two points can also be used to describe a line segment. And in ArtToy, a line segment is used in the anchored line drawing tool. The user clicks on an initial (anchor) point and then clicks on a second (end) point. ArtToy draws a line between the two points. This drawing shape is familiar to any user of other paint applications.

The rectangular shapes and the anchored line can be described by two points, that is, two sets of grid coordinates. This fact gives rise to another abstract class of objects, RegularMouseTool, defined in Listing 6-4.

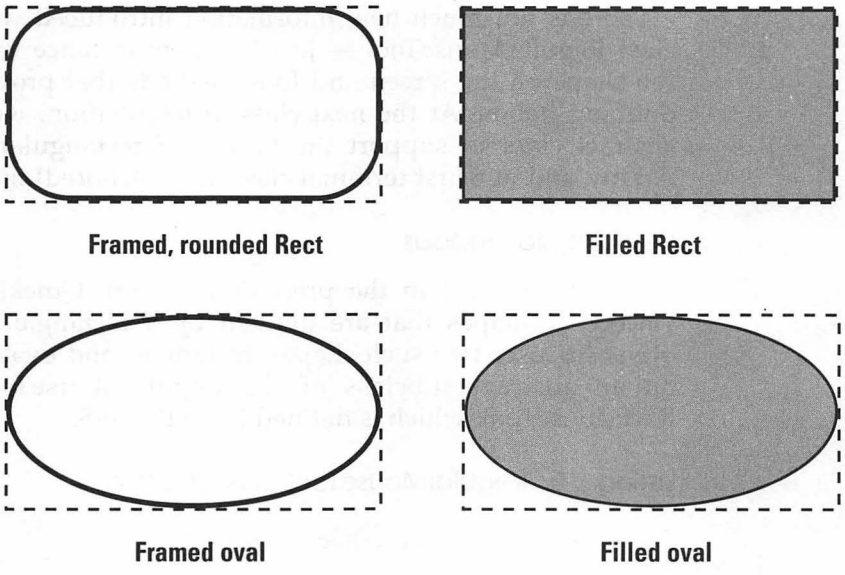


Figure 6-1. Regular shapes

Listing 6-4. RegularMouseTool class definition

```
struct RegularMouseTool : BasicMouseTool
{
  Point  anchorPt;          /* The defining points */
  Point  endPt;

  void Sketch(              /* Override for special sketching */
    Point anchorPt );

  void SetAnchorPt(         /* Access to defining points */
    Point thePt );

  Point  GetAnchorPt( void );

  void SetEndPt(
    Point  thePt );

  Point  GetEndPt( void );
};
```

There is not much new information introduced with the new subclass RegularMouseTool — just two new instance variables to define the shape on the screen and four methods that provide access to the defining points. At the next class differentiation, we'll define a new abstract class to support the family of rectangular shapes used in ArtToy, and our first terminal class, the AnchoredLineTool.

Rectangular Shapes

As we mentioned in the preceding section, QuickDraw supports a variety of shapes that are defined by a rectangle. ArtToy drawing offers the user two such shapes: rectangles and ovals. Both are based on an abstract subclass of the RegularMouseTool object called RectMouseTool, which is defined in Listing 6-5.

Listing 6-5. RegularMouseTool class structure

```
struct RectMouseTool : RegularMouseTool
{
  void Sketch(          /* sketch a rectangular shape */
    Point anchorPt );

  void MakeRect(
    Rect *resultR );    /* makes Rect from anchor and end points */
};
```

The definition of RectMouseTool doesn't include any new instance variables; the two points that describe all regular shapes are sufficient. What is new is the **Sketch**() method. It is introduced here, in an abstract superclass, because the same method — literally the same code — can sketch any rectangular shape.

The RectMouseTool class also needs a method, **MakeRect**(), that examines the *anchorPt*, and *endPt* instance variables and creates from them a Rect structure. It assures that the top of the rectangle is above the bottom and that the left side is to the left of the right side; the anchor point and the end point don't imply any such relationship between their respective coordinates.

The **Sketch**() method records the initial mouse coordinates as the object's "anchorPt" and then runs in a **WaitMouseUp**() loop watching the movement of the mouse. While tracking the mouse, it repeatedly draws, erases, and redraws the shape being sketched. When the user releases the mouse button, the **Sketch**() method records the final mouse coordinates as the "endPt," thus defining the regular object. Listing 6-6 shows the code for the RectMouseTool's **Sketch**() method.

Listing 6-6. RectMouseTool::Sketch() method

```
/*
** Sketching a rectangular object.
*/
void RectMouseTool::Sketch(
  Point  firstPt )
{
Point    oldMouse, newMouse;
Boolean  isDrawn;

  isDrawn = FALSE;
  oldMouse = newMouse = firstPt;
  this->SetAnchorPt( firstPt );
  while (WaitMouseUp())
  {
    /* Track the mouse */
    GetMouse( &newMouse );
    if (!EqualPt( newMouse, oldMouse ) &&
        PtInRect( newMouse, &(this->limitRect) ))
    {
      if (isDrawn)
        this->Draw( srcXor );     /* Erase the old one */
      this->SetEndPt( newMouse );
      this->Draw( srcXor );       /* Draw new one */
      oldMouse = newMouse;
      isDrawn = TRUE;
    }
  }

  /* Erase the last one drawn */
  if (isDrawn)
    this->Draw( srcXor );
}
```

Notice that the last thing the Sketch() method does is erase whatever it finally drew. This method handles only the sketching of the rectangular shape. The Sketch() method exploits the fact that a shape drawn twice in patXor copy mode is effectively drawn and erased. The final rendering of that shape, which is left to another method, is usually handled in the following way:

1. Render the shape into the offscreen representation of the document.
2. Invalidate the portion of the window where the shape is to appear. This generates an update event.

3. The application "pretends" that it's finished with drawing the shape. However, it will receive an update event, at which point it will transfer the offscreen shape into the window. The time between steps 2 and 3 is usually quite short and often imperceptible to the user.

Notice also that the Sketch() method assumes that it is being called after the mouse button has been pressed. Its job is to do its thing while the button is held down. It should be called from mouseDown event processing.

The RegularScreenObject's Sketch() method is the first place that you realize direct benefit from using the object-oriented programming approach. The object's Draw() method is called for all drawing; in fact, the Draw() method is the only method that contains any QuickDraw calls. The Sketch() method does only its specific job: It handles the user interaction for sketching a rectangular object. Each specific rectangular object tool will define its own Draw() method.

With the Sketch() in place, you're ready to define the terminal rectangular shape drawing objects for use in ArtToy. Listing 6-7 defines the framed rectangle and framed oval shape drawing tools.

Listing 6-7. RectangleTool and OvalTool class definitions

```
struct RectangleTool : RectMouseTool
{
  void Draw(          /* Draws a rectangle in the current port */
    short transferMode );
};

struct OvalTool : RectMouseTool
{
  void Draw(          /* Draws an oval in the current port */
    short transferMode );
};
```

At this point of object differentiation, all we need to distinguish between the rectangular shape drawing tools is the **Draw**() method: That's all that is unique about these objects. Listing 6-8 shows the **Draw**() methods themselves. You can see how simple the methods have become because all they do is encapsulate the QuickDraw traps that draw the specific shape in the current port. All of the information

they need for the drawing they do is in the two regular shape-defining points (the instance variables of the RegularMouseTool class) and the copy mode that was sent in by the caller.

Listing 6-8. RectangleTool::Draw() and OvalTool::Draw() methods

```
/*
** Draw a rectangle in current port.
*/
void RectangleTool::Draw(
     short  transferMode )
{
PenState    savedPen;
Rect        targetR;

  GetPenState( &savedPen );   /* Remember current pen mode */
  PenMode( transferMode );    /* Set caller's pen mode */
                              /* preference */
  this->MakeRect( &targetR ); /* Create Rect from anchorPt, */
                              /* endPt */
  FrameRect( &targetR );      /* Draw the rectangle */
  SetPenState( &savedPen );   /* Restore original pen mode */
}


/*
** Draw an oval in current port.
*/
void OvalTool::Draw(
  short  transferMode )
{
PenState    savedPen;
Rect        targetR;

  GetPenState( &savedPen );   /* Remember current pen mode */
  PenMode( transferMode );    /* Set caller's pen mode */
                              /* preference */
  this->MakeRect( &targetR ); /* Create Rect from anchorPt, */
                              /* endPt */
  FrameOval( &targetR );      /* Inscribe the oval */
  SetPenState( &savedPen );   /* Restore original pen mode */
}
```

You see more similarities than differences between the two methods in Listing 6-8; they differ only in the substitution of **FrameOval**() for **FrameRect**().

With the rectangular shape-drawing objects fully defined, let's look at how one of them, RectangleTool, is used by ArtToy.

Using the RectangleTool Object

Once the mouseDown event processing in ArtToy has determined that the cursor is in the content region of the frontmost document window and that the current mouse tool is the framed rectangle drawing tool, it is ready to use the RectangleTool object. It uses a specific version of the drawing tool use algorithm, which was presented in Listing 6-3. In this example, offScreenWorld is the front window's offscreen drawing environment and localMouse is the mouse position converted from the EventRecord into the front window's local coordinates.

Listing 6-9 revisits the algorithm used in the mouseDown processing routine.

Listing 6-9. MouseDown processing example for rectangle shape

```
enum
{
  PencilToolID  = 1,
  RectToolID    = 2,
  MarqueeToolID = 3
};

extern short  gCurrentToolID;

extern GWorldPtr GetDocGWorld(
  WindowPtr   theWindow );


void HandleMouseDown(
  Point     globalMouse )
{
RectangleTool *aDrawTool;
GDHandle      savedGDev;
GrafPtr       savedPort;
WindowPtr     theWindow;
Point         localMouse;
short         windowPart;
```

```
windowPart = FindWindow( globalMouse, &theWindow );
if (theWindow != FrontWindow())
  return;

switch (windowPart)
{
  case inContent:
    switch (gCurrentToolID)
    {
      case RectToolID:
        /* Determine the window's off-screen buffer */
        /* •••Note: application-defined routine! */
        offScreenWorld = GetDocGWorld( theWindow );
        if (offScreenWorld == NULL)
          return;

        /* Set up for sketching */
        GetGWorld( &savedPort, &savedGDev );
        SetPort( theWindow );
        localMouse = globalMouse;
        GlobalToLocal( &localMouse );

        /* Create and initialize new drawing object */
        aDrawTool = new( RectangleTool );
        aDrawTool->Init();

        /* Let the user go nuts */
        aDrawTool->SetLimitRect(
          &(((GrafPtr)theWindow)->portRect) );
        aDrawTool->Sketch( localMouse );

        /* Render the rectangle off-screen */
        SetGWorld( offScreenWorld );
        aDrawTool->Draw( srcCopy );

        /* Object has done its job: get rid of it */
        aDrawTool->ObjKill();
        delete( aDrawTool );

        /* Force an update for this window */
        InvalRect( &(((GrafPtr)theWindow)->portRect) );
        SetGWorld( savedPort, savedGDev );
        break;
```

```
            default:
              break;
          }
      }
  }
```

One thing that the code in Listing 6-9 makes apparent is the reason that the **Draw**() method accepts a copy mode as an argument. It lets the same method — the encapsulation of the QuickDraw rendering calls dedicated to the object's shape — be used for both sketching (with patXor) and drawing (with patCopy).

Figure 6-2 is a representation of what happens on the screen during the sketching activity — while the mouse button is held down. The user points the cursor at the anchor point and presses the button. With the button held down, the cursor is dragged along the path to the end point, where the mouse button is released. Along the way, at every iteration of the **WaitMouseUp**() loop in the **Sketch**() method, one after another rectangle is drawn in the window. Before each of these intermediate rectangles (represented with dotted lines in Figure 6-2) is drawn, the preceding one is erased.

When the mouse button is finally released at the end point, the last of the intermediate rectangles is erased. The final rectangle, defined by the anchor point and the end point, is drawn into the window's offscreen GWorld. During update event processing, the offscreen version of the window contents is rendered on the screen.

Note ▶

The final rendering of the window's contents on the screen is done during update processing using **CopyBits**(). RectangleTool class is not only *not* used for this purpose, it doesn't exist during update processing because it was deleted immediately after the sketching process.

It is easy to see how the great similarities among the rectangle-based drawing tools can be exploited to create different shape-drawing tools from objects that differ from one another in just one line of code (out of hundreds) in the **Draw**() method.

In the next section we'll look at the anchored-line drawing tool, which differs only slightly from the rectangle-based drawing tools.

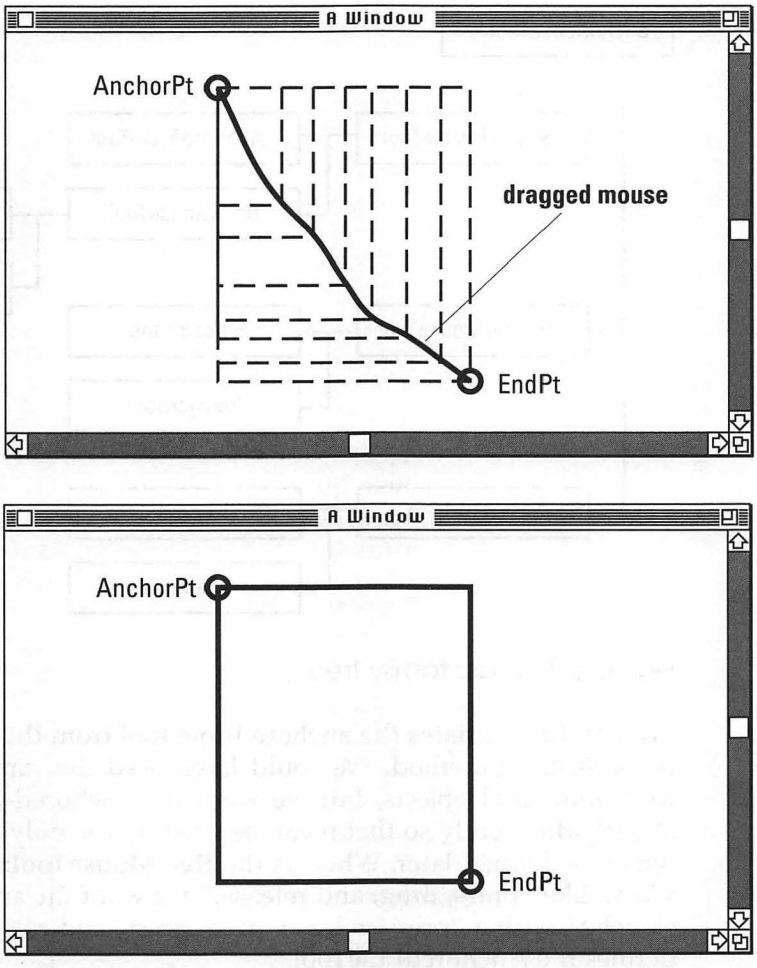


Figure 6-2. Drawing a rectangle

The Anchored-Line Drawing Tool

If you look at the family tree for the ArtToy drawing classes (Figure 6-3), you see that the AnchoredLineTool branches off from all of the rectangular shape drawing tools even though it is a "regular" shape — one defined by two points in the QuickDraw coordinate plane.

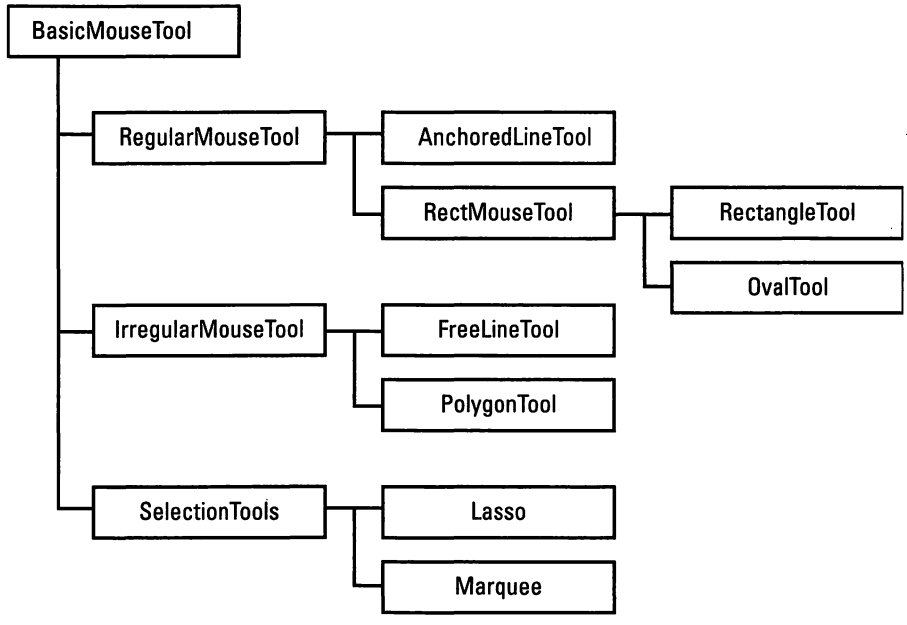


Figure 6-3. Class family tree

What differentiates the anchored-line tool from the rectangular tools is its Sketch() method. We could have used the same method as the RectMouseTool objects, but we want the anchored-line tool to work slightly differently so that it can be used by the polygon drawing tool, which is defined later. Whereas the **RectMouseTool::Sketch()** method works like "press, drag, and release," we want the anchored line to be sketched with a "press, release, drag, press, and release." Listing 6-10 defines the AnchoredLineTool.

Listing 6-10. AnchoredLineTool class structure

```
struct AnchoredLineTool : RegularMouseTool
{
  void Sketch(              /* They get sketched their own way */
    Point anchorPt );

  void Draw(                /* Draws the line */
    short transferMode );

  Point GetEndPt( void ); /* Returns line's endpoint */
};
```

As we said, the AnchoredLineTool has its own Sketch() method. And, as with all of the terminal-class drawing tool objects, it has its own Draw() method, the method that gives the tool its shape. There are no new instance variables because, like the rectangle-based shapes, the anchored line can be described by two points.

In order to facilitate the "press, release, drag, press, and release," means of sketching, you have to make a couple of minor but crucial alterations to the RectMouseTool sketching code, as shown in Listing 6-11.

Listing 6-11. Anchored-line sketching

```
/*
**Lets the user sketch an anchored line.
*/
void AnchoredLineTool::Sketch(
  Point  anchorPt )
{
EventRecord   mouseEvent;
Point         newMouse, oldMouse;
Boolean       isDrawn;
long          tempLong;

  isDrawn = FALSE;
  oldMouse = anchorPt;
  while (WaitMouseUp() ||
    !WaitNextEvent( mDownMask, &mouseEvent, 0L, NULL ))
  {
    /* Track the mouse, while staying within the limit */
    GetMouse( &newMouse );
    tempLong = PinRect( &(this->limitRect), newMouse );
    newMouse = *(Point*)&tempLong;

    if (!EqualPt( newMouse, oldMouse ))
    {
      if (isDrawn)
        this->Draw( srcXor );     /* Erase previous line */
      this->SetEndPt( newMouse ); /* Define new line */
      this->Draw( srcXor );       /* Draw new line */
      oldMouse = newMouse;
      isDrawn = TRUE;
    }
  }

  /* Erase last line */
  if (isDrawn)
    this->Draw( srcXor );
}
```

Two important things are being done to make the way the anchored line is sketched different from the way the rectangles were sketched. Notice the logical text that controls the execution of the dragging loop:

```
while (WaitMouseUp() || !GetNextEvent( mDownMask, &mouseEvent ))
{
}
```

Even after the user lets up on the mouse button, the loop continues to "drag" the line's end point around the screen until the mouse button is pressed a second time. It's still a very focused event loop in that, like the simpler **WaitMouseUp**() loop, it concentrates on the actions of the mouse button. But it differs in that it "drags" with the mouse button up.

The other significant difference is the use of **PinRect**() to qualify the candidate point's coordinates to be inside the object's limits. What this means is that, even though the cursor may be moved outside the limit rectangle, the point used for sketching will be that point on the boundary of the limit rectangle that is closest to the mouse.

We could just as easily have chosen to ignore the mouse when it roams outside the limit rectangle, just as we could have chosen "press, drag, and release" button control. But this provides an opportunity to make a point that applies to all of ArtToy's drawing objects: There is nothing sacred in the way these objects work. Defining them is a process that involves making a lot of arbitrary design decisions that can just as easily go a different way. Keep this in mind as you read these object definitions and methods.

One other subtle side effect results from using the algorithm in Listing 6-11. The while() condition will terminate the loop if **WaitNextEvent**() returns an event. The event mask that is being sent into **WaitNextEvent**() is intended to mask out all events except mouseDown, which is the only event that we want to terminate the sketching and define a line. But if the user clicks the mouse on another application's window, then the application will receive a suspend event in the form of an app4Evt, rather than the expected mouseDown event. This event type cannot be masked out. Being suspended while sketching an anchored line may be something that the caller of the Sketch() method should take into account.

Figure 6-4 is a representation of what the user sees when an anchored line is being sketched. As the mouse follows the path indicated by the heavy dotted line, one after another line is drawn and erased as represented by the lightly drawn dotted lines. When the mouse button

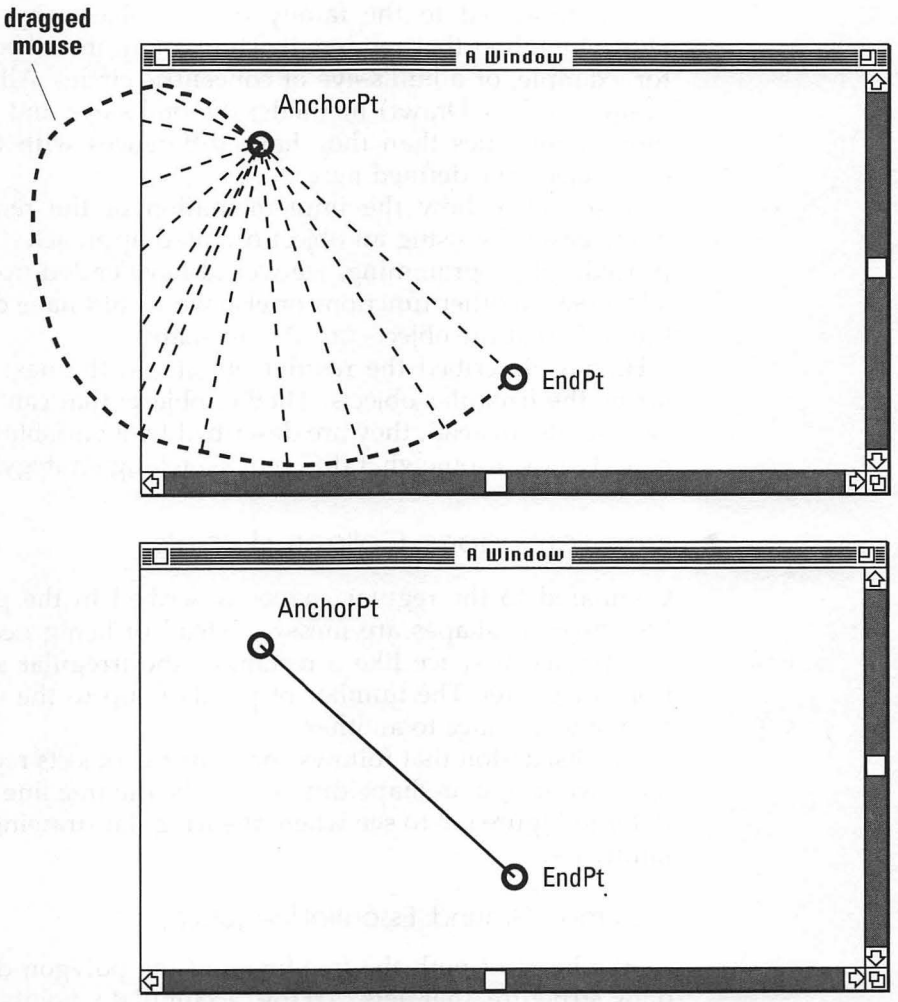


Figure 6-4. Sketching an anchored line

is pressed a second time, the final end point is accepted and the line is drawn. The code that ArtToy uses during mouseDown processing to draw an anchored line is the same as that presented in Listing 6-11 for drawing a rectangle except that, instead of allocating a new RectMouseTool, it allocates a new AnchoredLineTool. Notice how **PinRect**() tracks the mouse along the inside of the document window.

You should realize a couple of things about the objects and classes we defined for the regular tools. For one thing, any regular shape

could be added to the family of tool objects by simply adding a Draw() method that renders it. Think of an inscribed pentagonal star, for example, or a bull's-eye of concentric circles. All that makes them unique is their Draw() methods; the bull's-eye and the star share far more similarities than they have differences with the rectangle and oval objects we defined here.

Also realize how the implementation of the regular objects was made easier by using an object-oriented approach. If we used strictly procedural programming, we would have ended up passing function addresses to other functions or else we would have duplicated a lot of the code that the objects are able to share.

Having described the regular objects, in the next section we'll describe the irregular objects. They're objects that can't be described by two points. Instead, they are described by a variable number of points collected by a somewhat different sketching strategy.

▶ Irregular Shapes: Collected Points

Compared to the regular shapes described in the preceding section, the irregular shapes are messy. Instead of being neatly described by two points in space like a rectangle, the irregular shapes are collections of points. The number of points is up to the user and it varies from one instance to another.

The discussion that follows concerns the objects required to give the user two irregular-shape drawing tools: the free line and the polygon. Refer to Figure 6-3 to see where the irregular drawing objects fit in the family tree.

Common Ground: Essential Irregularity

At the heart of both the free-line and the polygon drawing tools is a data structure that lets ArtToy accumulate points by tracking the mouse as the user sketches. The basic irregular object, an abstract superclass that is defined in Listing 6-12, accumulates point values into a Handle.

The definition in Listing 6-12 provides the basic mechanism that supports both irregular objects. By adding just two different **Sketch()** methods, the free line and the polygon are endowed with their separate personalities.

Listing 6-12. IrregularMouseTool class definition

```
struct IrregularMouseTool : BasicMouseTool
{
  long     numPoints; /* How many points define this thing */
  Handle   ptBuffHdl; /* Heap storage for accumulated points */

  void Init( void );   /* Allocates handle and zeros numPoints */

  long GetNumPoints( void );     /* Returns number of points */

  Point GetPoint(                /* Returns a desired point */
    long   ptIndex );

  void SetPoint(                 /* Changes a point */
    long   ptIndex,
    Point  newPt );

  OSErr AddPoint(                /* Adds a new point */
                                 /* to accumulation */
    Point  newPt );

  void Draw(                     /* Connects the dots */
    short transferMode );

  RgnHandle   MakeRgn( void );   /* Creates region from */
                                 /* collected points */

  PolyHandle  MakePoly( void );  /* Polygon from accumulation */

  void ObjKill( void );          /* Free the point buffer */
};
```

The two instance variables necessary for the irregular-shape drawing tools are "numPoints," which is the number of points that were accumulated during sketching, and "ptBuffHdl," a Handle to the point buffer in the application heap. Contrast these two descriptive instance variables with the two Point instance variables used by the RegularMouseTool definition. In the case of the RegularMouseTool, you know that, by definition, a regular object can be described with two points in the document window. But, in the case of the IrregularMouseTool objects, you can't know ahead of time how many points it will take to describe the shape that the user sketches in the window. So all you can do ahead of time is be ready to accumulate the points in memory and keep track of how many points there are.

An object's point buffer is filled by its Sketch() method. In general, a Sketch() method tracks the mouse; each time it gets a new set of coordinates, it appends the new point onto the accumulated points and increments the counter. The Sketch() methods are discussed in detail in the following sections.

Compared to the regular shaped objects, the IrregularMouseTool requires a lot of methods for support. We look at them one at a time. Listing 6-13 shows the initialization method.

Listing 6-13. IrregularMouseTool::Init() method

```
void IrregularMouseTool::Init( void )
{
   /* Perform normal initializations */
   inherited::Init();

   /* No points in the buffer yet */
   /* Reserve an empty handle */
   this->numPoints = 0;
   this->ptBuffHdl = NewHandle( 0L );
}
```

The IrregularMouseTool declares its own Init() method but notice the first thing it does: It calls the inherited Init() method to initialize itself as a BasicMouseTool and then it sets its own instance variables to known values. This overrides its inherited method, but, because it invokes the inherited method, it's functionally a supplement to the old methods. The GetNumPoints() method shown in Listing 6-14 just returns value of the numPoints instance variable to the caller.

Listing 6-14. IrregularMouseTool GetNumPoints method

```
long IrregularMouseTool::GetNumPoints( void )
{
   return this->numPoints;
}
```

It exists because some ArtToy code will want to know the number of points in the object and you don't want that code to reference the instance variable directly. We'll stretch credibility here to make a point about object-oriented programming. Suppose that in the future, in order to reduce the size of the object by 4 bytes, you decide that instead of actually counting the number of points in numPoints, you

want to calculate it from the size of the Handle. After making the change and removing the numPoints instance variable from the object, all of the code in ArtToy that had been referring directly to numPoints would have to be changed. But the code that got the number of points in the object by calling the GetNumPoints() method could go right on making the same call, without knowing or caring that the method had been changed. This is an example of encapsulation and it's a practice worth developing into a habit.

The SetPoint() method, shown in Listing 6-15, coerces the point buffer's Handle to be treated as a reference to an array of Points. It then uses the indicated point number to subscript the array in order to get access to the place in the buffer where the desired point is stored. Note the sense of paranoia exhibited by SetPoint(): Before modifying memory it makes sure that the point index value makes sense with respect to the known number of points in the buffer and that the buffer is really at least as large as the index implies it is.

Listing 6-15. IrregularMouseTool SetPoint() method

```
void IrregularMouseTool::SetPoint(
  long   ptIndex,
  Point  newPt )
{
  if ((ptIndex < this->GetNumPoints()) &&
      (ptIndex >= 0) &&
      (GetHandleSize( this->ptBuffHdl ) >= (ptIndex+1) *
       sizeof( Point )))
  {
    /* Retrieve a point from the buffer. */
    ((Point*)(*this->ptBuffHdl))[ptIndex] = newPt;
  }
}
```

Listing 6-16 shows the SetPoint() method in its entirety.

Listing 6-16. IrregularMouseTool AddPoint() method

```
OSErr IrregularMouseTool::AddPoint(
  Point newPt )
{
Size     newSize;
OSErr    errStat;
```

```
  errStat = noErr;
  this->numPoints += 1;                    /* Count the point. */
  newSize =
    GetHandleSize( this->ptBuffHdl ) +
    (this->numPoints * sizeof(Point));
  SetHandleSize( this->ptBuffHdl, newSize );  /* Room for one */
                                              /* more point. */
  errStat = MemErr();
  if (errStat == noErr)
    this->SetPoint( this->numPoints-1, newPt ); /* Remember */
                                                /* coordinates. */
  return errStat;
}
```

The AddPoint() method is the main reason that the SetPoint() method exists. AddPoint() manages the object's point buffer. When called, it increments the point counter and resizes the point buffer to make room for one more point. Then it calls the SetPoint() method to add the new point to the end of the buffer.

We hope that you're getting a sense of how the methods are being used to do just one job each. For example, AddPoint() does all of the memory allocation, and SetPoint() is the only method that changes the values in the buffer. By keeping the methods simple and focused, they are more easily used by the object's subclasses as building blocks to achieve various kinds of functionality. "So when do we draw something?" you might ask. Listing 6-17 presents a method for that, too.

Listing 6-17. IrregularMouseTool::Draw() method

```
/*
**Connect the dots.
*/
void IrregularMouseTool::Draw(
  short  transferMode )
{
PenState    savedPen;
Point       curPt;
long        ptCount, numPts;

  GetPenState( &savedPen );
  PenMode( transferMode );

  curPt = this->GetPoint( 0 );     /* Get the first point */
  MoveTo( curPt.h, curPt.v );
  numPts = this->GetNumPoints();
```

```
  /* draw lines to each point in succession */
  for (ptCount=1; ptCount<numPts; ptCount++)
  {
    curPt = this->GetPoint( ptCount ); /* Get the point */
    LineTo( curPt.h, curPt.v );        /* Draw a line from */
                                       /* last pt */
  }

  SetPenState( &savedPen );
}
```

The Draw() method reads points out of the buffer one at a time and draws lines between them. The lines are drawn into the current port in the current pen style. The Draw() method lets the caller give it a pen mode to use so that it can be used both in the Sketch() method, which tells it to use patXor, and by the ArtToy mouseDown event handling code, which calls it with patCopy to render the sketched shaped into an offscreen GWorld.

It might seem odd that two different drawing tools share the same Draw() method, and that the Draw() method is defined in an abstract object. It worked out this way because the two objects are fundamentally the same: Both the free-line tool and the polygon tool are drawn as a series of connected points. What sets them apart is the way the user defines the points, that is, the Sketch() method. Contrast this with the way the rectangular objects interacted. They used the same Sketch() method for user interaction but differentiated themselves in their Draw() methods.

The ObjKill() method, a standard method inherited by all descendants of the BasicMouseTool class, is overridden in the IrregularMouseTool (see Listing 6-18). Like the Init() method, ObjKill() takes an action peculiar to the irregular objects before invoking the ancestral method.

Listing 6-18. IrregularMouseTool Draw() method

```
void IrregularMouseTool::ObjKill( void )
{
  if (this->ptBuffHdl != NULL )
    DisposHandle( this->ptBuffHdl );

  inherited::ObjKill();
}
```

The ObjKill() method frees up the point buffer by disposing of the handle that was allocated in the object's Init() method.

The last method defined for IrregularMouseTool exists in anticipation of the drawing objects being used by selection tools. It is MakeRgn(), as shown in Listing 6-19. It uses the accumulated points in the drawn shape to describe an area in the window. MakeRgn() creates a QuickDraw region and returns its RgnHandle. The selection tools, which are described in a later section, use the shape-drawing objects to let the user sketch the area of the screen to be selected. Then they use the drawn shape and convert it to a region.

Listing 6-19. IrregularMouseTool::MakeRgn() method

```
/*
**Use accumulated points to create a region.
*/
RgnHandle IrregularMouseTool::MakeRgn( void )
{
Rect          polyLimitR;
Point         curPenSize;
PolyHandle    resultPoly;
RgnHandle     resultRgn;
GWorldPtr     sketchingGW;
GDHandle      savedGD;
CGrafPtr      savedPort;
PixMapHandle  gwPixels;
OSErr         errStat;

  if (this->GetNumPoints() < 1)
    return NULL;

  GetGWorld( &savedPort, &savedGD );
  curPenSize = savedPort->pnSize;

  /* create a polygon of enclosed pixels and get its bounding box */
  /* Note: this presumes that the ::Draw() method */
  /* uses LineTo() calls! */
  resultPoly = OpenPoly();
  this->Draw( srcCopy );
  ClosePoly();

  /* correct the bounding box for the pen size */
  polyLimitR = (**resultPoly).polyBBox;
  polyLimitR.right += curPenSize.h;
  polyLimitR.bottom += curPenSize.v;

  /* create a 1-bit GWorld in which to accumulate a pixel image */
  errStat = NewGWorld( &sketchingGW, 1,
                       &polyLimitR, NULL, NULL, (GWorldFlags)0 );
```

```
  if ((errStat != noErr) || (sketchingGW == NULL))
  {
    KillPoly( resultPoly );
    return NULL;
  }

  /* prepare for offscreen drawing */
  gwPixels = GetGWorldPixMap( sketchingGW );
  if (!LockPixels( gwPixels ))
  {
    KillPoly( resultPoly );
    return NULL;
  }
  SetGWorld( sketchingGW, NULL );

  /* start with a clean pixel image */
  /* fill the area described by the enclosed pixels */
  /* add the "sketched" pixels */
  EraseRect( &polyLimitR );
  FillPoly( resultPoly, black );
  FramePoly( resultPoly );

  /* the GWorld's pixel image now describes both the enclosed */
  /* and lower-right pixels, so create a region from it */
  resultRgn = NewRgn();
  errStat = BitMapToRegion( resultRgn, (BitMap*)*gwPixels );
  if (errStat != noErr)
  {
    DisposeRgn( resultRgn );
    resultRgn = NULL;
  }

  /* clean up */
  KillPoly( resultPoly );
  DisposeGWorld( sketchingGW );
  SetGWorld( savedPort, savedGD );

  return resultRgn;
}
```

This method makes use of the **BitMapToRegion**() trap, which generates a region from a 1-bit PixMap or BitMap. This trap is described in greater detail in Chapter 7.

You recall that the BasicMouseTool also has a MakeRgn() method. It works very simply by opening a new region, calling the object's Draw() method, and then closing the region. It turns out that that isn't

sufficient for the kind of results we want from the irregularly drawn selection tools, which are based on the basic drawing tools.

If you were to use the basic method for creating a region, the resulting region would be drawn inside of the points that were connected to form it. But the IrregularMouseTool's Sketch() method uses a series of calls to **LineTo()** to connect the points. You know from earlier chapters that QuickDraw draws into pixels below and to the right of the points it's drawing. For a given closed region, if you use the same **LineTo()** traps to create a region that you used while sketching, the sketched pixels below and to the right of the region's boundary points would be excluded from the region.

Figure 6-5 is a magnified view of the pixels, points, and grid lines near the lower-right boundary of a closed region, which was created by connecting points A, B, and C. The crosshatched area represents QuickDraw's mathematical definition of the region that would be created by connecting points A, B, and C with calls to **LineTo()**; the gray pixels would define the region's boundary. But the black pixels are the ones that the user saw while sketching, which was also done by calling of **LineTo()**. This creates a problem: In the interest of WYSIWYG and consistency, you don't want to let the user sketch a line only to have the visible line excluded from the resulting region. This problem can be solved by creating a region that includes the pixels enclosed in the sketch, drawing that region in an offscreen pixel image, and then duplicating the sketched **LineTo()** calls into the same pixel image. After the region and the sketch lines are combined in an offscreen pixel image, that pixel image can be used to create a region (**BitMapToRgn()**) that contains both the enclosed and visibly sketched pixels.

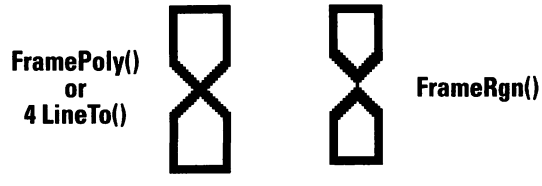


Figure 6-5. Making a rgn with FramePoly() versus LineTo()

Listing 6-19, ArtToy's MakeRgn() method, shows how ArtToy solves the problem. It first creates an offscreen GWorld with a pixel depth of one bit; the PixMap associated with the offscreen GWorld is essentially a BitMap. Next it creates a region that includes the enclosed

pixels by calling its Draw() method between calls to **OpenRgn**() and **CloseRgn**(). The region is then used to color pixels in the offscreen GWorld (**FillRgn**()). After the region's pixels are in the GWorld the Draw() method is called again. This effectively duplicates the user's sketching and colors the pixels that were sketched but lay outside of the defined region. When the offscreen GWorld contains both the region and the sketched pixels its PixMap is used to create a region by calling **BitMapToRgn**(). The output of **BitMapToRgn**() is sent back to the caller after the temporary GWorld is disposed of. It's the calling routine's responsibility to properly dispose of the returned region. As you'll see in subsequent sections, the "caller" of the **MakeRgn**() methods is a selection object, which keeps the region as one of its instance variables and disposes of it when its version of **ObjKill**() is called; therefore, the threat of regions being left around and unaccounted for is mitigated by the encapsulation properties of the selection objects.

By the way, the problem solved by using multiple regions for the irregular shapes doesn't occur with the rectangular shapes. Their **Draw**() methods use the framing traps **FrameRect**() and **FrameOval**(), which draw inside of the resulting region so their WYSIWYG-ness is intact.

Free-Line Drawing

Having defined the abstract class IrregularMouseTool, using it to define a free-line drawing tool is relatively simple. As the definition in Listing 6-20 shows, creating a free-line drawing tool out of the IrregularMouseTool just requires a sketching method.

Listing 6-20. FreeLineTool class structure

```
struct FreeLineTool : IrregularMouseTool
{
  OSErr Sketch(
    Point  startPt,
    short  copyMode );
};
```

Listing 6-21 is the listing for the FreeLineTool's Sketch() method.

Listing 6-21. FreeLineTool::SketchPlusMode() method

```
void FreeLineTool::SketchPlusMode(
  Point    startPt,
  short    copyMode )
{
```

```
PenState savedPen;
Point    newMouse, oldMouse;

  GetPenState( &savedPen );
  PenMode( srcXor );

  oldMouse = startPt;
  this->AddPoint( startPt );
  MoveTo( startPt.h, startPt.v );

  while (WaitMouseUp())
  {
    GetMouse( &newMouse );
    if (!EqualPt( newMouse, oldMouse ) &&
        PtInRect( newMouse, &(this->limitRect) ))
    {
      LineTo( newMouse.h, newMouse.v );
      this->AddPoint( newMouse );
      oldMouse = newMouse;
    }
  }

  this->Draw( srcXor );  /* Erase sketch marks */
  SetPenState( &savedPen );
}
```

Listing 6-21 adds one new twist that the previous Sketch() methods didn't need: The caller sends a "transferMode" argument to it. Unlike the regular shape drawing tools, the SketchPlusMode() method used by the irregular shapes doesn't use the "draw-erase-redraw-erase" strategy of sketching. Instead, it follows the mouse around the screen, drawing all the time. As we mentioned while describing the IrregularMouseTool, the irregular-shape drawing tools are used both for drawing on the screen and as a way for the selection tools to describe an area in the document window.

When used for drawing, the tool needs to show what has been drawn, that is, the result of drawing. For this the SketchPlusMode() method should be using patCopy drawing mode. But when the free-line drawing tool is being used for selection, its SketchPlusMode() method should show the user *where*, not *what* he or she has drawn. In order to make the sketched line stand out visibly against the background, it should do its drawing in patXor mode. In order to use the same sketching code for both purposes, let the caller, who knows what's going on, pass in the desired pen mode.

You should notice, however, that the free-line SketchPlusMode() method is basically the same as the sketching method used by the regular shaped objects: The mouse is tracked to define the shape while its button is held down.

Polygon Drawing

As with free-line drawing, turning the abstract IrregularMouseTool object into an irregular polygon drawing tool requires only a new Sketch() method. The irregular polygon sketching method is unique in two ways. First, the sketching loop terminates not when the mouse button is released but when the polygon in the window is "closed." Second, the polygon sketching method uses another object, AnchoredLineTool, to do part of its sketching. Listing 6-22 is the definition of the PolygonTool class.

Listing 6-22. PolygonTool class structure

```
struct PolygonTool : IrregularMouseTool
{
  void Sketch(          /* Lets user sketch a polygon. */
    Point     startPt );
};
```

To the user, polygon sketching works by clicking on a starting point in the document window. From the starting point the mouse operates just as if the user were sketching an anchored line: A line anchored at the starting point follows the mouse until the user clicks the mouse button a second time. From the second point, a second anchored line is started that is terminated by a second mouse click. The process continues — the user draws one after another anchored line — until the polygon is closed. Figure 6-6 represents sketching an irregular polygon defined by six points.

Although we've mentioned the term "closed," we've been intentionally vague about just what it means. Closure occurs when the user has described an enclosed area, but how the user does that is up to the application. There are a number of equally good ways to let the user close a polygon; choosing the best one depends on the type of drawing called for. You could watch the line segments to see if one crosses another and consider that to be closure, or you could say that any mouse click outside of the window indicates closure. Double-click could be used as an indicator of closure. The list is long and there is no clear choice. In fact, more than one could be combined. The method we've chosen for closure is what we call the "close enough" method.

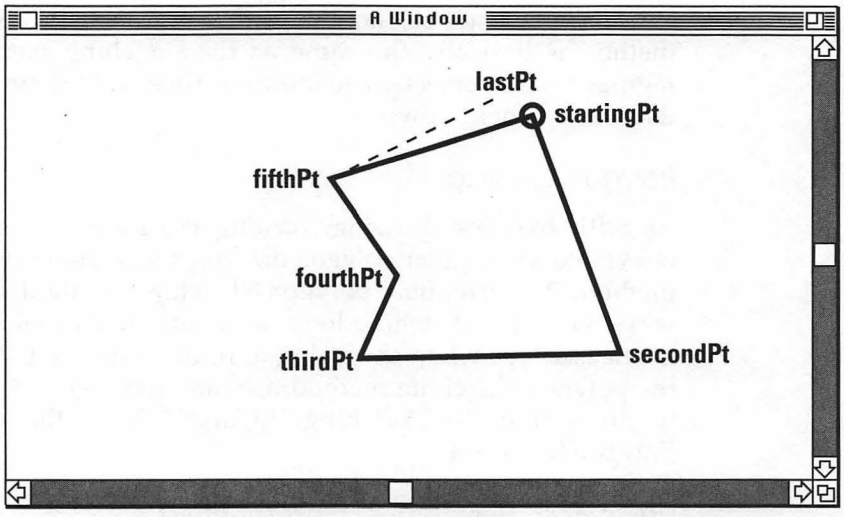


Figure 6-6. Sketching irregular polygon

The "close enough" method essentially says that the polygon being sketched is considered closed when the user clicks on the starting point a second time. But clicking on the point a second time is a lot to ask of the user, so we say that if the user gets within five pixels of the starting point, that's "close enough."

The example in Figure 6-6 shows the sixth point, labeled lastPt, which is close enough to the starting point. When the user clicked on the last point, the sketching software found it to be within the close enough distance and, instead of using the coordinates of the last point to draw the line (the dotted line), the software used the coordinates of the starting point for the ending point of the last line and terminated sketching. Listing 6-23 is the PolygonTool's Sketch() method.

Listing 6-23. PolygonTool::Sketch() method

```
/*
**Allow user to sketch a polygon.
*/
void PolygonTool::Sketch(
  Point  startPt )
{
AnchoredLineTool   *curLineSeg;  /* We'll need help sketching */
RgnHandle          threshRgn;
Rect               closeEnoughR;
```

```
Point         newPt;
long          pointsCount;
short         thresh;

  /* Define "close enough" and create a region */
  /* for testing for closeness to starting point */
  thresh = 5;
  SetRect(
    &closeEnoughR,
    startPt.h - thresh, startPt.v - thresh,
    startPt.h + thresh, startPt.v + thresh );
  threshRgn = NewRgn();
  OpenRgn();
  FrameOval( &closeEnoughR );
  CloseRgn( threshRgn );

  /* Allocate a new anchored line */
  curLineSeg = new( AnchoredLineTool );

  newPt = startPt;
  this->AddPoint( newPt );
  do
  {
    curLineSeg->Init();
    curLineSeg->Sketch( newPt );     /* User draws a line */
    curLineSeg->Draw( srcXor );      /* Draw link in */
                                     /* disappearing ink */

    newPt = curLineSeg->GetEndPt();
    this->AddPoint( newPt );      /* Accumulate points */
  }
  while (!PtInRgn( newPt, threshRgn ));

  /* dispose local data allocations */
  curLineSeg->ObjKill();
  delete( curLineSeg );
  DisposeRgn( threshRgn );

  /* Erase the last sketch */
  this->Draw( srcXor );

  /* Make last point equal to start point */
  pointsCount = this->GetNumPoints();
  this->SetPoint( pointsCount-1, startPt );
}
```

There are a few things to notice in the implementation of the **Sketch()** method. The area of the screen that is considered close enough to cause closure is described by a QuickDraw region. The region is constructed by inscribing an oval in a 10-by-10-pixel box centered on the starting point. Every new mouse point added to the polygon is used to test for closure before the loop reiterates.

During every iteration of the loop, a new AnchoredLineTool is created. This object is used to control the sketching of one line segment (the definition of a new vertex). The AnchoredLineTool is destroyed after it sketches a line whose end point is added to the accumulating vertex points.

After closure has been achieved, the last point, the one that closed the polygon, gets changed. Its value is reset to be that of the starting point so that when the polygon is drawn with the IrregularMouseTool::Draw() method, true closure occurs.

▶ Selection Tools

Along with drawing, the other interesting thing the user can do with an ArtToy mouse tool is to select an area in the document window. We call this process *selection*. The idea is that the user sketches an area of the screen in the same way that he or she might sketch with one of the drawing tools. The sketched area becomes highlighted with an animated boundary and it can then be cut, copied, or otherwise manipulated. The rest of this chapter deals with selection and the tools and techniques selection requires.

We'll discuss two selection tools in detail: the lasso selection tool and the rectangular marquee (or just marquee). They are both derived from the BasicMouseTool class by way of an abstract object that includes what all selection tools have in common. The common ancestor for the selection tools is the abstract superclass SelectionTool, which is defined in Listing 6-24. We'll describe some of the simple methods now, but the DragRgn() method and the methods for animating the selection will mean more after we describe the specific selection tools.

Listing 6-24. SelectionTool class definition

```
struct SelectionTool : BasicMouseTool
{
  RgnHandle   theRgn;     /* Selected area */
  GWorldPtr   rgnCopy;    /* Off-screen copy of selected area */
  short       currPattIndex;  /* index of current animating */
                              /* pattern */
```

```
  short       pattCount;  /* How many patterns are used */
  short       pattResID;  /* Recource ID of animating patterns */


  void Init( void );          /* Initialization */

  Boolean DragRgnOutline(     /* Facilitate mouse drag of */
                              /* outline */
    Point*      startPt,      /* Initial mouse location */
    GWorldPtr   windDataGW ); /* Offscreen window pixel buffer */

  Boolean DragRgnPixels(      /* Facilitate mouse drag of */
                              /* PixMap */
    Point*      startPt,      /* Initial mouse location */
    GWorldPtr   windDataGW ); /* Offscreen window pixel buffer */

  Boolean PtInSelection(      /* Tests point in selected area */
    Point thePt );

  Boolean SelectionEmpty( void ); /* Tests for empty */
                                  /* selection area */

  void AnimateSelection( void );  /* Run ants around */
                                  /* selected area */

  void FrameSelection(            /* Draw ants */
    Pattern   thePatt,
    short     transferMode );

  void RemoveAnimation( void );   /* Remove ants */

  void ObjKill( void );           /* Destroy the selection */
};
```

All selection objects need the three instance variables in Listing 6-24. The *theRgn* instance variable is a QuickDraw region. It holds the definition of the selected area of the window. The *rgnCopy* variable points to an offscreen buffer to hold a copy of the pixels in the selected region. It can be used for pixel-based manipulations such as dragging, rotating, and filtering. The last of the instance variables, *currPattIndex*, tells the object which of the region boundary animation patterns is currently used on the screen.

The Init() method, shown in Listing 6-25, sets the instance variables to a known state after first calling the inherited method.

Listing 6-25. SelectionTool::Init() method

```
/*
**Basic initialization.
*/
void SelectionTool::Init( void )
{
  inherited::Init();

  theRgn = NULL;
  rgnCopy = NULL;
  currPattIndex = (-1);
}
```

The PtInSelection() and SelectionEmpty() methods both perform simple Boolean tests. They are shown in Listing 6-26.

Listing 6-26. SelectionEmpty() and PtInSelection() methods

```
/*
**Tests for an empty selection.
*/
Boolean SelectionTool::SelectionEmpty( void )
{
  if (this->theRgn == NULL)
    return FALSE;
  else
    return EmptyRgn( this->theRgn );
}

/*
**Tests a point for membership in the selection.
*/
Boolean SelectionTool::PtInSelection( Point thePt )
{
  if (this->SelectionEmpty())
    return FALSE;
  else
    return PtInRgn( thePt, this->theRgn );
}
```

A Rectangular Marquee

The rectangular marquee lets the user sketch a rectangle that is then used to define a rectangular region in the document window. It's pretty simple to turn the abstract SelectionTool object into a marquee selection tool. The definition of the marquee object is shown in Listing 6-27.

Listing 6-27. MarqueeSelection class definition

```
/*
**Rectangular selection tool.
*/
struct MarqueeSelection : SelectionTool
{
  void Sketch(
    Point  startPt );
};
```

All that is required to create the MarqueeSelection object is to add a Sketch() method to the SelectionObject. The Selection() method is shown in Listing 6-28.

Listing 6-28. MarqueeSelection::Sketch() method

```
void MarqueeSelection::Sketch(
  Point  startPt )
{
RectMouseTool *rectSketcher;

  rectSketcher = new( RectMouseTool );
  rectSketcher->Init();

  rectSketcher->Sketch( startPt );
  this->theRgn = rectSketcher->MakeRgn();

  rectSketcher->ObjKill();
  delete( rectSketcher );
}
```

The Sketch() method creates a rectangle drawing tool, calls the rectangle tool's Sketch() method, and then has the rectangle tool create a region from its rectangle. And, *voilà*, a rectangular region is defined. Notice that the region is saved in the *theRgn* instance variable before the rectangle drawing tool object is disposed of.

Creating a marquee selection tool from the basic selection tool object was pretty straightforward. All selection objects are basically the same: They use a drawing object to let the user sketch a shape that is then used to define a region.

The Lasso

Like the MarqueeSelection object, the LassoSelection object is based on a predefined drawing tool, the free line. But there's a kink or two added to the lasso.

Everyone has used the lasso in MacPaint. You sketch a free line on the screen and the resulting selection is composed of all of the black pixels inside the lasso. It was a great idea; variations on it can be found in most graphic design and paint programs.

The way the lasso works in a black-and-white environment is easily understood. But how a lasso should work in a color environment is less clear. Let's briefly look at how the monochrome lasso works and then discuss some issues that arise when we take the lasso to go after a horse of a different color. We'll end up defining a lasso object that ArtToy can live with.

A Monochrome Lasso. Figure 6-7 represents a lasso sketched in a monochrome environment. The sketch is made up of eight connected points labeled A through H. When the user sketches the lasso in Figure 6-7 and then releases the mouse button, the lasso is constricted to select the crosshatched area shown in Figure 6-8.

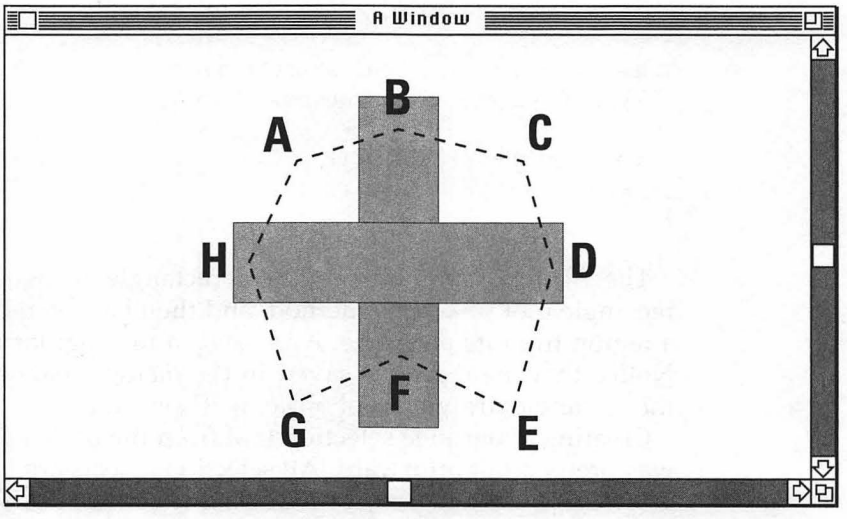


Figure 6-7. Black-and-white lasso sketch

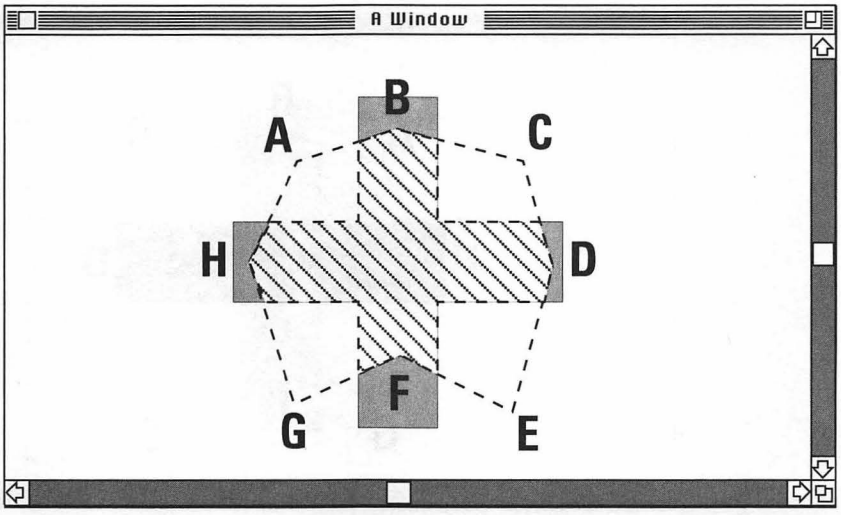


Figure 6-8. Constricted black-and-white lasso

The mechanism for constricting the lasso is a QuickDraw trap, **SeedFill**(). Given a starting point, **SeedFill**() returns a BitMap that represents where "paint could flow" in the image from the starting point. (A much simpler use of **SeedFill**() is shown in the paint bucket tool.) But consider Figure 6-8. The way to create the crosshatched region from the sketched region is to subtract from the sketched region the regions created by using **SeedFill**() (spilling paint) at each of the points in the sketch. It's pretty straightforward because paint spilled at the points in the black areas of the screen result in empty regions. But what happens when there are more than two colors to consider? Does paint flow as easily through red as it does through white? Is black the only color that can stop the spill? What color is the paint?

Lassoing Deep Pixels. The answers to the questions in the preceding section are elusive — there is no clear right way to handle the situation. To implement the lasso the developer must make an essentially arbitrary decision.

Figure 6-9 is the same sketched lasso outline as in Figure 6-7, but now it is lassoing color pixels.

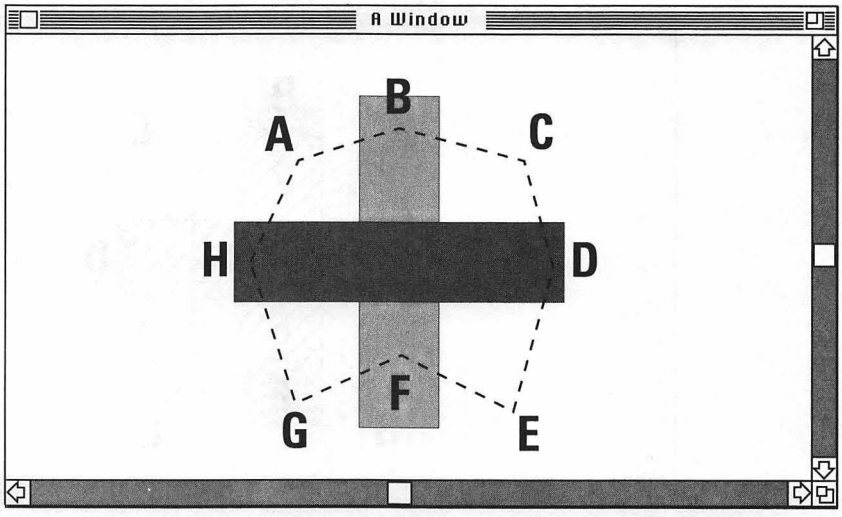


Figure 6-9. Color lasso sketch

Color QuickDraw provides an upgraded tool for tracking spilled paint:

```
void SeedCFill(
  BitMap    *srcBits,
  BitMap    *dstBits,
  Rect      *srcRect,
  Rect      *dstRect,
  short     seedH,
  short     seedV,
  ProcPtr   matchProc,
  long      matchData );
```

The first argument is the PixMap into which the paint is spilled. The second argument is a BitMap; **SeedCFill**() places a 1 in the bits that correspond to where the paint leaked. The BitMap is intended to be used as a mask in other operations. The rectangles let you restrict and target the spill. The seed values define the origin of the spill. The paint can leak to any adjacent pixel of the same RGB value as the pixel at the seed point. The last two arguments allow for customizing how **SeedCFill**() decides on where the paint can leak.

To perform a constricting algorithm, start with the whole region inside the sketch, which you get from the free-line sketching tool that you used to define the region. After constricting the lasso, you want to end up with the areas labeled 2, 4, and 6 in Figure 6-10.

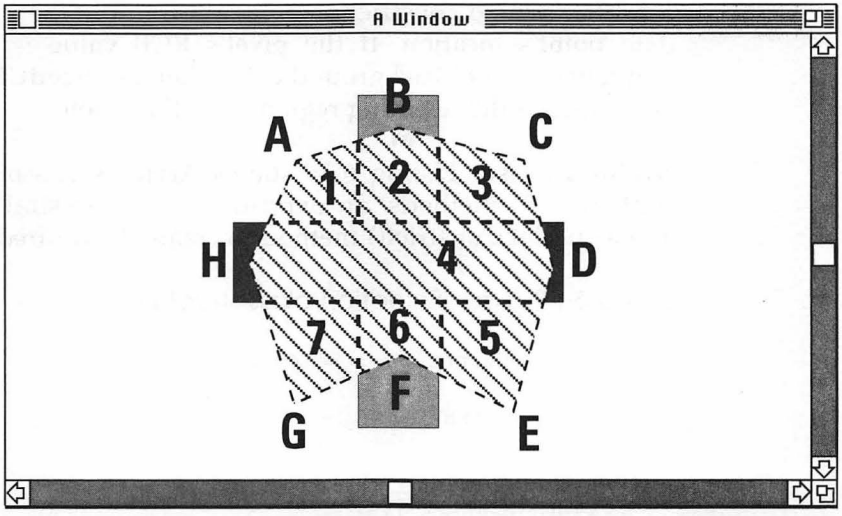


Figure 6-10. Constricted color lasso

Explore what happens as you go constricting from point to point around the sketch. Call **SeedCFill**() using the pixel value at point A as the seed RGB value. The resulting BitMap corresponds to the area labeled 1 in Figure 6-10. Convert the BitMap to a region and call it region 1. Subtract region 1 from the whole.

The next point is B. If you call **SeedCFill**() using point B, then after calling **BitMapToRgn**() to get a usable masking region, you have the region labeled 2 in Figure 6-10. But region 2 is to be left as part of the result. What's the difference between point 1 and point 2 that makes you skip point 2 while constricting? How you evaluate each candidate point depends on how you want the constricting to work.

Here's where the decisions have to be made. For your purposes, what distinguishes point A from point B is that the pixel at A is the background color of your drawing. But you could easily have chosen to accept A and reject B because A is the starting point of the user's sketch. The user thought A was special and told you so by clicking on it to start the sketch.

There are more variations on constricting that produce different effects than the one you chose. You could sample the pixel at A and determine a range of RGB values that are "leaky" with respect to your "paint." Then you could send **SeedCFill**() a special "matchProc" argument that looks, not for an exact match, but for pixels falling within the range of colors.

ArtToy's lasso constriction rule is to sample the pixel at the candidate point's location. If the pixel's RGB value is the same as the window's current background color, then use **SeedCFill()** at that point and subtract the resulting region from the whole.

ArtToy's Lasso. Listing 6-29 shows ArtToy's LassoSelection class. It contains two methods: an override of the ancestral Sketch() method and a special Constrict() method to create the desired region.

Listing 6-29. LassoSelection class structure

```
struct LassoSelection : SelectionObject
{
  FreeLine    *theLine;

  void Sketch(
    Point  startPt );

  void Constrict( void );
};
```

The Sketch() method is shown in Listing 6-30.

Listing 6-30. LassoSelection Sketch() method

```
void LassoSelection::Sketch(
  Point  startPt )
{
long  numPts;

  this->theLine = new( FreeLine );
  this->theLine->Init();

  this->theLine->SketchPlusMode( startPt, srcXor );
  numPts = this->theLine->GetNumPoints();
  this->theLine->SetPoint( numPts-1, startPt );
  this->Constrict();

  this->theLine->ObjKill();
  delete( this->theLine );
}
```

The Sketch() method creates a free-line drawing tool and then uses it for sketching. Notice that the sketching is done in patXor mode so that

all of the points being sketched will show up in the window. After the user is done sketching, the final point in the sketch is set to be equal to the first point so that the shape represents a closed form. Before discarding the drawing tool, the lasso uses the accumulated points held in the drawing to constrict the shape. Listing 6-31 is the lasso's Constrict() method.

Listing 6-31. LassoSelection::Constrict() method

```
void LassoSelection::Constrict( void )
{
RGBColor      targetRGB, curRGB;
RgnHandle     fillingRgn;
GWorldPtr     rgnBitsWorld;
CGrafPtr      savedPort;
GDHandle      savedGDev;
PixMapHandle  currentPM, rgnPM;
Rect          fillingBoundsR;
Point         curPt;
OSErr         errStat;
long          ptIndex, totalPts;

  /* Create the starting region */
  this->theRgn = this->theLine->MakeRgn();
  if ((this->theRgn == NULL) || EmptyRgn( this->theRgn ))
    return;

  GetGWorld( &savedPort, &savedGDev );
  currentPM = savedPort->portPixMap;
  errStat =
    NewGWorld(
      &rgnBitsWorld, 1, &(savedPort->portRect),
      NULL, NULL, 0L );
  if (errStat != noErr)
  {
    DisposRgn( this->theRgn );
    this->theRgn = NULL;
    return;
  }

  /* set up for point by point exclusion */
  SetGWorld( rgnBitsWorld, NULL );
  rgnPM = GetGWorldPixMap( rgnBitsWorld );
  (void)LockPixels( rgnPM );
```

```
    totalPts = this->theLine->GetNumPoints();
    RGBBackColor( &targetRGB );

    for (ptIndex=0; ptIndex<totalPts; ptIndex++)
    {
      /* Bypass redundant pixels */
      curPt = this->theLine->GetPoint( ptIndex );
      if (PtInRgn( curPt, this->theRgn ))
      {
        GetCPixel( curPt.h, curPt.v, &curRGB );
        if ((curRGB.red == targetRGB.red) &&
            (curRGB.green == targetRGB.green) &&
            (curRGB.blue == targetRGB.blue))
        {
          fillBoundsR = (**(this->theRgn)).rgnBBox;
          SeedCFill(
            (BitMap*)*currentPM, (BitMap*)*rgnPM,
            &fillBoundsR, &fillBoundsR,
            curPt.h, curPt.v, NULL, 0 );
          fillingRgn = NewRgn();
          errStat = BitMapToRegion( fillingRgn, (BitMap*)*rgnPM );
          if ((errStat == noErr) && !EmptyRgn( fillingRgn ))
            DiffRgn( this->theRgn, fillingRgn, this->theRgn );
          DisposeRgn( fillingRgn );
        }
      }
    }

    DisposeGWorld( rgnBitsWorld );
    SetGWorld( savedPort, savedGDev );
}
```

The **SeedCFill()** trap isn't fast and there can be a lot of points in a lasso sketch, so we added a shortcut to the algorithm to speed it up. Before retrieving a pixel's RGB value and testing it against the target RGB value, the algorithm checks to see whether or not the candidate point is in the remaining region, which is getting nibbled away by successive fills and subtractions. All of the original points are in the region, but picture what would happen if there were a point A ½ on the boundary in the white area between A and B. After the constriction is performed on point A and region 1 is removed from the whole, constricting at point A ½ would be redundant, so you do the relatively quick **PtInRgn()** test, skip it, and move on.

Showing the Region: Running Ants

Once the user has selected a region in the document window, it's important to show the region. After all, the user selected the region for a reason. He or she is going to use it as the object of some kind of action in the near future — it might be cut or copied, or it might be moved to a different part of the window. The accepted way to show the user the selected region is to frame it with a moving dotted outline, sometimes referred to as *running ants* because ... well, it just is, even though the effect doesn't look anything like running ants to us.

The basic idea in implementing running ants is that a line drawn with a nonsolid pattern in place comes out looking like a dotted line. And if subsequent lines drawn in the same place on the screen are drawn with two patterns that are similar but just lightly offset, the dots (or "ants") in the first line appear to have shifted to a new position in the second line. By repeatedly redrawing a line with a series of offset patterns, and if the last pattern in the series makes a smooth transition back to the first, the dotted line seems to run like the lights on a theater marquee.

A QuickDraw bit pattern is composed of 64 bits that are logically considered to form an 8-by-8-pixel design. If you want the ants to move by 1 bit for each shift (redraw), eight different patterns are required. The eight patterns are shown in Figure 6-11. The series of

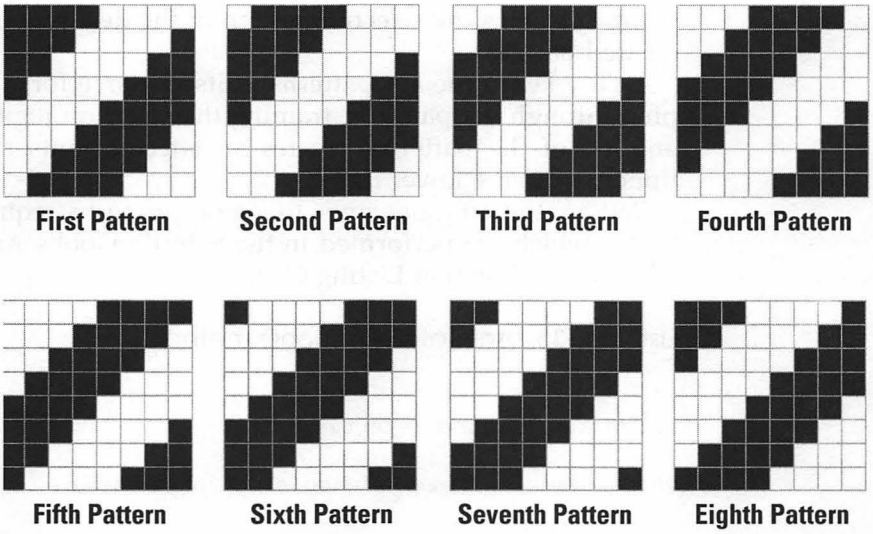


Figure 6-11. Ant patterns

patterns progresses as the labels under the patterns indicate, from first to eighth. The "line" on which the ants run is drawn in the current port by the selection object's FrameSelection() method, which is shown in Listing 6-32.

Listing 6-32. SelectionTool::FrameSelection() method

```
void SelectionTool::FrameSelection(
  Pattern  thePatt,
  short    copyMode )
{
PenState savedPen;

  if (this->SelectionEmpty())
    return;

  GetPenState( &savedPen );
  PenPat( thePatt );
  PenMode( copyMode );
  FrameRgn( this->theRgn );
  SetPenState( &savedPen );
}
```

Notice in Listing 6-32 that the FrameSelection() method accepts a pattern and a copy mode as parameters. Its job is pretty straightforward: It frames the selection region in the desired pen mode with the desired pattern.

ArtToy keeps the ant patterns in its resource fork. Repeatedly stepping through the patterns, framing the selection region with one after another of the patterns, creates an effect of ants running from the upper left to the lower right.

Getting the running ants to work properly requires a couple of tricks, which are performed in the SelectionTool's AnimateSelection() method, as shown in Listing 6-33.

Listing 6-33. AnimateSelection() method

```
/*
**Run ants around selection.
*/
void SelectionTool::AnimateSelection( void )
{
Handle   resHandle;
Pattern  curPatt, resultPatt;
short    pattResID, i;
```

```
  if (this->SelectionEmpty())
    return;

  if (this->currPattIndex < 1)
  {
    /* first time, use starting pattern */
    this->currPattIndex = 1;
    resHandle = (Handle)GetResource( 'PAT ', pattResID );
    if (resHandle != NULL)
    {
      this->pattCount = *(short*)(*resHandle);
      GetIndPattern( curPatt, pattResID, 1 );
      this->FrameSelection( curPatt, srcCopy );
    }
    return;
  }

  /* retrieve current patterns */
  GetIndPattern( curPatt, this->pattResID, this->currPattIndex );

  /* bump index and retrieve next pattern */
  this->currPattIndex++;
  if (this->currPattIndex >= this->pattCount)
    this->currPattIndex = 1;
  GetIndPattern( resultPatt, this->pattResID,
                 this->currPattIndex );

  /* combine current and next patterns: */
  /* byte by byte, create a difference pattern */
  for (i=0; i<sizeof(Pattern); i++)
    (((char*)resultPatt)[i]) =
      (((char*)curPatt))[i] ^ (((char*)resultPatt))[i];

  /* frame selection with the calculated difference pattern */
  this->FrameSelection( resultPatt, srcXor );
}
```

The first time we frame the selection we want to use the first of our patterns and write them in patXor mode. We chose to write the framing pixels in patXor mode for the sake of appearance and to make the programming easier. The easier programming comes when we want the ants to disappear.

As to appearance, if the ants are written in srcCopy mode, one pattern after another, the animation works, but foreground-colored ants run on a background-colored track. This is certainly all right in a

black-and-white world and it might be OK in colored pictures, but we think the effect is better if the user sees Xor-ed ants running over the picture's pixels.

Getting the ants to work in patXor mode might at first seem to be less desirable because, before you can write the next pattern to the screen, you have to remove the current pattern, which would cause undesirable flashing. The trick is to write only the first pattern and then, for subsequent patterns, to write a *difference* pattern. Figure 6-12 shows two equations for Boolean operations on bit patterns. The first one is for the creation of a difference pattern. This is done in memory every time the AnimateSelection() method is called subsequent to the first time.

The pattern labeled "Current pattern" represents what is on the screen when the AnimateSelection() method is called. "Next pattern" is what we want to replace it with. The method retrieves both patterns from the resource fork and creates a difference pattern as shown in the top equation of Figure 6-12. The next equation represents what happens when the difference pattern is sent to the FrameSelection() method. The result on the screen is the desired "Next pattern."

Choosing to write the ants in patXor mode can also make some programming easier. When the user clicks in a document window in which there is an animated selection, but clicks outside of the selection, the ants should disappear. If you had been drawing in patCopy

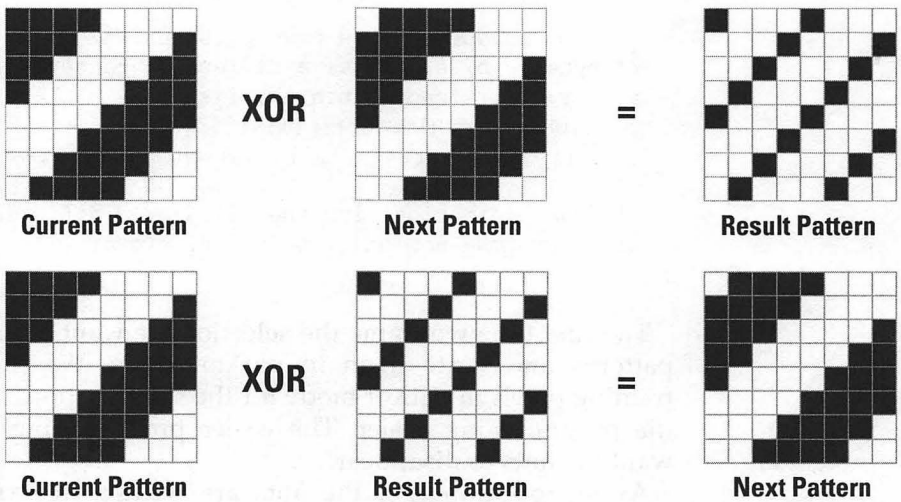


Figure 6-12. Ant pattern logic

mode, the screen contents would have been overwritten by the ants and their background. The only way to make them go away is to rewrite the screen in the ant track. Doing so means choosing between fancy region manipulation to isolate the ant track and the possibility of over-updating to rewrite the whole region. But because you write the ants in patXor, you can make them go away by writing them again in the current pattern. Listing 6-34 shows this being done in the **RemoveAnimation**() method.

Listing 6-34. SelectionTool::RemoveAnimation() method

```
/*
** Remove the ants.
*/
void SelectionTool::RemoveAnimation( void )
{
Pattern  curPatt;

  GetIndPattern( curPatt, this->pattResID, this->curPattIndex );
  this->FrameSelection( curPatt, srcXor );

  /* Have to start over */
  this->curPattIndex = (-1);
}
```

For most applications, removing the ants is a good idea when the window containing the selection goes behind another window; this could be a result of another of the program's windows being selected (deactivate event), another application taking over (suspend event), or a modal dialog coming up in front of the window. In all of these cases, if the ants were removed in response to the events we mentioned, resuming the animation would just be a matter of calling the **AnimateSelection**() method, which will behave as if it were animating the ants for the first time.

Implementing running ants provides a lot of room for alteration to suit your needs and your taste. One thing you might consider is a speed governor. The speed at which QuickDraw can frame a region varies depending on the complexity of the region. The **FrameRgn**() trap is optimized to recognize some special case regions; as a result, framing a rectangular region happens very quickly compared to other shapes, maybe too fast for good animation effects.

Dragging a Selected Region

When the user points into a selected region in the document window and clicks the mouse, ArtToy, like most picture-oriented applications, lets the user drag the selected region around the window. Two kinds of dragging are available: dragging outlines and dragging PixMaps. We discuss them both in the next two sections. ArtToy's SelectionTool object has only one method for dragging the selection, but we'll present two different versions of it to demonstrate the two different approaches to dragging regions.

Dragging an Outline. The simplest dragging to implement is to show the user the outline of the selection while it's in motion (that is, while the mouse button is held down). The SelectionTool's Drag() method uses the **DragGrayRgn**() trap, that does just what its name implies.

```
long DragGrayRgn(
  RgnHandle   rgn,
  Point       pt,
  Rect        *bounds,
  Rect        *slop,
  short       axis,
  ProcPtr     action );
```

The first two arguments are the region that is to be dragged and a point inside the region. The point is typically composed of the mouse coordinates where the user clicked inside the region.

The "bounds" and "slop" arguments are used to restrict the dragging to certain areas of the screen. (Remember: The mouse won't leave the visible area of the desktop so you don't have to worry about that.) The "bounds" argument must lie entirely within the slop rectangle (they may be the same rectangle). As the user drags the region on the screen, the point sent in is "pinned" inside the "bounds" rectangle — not the outline region, just the point. Figure 6-13 shows a region with its bounding rectangle (rgnBBox), a point inside the region, and a rectangle (dragRect) that will tell **DragGrayRgn**() to restrict dragging to keep the entire region in the window.

The "axis" argument can have one of three predefined values: *noConstraint*, which means that the region is free to move both horizontally and vertically; *hAxisOnly*, which means that the region can only move horizontally; and *vAxisOnly*, which means vertical movement only.

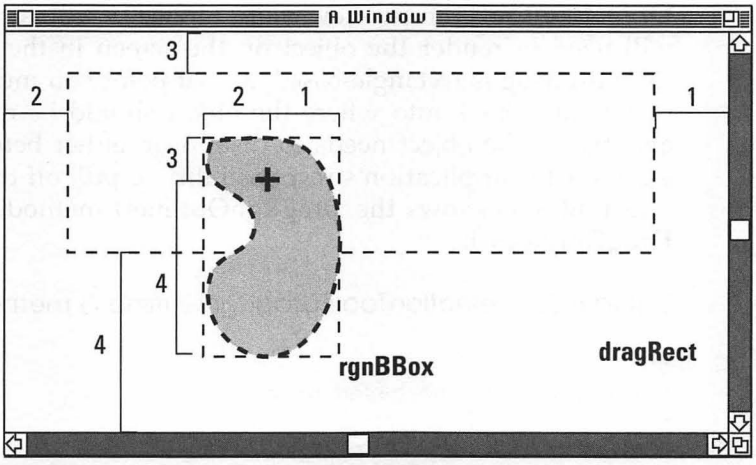


Figure 6-13. Dragging a region

The "actionProc" argument contains a pointer to a routine to be called repeatedly while the mouse button is held down.

The **DragGrayRgn**() trap returns the new coordinates of a point where the user dropped the region by letting up on the mouse button. The coordinates are stored in the 32-bit return code with the high-order 16 bits holding the vertical value and the low-order 16 bits holding the horizontal value. If the mouse was let up outside the "slop" rectangle, the return value is $80008000 hexadecimal.

Dragging the thumb in a vertical scroll bar is controlled by a call to **DragGrayRgn**() and is a perfect illustration of how this trap works. When you click in the thumb box, its outline becomes a gray region. As the mouse is moved, the thumb's outline tracks up and down in the scrollbar; the axis argument value is *vAxisOnly*. The mouse doesn't have to be inside the thumb while dragging, but if it gets too far away the thumb is dropped and the outline region disappears. This is the function of the "slop" argument: It provides a bit of tolerance so that the mouse location merely has to be close to the screen object, and not completely inside. When the mouse leaves the "slop" area, then the dragging is suspended until the mouse re-enters the "slop" area or the button is released.

One important thing about using **DragGrayRgn**() is that, before making the call, you should calculate and remember your point's offset from the region's bounding rectangle. You can think of the offset as local coordinates within the region's bounding rectangle. When

DragGrayRgn() returns new, valid mouse coordinates, the application will need to render the object on the screen in the new location. As most drawing is rectangle-based, at that point you must calculate a new offset value back into where the object should be drawn. Remember also that if the object needs to disappear, either before or after dragging, it's the application's responsibility to pull off the vanishing act.

Listing 6-35 shows the DragRgnOutline() method, and how it uses **DragGrayRgn**().

Listing 6-35. SelectionTool::DragRgnOutline () method (gray outline)

```
/*
**Facilitate mouse-based dragging
*/
Boolean SelectionTool::DragRgnOutline(
  Point*      startPt,
  GWorldPtr   windowGW )
{
Rect          rgnBoundsR, offBoundsR, dragLimitR, slopLimitR;
Point         draggedToPt;
PixMapHandle  windowCopyPM, offscreenPM, windowPM;
CGrafPtr      savedPort;
GWorldPtr     windowCopyGW;
GDHandle      savedGDev;
OSErr         errStat;
long          newDragValue;
short         curPixDepth;
Boolean       didDrag;

  /* better safe than real sorry! */
  if (this->theRgn == NULL)
    return FALSE;

  /* determine current graphics environment */
  GetGWorld( &savedPort, &savedGDev );

  /* determine a dragging limit that keeps the region */
  /* entirely within the window during dragging */
  rgnBoundsR = (**(this->theRgn)).rgnBBox;
  SetRect(
    &dragLimitR,
    startPt->h - rgnBoundsR.left,
    startPt->v - rgnBoundsR.top,
    savedPort->portRect.right - (rgnBoundsR.right - startPt->h),
    savedPort->portRect.bottom - (rgnBoundsR.bottom - startPt->v));
```

```
slopLimitR = dragLimitR;
InsetRect( &slopLimitR, -10, -10 );

/* let the user drag the region around */
newDragValue =
  DragGrayRgn(
    this->theRgn, *startPt, &dragLimitR, &slopLimitR,
    noConstraint, NULL );

/* determine if the user performed a valid drag operation */
didDrag = (newDragValue != 0x80008000);

/* perform the move */
if (didDrag)
{
  windowPM = GetGWorldPixMap( windowGW );
  if (!LockPixels( windowPM ))
    return FALSE;

  /* copy the selected pixel values */
  curPixDepth = (**windowPM).pixelDepth;
  offBoundsR = rgnBoundsR;
  OffsetRect( &offBoundsR, -offBoundsR.left, -offBoundsR.top );
  errStat =
    NewGWorld(
      &windowCopyGW, curPixDepth, &offBoundsR,
      NULL, NULL, (GWorldFlags)0 );
  if (errStat != noErr)
  {
    DisposeGWorld( windowCopyGW );
    UnlockPixels( windowPM );
    return FALSE;
  }

  windowCopyPM = GetGWorldPixMap( windowCopyGW );
  if (!LockPixels( windowCopyPM ))
  {
    DisposeGWorld( windowCopyGW );
    UnlockPixels( windowPM );
    return FALSE;
  }
  CopyBits(
    (BitMap*)*windowPM, (BitMap*)*windowCopyPM,
    &rgnBoundsR, &offBoundsR, srcCopy, this->theRgn );
```

```
        /* erase pixel values at original location */
        EraseRgn( this->theRgn );

        /* play a game to type-coerce the drag value into a Point */
        /* and move the region to the new dragged location */
        draggedToPt = *(Point*)&newDragValue;
        OffsetRgn( this->theRgn, draggedToPt.h, draggedToPt.v );
        rgnBoundsR = (**(this->theRgn)).rgnBBox;

        /* copy original pixel values into new location */
        CopyBits(
          (BitMap*)*windowCopyPM, (BitMap*)*windowPM,
          &offBoundsR, &rgnBoundsR, srcCopy, this->theRgn );

        /* clean up */
        UnlockPixels( windowPM );
        DisposeGWorld( windowCopyGW );
    }

    SetGWorld( savedPort, savedGDev );
    return didDrag;
}
```

The DragRgn() method, as shown in Listing 6-35, makes a copy of the pixels under the selection in the window before letting the user drag the gray outline. After the user has completed dragging, DragRgn() leaves a hole where the selection originally was and copies the selected pixels to their new location in the window.

Like the Sketch() methods, the DragRgn() method is part of the general "point, click, drag, and release" processing common to all mouse-based tools. As such, it employs a **WaitMouseUp**() loop, but you can't see it in the code because it's in the call to **DragGrayRgn**(). The next section describes how ArtToy works if its DragRgn() method actually drags the pixels instead of using **DragGrayRgn**() to drag an outline. You'll also see the **WaitMouseUp**() loop at work.

Dragging the Selected Pixels. Let's change the visual effect of the DragRgn() method. Instead of the user dragging a gray outline of the region around the window, you can create the effect of actually dragging the pixels. This change requires minor changes to the way the **DragRgn**() method was coded in the preceding section. Specifically, you need to replace the call to **DragGrayRgn**() and take care of the **WaitMouseUp**() processing in the method.

The essence of the change is that instead of offsetting and framing the region as **DragGrayRgn**() does in its **WaitMouseUp**() loop, you want to call **CopyBits**() to paint the screen from the temporary GWorld in which you saved the selected pixels. Listing 6-36 shows the new version of the DragRgn() method.

Listing 6-36. SelectionTool::DragRgnPixels() method

```
/*
** Facilitate mouse-based dragging.
*/
Boolean SelectionTool::DragRgnPixels(
  Point*      startPt,
  GWorldPtr   windowGW )
{
PixMapHandle  selCopyPM, windowPM;
GWorldPtr     selCopyGW, savedPort;
GDHandle      savedGDev;
RgnHandle     curRgn, newRgn, updateRgn;
Rect          sourceR, destR, offWorldR, rgnBoundsR, dragLimitR;
Point         curPt, lastPt;
short         curPixDepth;
OSErr         errStat;

  /* better safe than real sorry! */
  if (this->theRgn == NULL)
    return FALSE;

  /* determine current graphics environment */
  GetGWorld( &savedPort, &savedGDev );

  /* allocate an image buffer to hold a copy of the selected pixels */
  curPixDepth = (**(savedPort->portPixMap)).pixelDepth;
  offWorldR = (**(this->theRgn)).rgnBBox;
  OffsetRect( &offWorldR, -offWorldR.left, -offWorldR.top );
  errStat =
    NewGWorld(
      &selCopyGW, curPixDepth, &offWorldR,
      NULL, NULL, (GWorldFlags)0 );
  if (errStat != noErr)
    return FALSE;

  selCopyPM = GetGWorldPixMap( selCopyGW );
  if (!LockPixels( selCopyPM ))
  {
```

```
    DisposeGWorld( selCopyGW );
    return FALSE;
  }
  windowPM = GetGWorldPixMap( windowGW );
  if (!LockPixels( windowPM ))
  {
    DisposeGWorld( selCopyGW );
    return FALSE;
  }

  /* copy the selected pixels into the new GWorld */
  destR = (**(this->theRgn)).rgnBBox;
  SetGWorld( selCopyGW, NULL );
  CopyBits(
    (BitMap*)*windowPM, (BitMap*)*selCopyPM,
    &destR, &offWorldR, srcCopy, this->theRgn );

  /* determine a dragging limit that keeps the region */
  /* entirely within the window during dragging */
  rgnBoundsR = (**(this->theRgn)).rgnBBox;
  SetRect(
    &dragLimitR,
    startPt->h - rgnBoundsR.left,
    startPt->v - rgnBoundsR.top,
    savedPort->portRect.right - (rgnBoundsR.right - startPt->h),
    savedPort->portRect.bottom - (rgnBoundsR.bottom - startPt->v) );

  curRgn = NewRgn();
  newRgn = NewRgn();
  updateRgn = NewRgn();
  CopyRgn( this->theRgn, curRgn );

  /* restore port, hopefully to the target window */
  SetGWorld( savedPort, savedGDev );

  curPt = *startPt;
  while (WaitMouseUp())
  {
    /* get current mouse location */
    GetMouse( &curPt );

    if (!EqualPt( curPt, lastPt ) && PtInRect( curPt, &dragLimitR ))
    {
      /* copy and relocate the region to new mouse location */
      CopyRgn( curRgn, newRgn );
      OffsetRgn( newRgn, curPt.h - lastPt.h, curPt.v - lastPt.v );
```

```
            /* determine area to be redrawn */
            SectRgn( newRgn, curRgn, updateRgn );

            /* update the invalid portion of the window */
            /* just uncovered by the move */
            sourceR = (**windowPM).bounds;
            destR = savedPort->portRect;
            CopyBits(
               (BitMap*)*windowPM, (BitMap*)*(savedPort->portPixMap),
               &sourceR, &destR, srcCopy, updateRgn );

            /* draw the region in the new location within the window */
            sourceR = savedPort->portRect;
            destR = (**newRgn).rgnBBox;
            CopyBits(
               (BitMap*)*selCopyPM, (BitMap*)*(savedPort->portPixMap),
               &sourceR, &destR, srcCopy, newRgn );

            lastPt = curPt;
         }
      }

      /* update the window pixel buffer */
      SetGWorld( windowGW, NULL );

      /* leave a hole and copy selection to new location */
      sourceR = (**selCopyPM).bounds;
      destR = (**newRgn).rgnBBox;
      EraseRgn( this->theRgn );
      CopyBits(
         (BitMap*)*selCopyPM, (BitMap*)*windowPM,
         &sourceR, &destR, srcCopy, newRgn );

      /* move the region */
      OffsetRgn( this->theRgn, curPt.h - startPt->h, curPt.v -
                 startPt->v );

      /* clean up */
      UnlockPixels( windowPM );
      DisposeRgn( curRgn );
      DisposeRgn( newRgn );
      DisposeRgn( updateRgn );
      DisposeGWorld( selCopyGW );
      SetGWorld( savedPort, savedGDev );

      return TRUE;
   }
```

Using the Selection Tools

Listing 6-37, which is similar to Listing 6-9, shows how ArtToy uses a selection tool during mouseDown event processing. Unlike Listing 6-9, this time the current mouse tool is the marquee selection tool and the user is about to define a rectangular area in the document window.

Listing 6-37. MouseDown processing for marquee selection

```
SelectionTool *HandleMouseDown(
  Point  globalMouse )
{
MarqueeTool    *theDrawTool;
WindowPtr      theWindow;
short          windowPart;
Point          localMouse;

  windowPart = FindWindow( globalMouse, &theWindow );
  if (theWindow != FrontWindow())
    return NULL;

  switch (windowPart)
  {
    case inContent:
      switch (currentMouseTool)
      {
        case MarqueeToolID:
          SetGWorld( (CGrafPtr)theWindow, GetGDevice() );

          /* Create and initialize selection object */
          theDrawTool = new( MarqueeSelection );
          theDrawTool->Init();
          theDrawTool->SetLimitRect(
            &(((GrafPtr)theWindow)->portRect) );

          /* Let user go nuts */
          localMouse = globalMouse;
          GlobalToLocal( &localMouse );
          theDrawTool->Sketch( localMouse );

          if (!theDrawTool->SelectionEmpty())
            /* return selection object */
            return (SelectionTool*)theDrawTool;
          else
          {
            /* Object be gone! */
            theDrawTool->ObjKill();
            delete( theDrawTool );
```

```
            }
            break;

          default:
            break;
        }
    }

    return NULL;
}
```

Listing 6-37 looks like Listing 6-39 with two important differences in what happens when the sketching is finished. If the user defined a nonempty region, then the selection tool is saved. Whereas the drawing tools were disposed of as soon as they were drawn into the window's offscreen buffer, the selection tool gets saved in a global variable that can be used to animate the selection and provide access to the selected region for other routines to process further.

One way that the selection gets accessed is when the user drags it by pointing into the selected regions and pressing the mouse button. Listing 6-38 shows how mouseDown event processing in ArtToy goes when the user clicks inside the selection.

Listing 6-38. Handling a mouseDown event in a selection

```
void HandleAMouseDown(
  Selection   *aSelection )
{
WindowPtr   theWindow;
short       windowPart;

  windowPart = FindWindow( globalMouse, &theWindow );
  if (theWindow != FrontWindow())
    RETURN;

  switch (windowPart)
  {
    case inContent:
      if ((globalSelection != NULL) &&
          globalSelection->PtInSelection( localMouse ))
      {
        globalSelection->DragRgn();
      }
      break;
  }
}
```

▶ Summary

This chapter discussed mouse-based drawing. We looked at the way a paint application, ArtToy, uses the two basic kinds of mouse-based tools: drawing tools and selecting tools.

Both drawing and selecting tools require a sketching method that focuses on the mouse while the user is defining the desired shape. Inasmuch as the graphical user interface involves pointing, clicking, dragging, and releasing, the sketch method takes control after the click and does its thing until the mouse button is released.

Selection objects, which are based on the drawing objects, let the user define an area on the screen that can be turned into a QuickDraw region and manipulated. Defining an area with a lasso selection tool requires a constricting method based on the **SeedCFill**() trap.

Once defined, a selected area is highlighted by running ants. When the user clicks inside the selected area, the selected pixels are dragged around by the mouse. We also presented two methods of implementing dragging: dragging an outline with **DragGrayRgn**() and dragging pixels.

In Chapter 7 we discuss advanced **CopyBits**() usage and image processing.

7 ▶ Image Processing with QuickDraw

Photographers have a number of means at their disposal for producing custom images. They can place a special lens on the camera to warp or distort an image, or they can use special filters during the printing process to control the color of a print. They can even splice negatives or film together using special backgrounds to create effects, such as a boy flying through the air on a bicycle in the movie *E.T.*

While such optical techniques provide great flexibility, speed, and a nearly endless variety of output, many methods are complicated and require expensive, highly specialized equipment. Once an image is entered into a computer, however, the programmer has complete control of how that image is processed. Rather than requiring specialized equipment to produce each effect, the programmer simply needs a routine that performs the desired transformation. And with the computer, anything imaginable is possible.

It's no wonder that many of the special effects produced in Hollywood are either assisted or completely generated by computer. The Genesis planet-transformation scenes in *Star Trek III: The Search for Spock*, Raziel's transformation sequence in *Willow*, the hoverboard sequences in *Back to the Future*, many of the spiders in *Arachnophobia*, various scenes in *Ghost*, and the water animation in *The Abyss* are only a few examples of movies employing computer image processing. While some sequences are entirely generated by computer, most are filmed and then processed. Processing such images usually requires a programmer who creates special tools and an artist who uses those tools to produce the final image.

While QuickDraw does not have the built-in ability to perform the highly specialized image processing necessary to produce the kinds of effects seen in these movies, producing such effects is not beyond the capability of the Macintosh. Generating such effects involves creating a GWorld and manipulating its contents.

As far as QuickDraw is concerned, an image is a PixMap or a BitMap. Aside from the simple maintenance functions performed by the GWorld traps, **CopyBits**() and **CopyDeepMask**() are the workhorses of QuickDraw's image processing. With these traps, you can perform several standard image-processing operations on a PixMap, such as resizing (by stretching, shrinking, or clipping the image), colorizing, or masking certain areas. The following sections discuss how to perform these operations.

This chapter introduces the pixel-image-processing capabilities of QuickDraw. The concepts related to aliasing, and how to avoid it, are discussed in detail. The chapter continues with an in-depth investigation into QuickDraw's important pixel-image-processing traps, describing and demonstrating how the arguments to the pixel-processing traps can be manipulated to produce a variety of useful, interesting, or bizarre effects. The chapter concludes with a discussion of depth and geometry conversions of pixel images.

▶ Pixel-Processing Traps

Most QuickDraw drawing traps take shape definitions (such as lines, circles, or rectangles) as input arguments. **CopyBits**(), on the other hand, takes a pointer to a PixMap or BitMap as two of its input arguments. Like other QuickDraw traps, **CopyBits**() renders the result into a pixel image varible, either of type PixMap or BitMap. Thus, **CopyBits**() transfers pixel image data between two pixel image variables.

To fully understand the capabilities of **CopyBits**(), you need to have a solid conceptual model of all the variables that are used to transfer an image. There are explicit parameters as well as two global variables that affect the outcome of a call to **CopyBits**(). The roles that the global variables play are discussed in the following section, preceded by a description of the arguments explicitly passed to each trap.

The first of the three traps is **CopyBits**(), which has been around as long as the Macintosh. The second trap, **CopyMask**(), was introduced with the Macintosh Plus and System 3.2. The newest member of the family, **CopyDeepMask**(), was introduced with System 7.0 and is available only on Color QuickDraw machines. It encompasses and expands on the functionality of the first two Pixel processing traps.

```
void CopyBits(
   BitMap      *srcMap,
   BitMap      *destMap,
   Rect        *srcRect,
   Rect        *destRect,
   short       transferMode;
   RgnHandle   maskRgn );

void CopyMask(
   BitMap      *srcMap,
   BitMap      *maskMap,
   BitMap      *destMap,
   Rect        *srcRect,
   Rect        *maskRect,
   Rect        *destRect );

void CopyDeepMask(
   PixMap      *srcMap,
   PixMap      *maskMap,
   PixMap      *destMap,
   Rect        *srcRect,
   Rect        *maskRect,
   Rect        *destRect,
   short       transferMode,
   RgnHandle   maskRgn);
```

These three traps share the same underlying code; the difference is in the explicit argument passed to them, and whether the results of the traps can be saved into pictures or printed. Of the three Stretch traps, only **CopyBits**() uses the **StdBits**() bottleneck and is therefore the only one whose image results can be *directly* printed or saved in a picture. This limitation can be overcome by creating the desired image using **CopyMask**() or **CopyDeepMask**() and then copying the destination pixel image argument's contents using **CopyBits**().

Note ▶

Since these traps are similar, it's useful to refer to them generically. At Apple, it's customary to refer to them collectively as the *Stretch* traps, which happens to be the name of the source file that implements them. This convention is followed here.

▶ Basic CopyDeepMask() and CopyMask() Functionality

Combining two or more images to produce a third image is called called *image compositing* and can be observed in television commercials, introductions to sports broadcasts, and other contemporary, video-based media. Color QuickDraw contains two traps that provide substantial image compositing capabilities: **CopyDeepMask**() and **CopyMask**(). The functionality of the **CopyDeepMask**() trap is a perfect superset of those found in **CopyMask**().

Color Plates CP-2 and CP-3 show images produced with the System 7.0 version of **CopyMask**(). CP-2 simply uses a grayscale mask and blends between the source and the destination. CP-3 uses a solid red mask and demonstrates that the red color component is completely masked.

Even the complicated results produced in these colored figures only scratch the surface of the image compositing power of **CopyDeepMask**(). In all of the colored figures, both a srcCopy transfer mode and a NULL "maskRgn" argument were used.

Additionally, all colored figures were generated with either a relatively simple mask or simple source pixel image. Imagine the possibilities when two complex images are combined. The addition of an arithmetic transfer mode and/or a complex masking region provides even greater possibilities.

Note ▶

Many developers have been asking Apple for an alpha transfer mode — a way to write to the unused high bit of 16-bit pixels or to the high byte of 32-bit pixels. In most cases, the goal is to use this value for blending or merging images. Currently, Apple evangelizes the use of **CopyDeepMask**() for performing merging/blending operations.

▶ Explicit Arguments

Image Buffers: srcMap, maskMap, dstMap

The versions of QuickDraw contained in all versions of System 6 and the first public release of System 7.0 allow applications to pass a pointer to a CGrafPort instead of a pointer to a PixMap, even though it's not strictly correct. This may not hold true in future versions of QuickDraw.

Important ▶

While the source and destination PixMaps can arbitrarily overlap screen devices, the mask image cannot belong to a public graphics device; in other words, the mask image must be offscreen.

As with "srcMap" and "dstMap," "maskMap" can be either a BitMap or a PixMap pointer. For **CopyMask**(), prior to System 7.0 "maskBits" is restricted to being a BitMap or 1-bit-per-pixel PixMap. In this case, "maskMap" values are used in the same way as the "maskRgn" argument: They determine whether the resulting pixel is affected or not. If the pixel value in "maskMap" is black, the source value is used; if it's white, the destination is left unchanged.

With System 7.0 the "maskMap" argument for CopyMask and the new call **CopyDeepMask**() can be any valid pixel depth, including 16 and 32 bits per pixel. In this case, the index values in "maskMap" are used as a weighting factor between the corresponding source and destination pixel values, according to the same rules as the arithmetic weighting color described in Chapter 3. Simply put, low RGB component values in "maskMap" select more of the source pixel value in the result; greater "maskMap" values favor the destination pixel value.

Rectangles: srcRect, maskRect, destRect

These rectangles specify the active portion of the PixMap. A current restriction is that the source and mask rectangles must be the same size. The PixMap's coordinates are established by the bounds rectangle, and the "srcRect," "maskRect," and "destRect" arguments specify a subset of this space.

Transfer Mode

The "transferMode" argument allows you to explicitly specify a transfer mode for **CopyBits**(). Most other QuickDraw traps use the current port's pen mode as the transfer mode. Chapter 3 discusses the various transfer modes.

There are only a few things to add about transfer modes. First, search procedures are not called and colorizing does not occur for arithmetic transfer modes. Second, the transfer mode operation occurs after the source is colorized and the search procedure has been called.

Finally, the standard transfer modes operate in index space, not color space. Thus, or-ing a multi-color image on an indexed and a direct GDevice can produce different results, depending on the

images involved. This is most obvious when xor-ing a black image on a 50 percent gray background. On a direct device, the operation is not visible since colors in the result will all be $7FFF or $8000 hexadecimal: virtually indistinguishable. On an indexed device, the result depends on the organization of the destination color table. For a standard color table, yellow appears in the result image, a color unrelated to the operation. Yellow appears only because of its location in the color table. Figure 7-1 shows when the various operations occur during a **CopyDeepMask**() call.

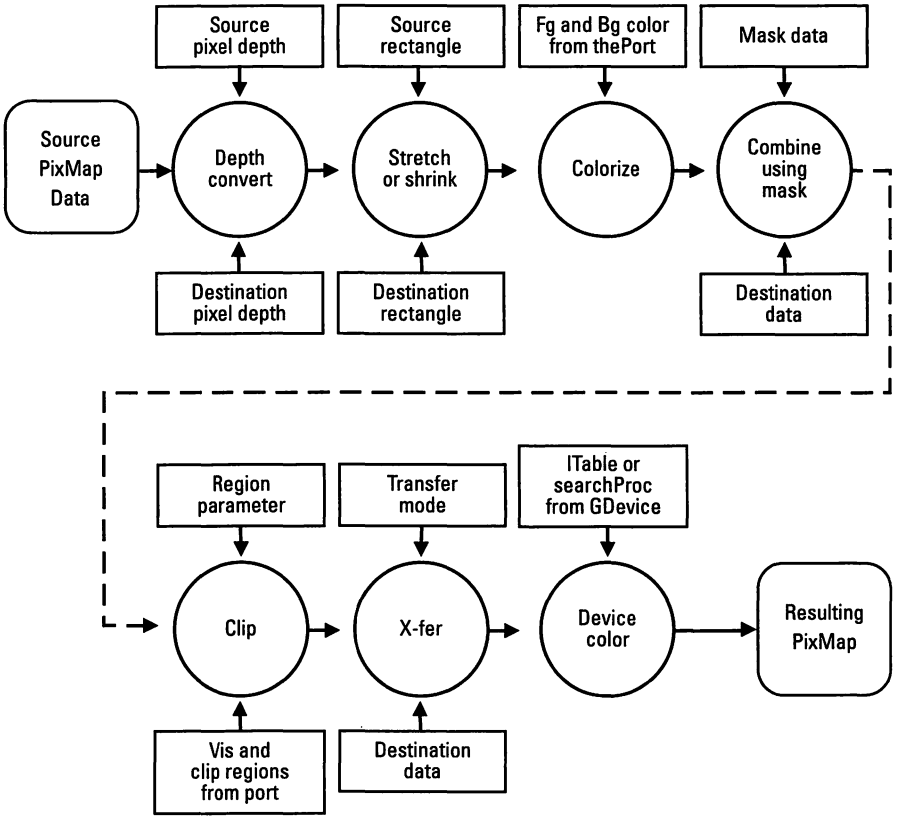


Figure 7-1. CopyDeepMask() functional diagram

Mask Region

This parameter is simply another region that further restricts the affected area. Like with all QuickDraw calls, drawing is automatically clipped to the intersection of the port's *visRgn* and *clipRgn*. When a non-NULL mask region is specified, drawing is clipped to the intersection of the mask region and the port's regions.

▶ Global Arguments

A cause of great confusion, and a number of bugs, stems from the way Color QuickDraw uses global variables when transferring pixel images. There are two global variables that are used: The current GrafPort or CGrafPort (typically stored in the application global variable "thePort"), and the current GDevice (pointed to by the low memory global "theGDevice"). Clipping, colorizing information, and a number of other variables are taken from the current port, while the GDevice is used to determine how colors will be represented in the destination pixel image, according to the rules of color mapping.

"thePort"

All QuickDraw traps use the current port for the destination of the drawing. Because of the calling interface it may appear that the three Stretch traps are an exception to this rule. As this section shows, these traps use the current port for clipping and color information, just like other QuickDraw calls.

CopyBits() provides slightly more flexibility than the standard QuickDraw traps by allowing applications to explicitly specify the destination PixMap. This can be confusing since it departs from the standard QuickDraw model in which the destination is always the current port's PixMap.Calls to **CopyBits**() are usually simpler than the trap interface would indicate. To illustrate this point, let's consider a simpler **CopyBits**().

```
void CopyBitsToPort(
  BitMap *srcMap,
  Rect   *srcRect,
  Rect   *destRect );
```

Since **CopyBits**() is primarily used to update the current port, the "dstBits" argument is simply a pointer to the *portBits* or *portPixMap* field of "thePort." The transfer mode is almost always srcCopy, and

no additional clipping is needed, other than the current port's regions. So the need to declare and assign variables for the arguments to **CopyBits()** is often nothing more than excess baggage. Listing 7-1 shows how the interface to **CopyBits()** can be simplified in this manner.

Listing 7-1. CopyBits() simplified

```
void CopyBitsToPort(
  BitMap *srcMap,
  Rect   *srcRect,
  Rect   *destRect )
{
Ptr      destMapPtr;
Boolean  curPortIsColor;

  /* check if the current port is color */
  curPortIsColor = ((thePort->portVersion) & 0xC000) != 0);

  /* get reference to the port's BitMap/PixMap */
  if (curPortIsColor)
    destMapPtr = *(((CGrafPtr)thePort)->portPixMap);
  else
    destMapPtr = &(thePort->portBits);

  CopyBits(
    srcMap, destMapPtr, srcRect, destRect,
    srcCopy, (RgnHandle)NULL );
}
```

This simplified version calls the "real" **CopyBits()** using the PixMap from the current port as the "dstBits" argument and "srcCopy" as the transfer mode argument. In this case no additional clip region is specified, and the operation will affect only those pixels within the intersection of the destination rectangle argument and the current port's clipRgn and visRgn.

Even though **CopyBits()** can be passed any BitMap or PixMap pointer to use as the destination, it's important to realize that **CopyBits()** still uses a number of variables from the current port to perform the operation. Specifically, the *fgColor, bkColor* (or *rgbFgColor* and *rgbBkColor* for color ports), *visRgn, clipRgn, colrBit,* and *picSave* fields are used. The *colrBit* field is used for color separation when printing classic colors as described in *Inside Macintosh,* Volume I. The *picSave* field is used to determine whether the call should be recorded in a picture.

Clipping and Visible Regions. As described in Chapter 3, all QuickDraw drawing operations are limited by the current port's visRgn and clipRgn. During a call to **CopyBits**() that passes the current port's pixel image as the destination, these regions further limit drawing by intersection with the mask region argument. If the destination pixel image is not that of the current port's, then only the "maskRgn" argument is used. Careless specification of destination rectangle and mask region arguments can cause **CopyBits**() to write to memory outside of the pixel image variable's baseAddr memory, usually causing application failure.

"theGDevice"

Like all other QuickDraw calls, the current GDevice determines how pixels transferred by **CopyBits**() are represented in the destination pixel image.

As discussed in Chapter 3, the default mapping of colors to a GDevice is facilitated with inverse table lookups for indexed devices, and by truncating RGB component values for direct devices. An application can augment, or completely replace, this default color mapping by attaching a search procedure to the GDevice. QuickDraw will then call the search procedure each time a color needs to be mapped to that device. Search procedures are described in Chapter 3.

Destination Color Table. The color information for the destination PixMap is taken from the current GDevice, *not* from the destination PixMap's ColorTable. Think about that for a moment: The destination PixMap's ColorTable has nothing to do with how colors are drawn to that PixMap. The ColorTable is used to describe what the pixel values mean once they are in the PixMap. QuickDraw handles this for you when drawing to the screen. When drawing offscreen an application must ensure that the GDevice's color table is synchronized with the destination PixMap's color table.

This is the same way the GDevice is used other places in QuickDraw. In most cases, when drawing to a port on the screen or into an offscreen GWorld, the current GDevice's PixMap and the destination PixMap are the same. Problems occur more often with **CopyBits**(), since it's easier to specify any PixMap as the destination.

Important ▶

The color information for the destination PixMap is taken from the current GDevice, not from the destination PixMap's color table; usually they are one and the same.

▶ Altering Images with CopyBits()

What Is Color Separation?

Color separation is a process that distills an image into a set of colors. The most common use of color separation is in printing, where the image is often printed in separate passes of cyan, magenta, yellow, and black.

What Is Colorizing?

Colorizing is the result of applying a "nonblack" foreground and/or a "nonwhite" background color to a QuickDraw drawing operation. For shape-drawing operations such as **FrameRect()**, the foreground and background colors in the port determine the color of the shape for all unspecified values in the pen pattern (see the discussion under "Color Patterns" in Chapter 3). This effect is quite predictable and usually desirable. It also gives applications a simple way to draw in color without having to explicitly use color patterns.

For pixel transfers, the effect of colorizing is that high-pixel RGB values in the source PixMap go to the foreground color and low-pixel RGB values go to the background color.

CP-4D shows an example of the QuickDraw process called *colorizing*: Once the destination pixel color is determined, all bits that are off are given the value of the corresponding bit in the foreground color, and all bits that are on are given the value of the corresponding bit in the background color. In most situations, colorizing is undesirable and can be suppressed by using a foreground color of black and a background color of white.

Colorizing a pixel is via the following formula:

```
Result = (bgColor AND src) OR (fgColor AND ~src)
```

From the formula it is easy to see that if the bgColor is white (all components $FFFF) and the fgColor is black (all components $0000) the operation leaves the source value unchanged. If the fgColor is white and the bgColor is black, the colorizing operation inverts the source.

Before System 7.0, the problem was that the index, rather than the color values, were being colorized. Thus the result depended on the organization of the destination GDevice's ColorTable. Colorizing in index space is defined only for source values that are zero or all ones

that go to the current port's "fgColor" and "bgColor" values respectively. In System 6, colorizing is useful primarily for transferring 1-bit source images. In System 7, colorizing takes place in color space. This form of colorizing is useful at all pixel depths. Figure 7-2 shows how colorizing operations are performed.

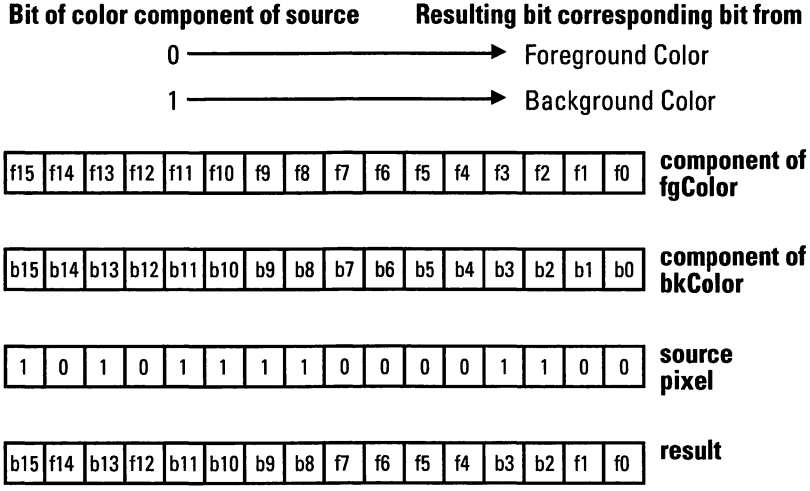


Figure 7-2. Colorizing operations

This operation probably seems a trifle convoluted at first, but it turns out to be quite useful. It provides an easy way to modify pixel color values, and a variety of interesting results can be produced. For example, by changing the foreground and background colors you can invert, posterize, or even perform color separation. Color plates CP-4A through CP-4F show a picture drawn several ways by changing only the foreground and background colors. These colorizing techniques are discussed in the following sections. Color plate CP-4D contains primarily red with some yellow and magenta and was generated by setting the foreground color to red ($FFFF, $0, $0) and the background color to white ($FFFF, $FFFF, $FFFF). Therefore, the red component is $FFFF hexadecimal for every pixel in the result. As CP-4D shows, the gray areas of the image transform to shades of red, the green areas go to yellow (red + green), and the blue areas to magenta (red + blue).

What Is Posterizing?

Posterizing is an effect produced by quantizing colors. For example, rather than using a complete range of reds, greens, and blues to represent an image, a posterizing effect can be achieved by using only the top bit of each RGB color component. To quantize or reduce the colors to eight different levels, 1-bit each of red, green, and blue, the foreground color should be set to black ($0, $0, $0) and the background color to gray ($8000, $8000, $8000). This strips the low bits of each color, thus yielding only eight colors in the result. You could just as easily maintain four levels (2-bits) of each color component using a foreground color of black ($0, $0, $0) and a background color of very light gray ($C000, $C000, $C000). This would yield

= 2 (2 bits × 3 components)
= 2^6
= 64 colors

The image in CP-4C, was generated using the image in CP-4A. CP-4C has only solid colors of red, green, blue, cyan, magenta, yellow, and gray, and was generated by using a background color of white ($FFFF, $FFFF, $FFFF) and a foreground color of gray ($8000, $8000, $8000).

Colorizing Techniques

Inverting. Inverting an image in color space simply means changing the black pixels to white and the white pixels to black. This basically produces a negative of the source image. This is useful for highlighting user interface objects such as buttons, as shown in Figure 7-3.

This effect can be achieved by using the notSrcCopy transfer mode, or by setting the foreground color to white and the background color to black. When the foreground color is white ($FFFF, $FFFF, $FFFF), every bit that is off will be turned on. When the background color is black ($0, $0, $0), every bit that is on will be turned off. The result is a color space inversion of the source image.

You can also invert just one color component. For example, to invert the red component you would set the foreground color to red ($FFFF, $0, $0) and the background color to cyan ($0, $FFFF, $FFFF).

Three of the images (CP-4B, CP-4E, and CP-4F) use inversion. CP-4B was generated using a foreground color of white ($FFFF, $FFFF, $FFFF) and a background color of black ($0, $0, $0).

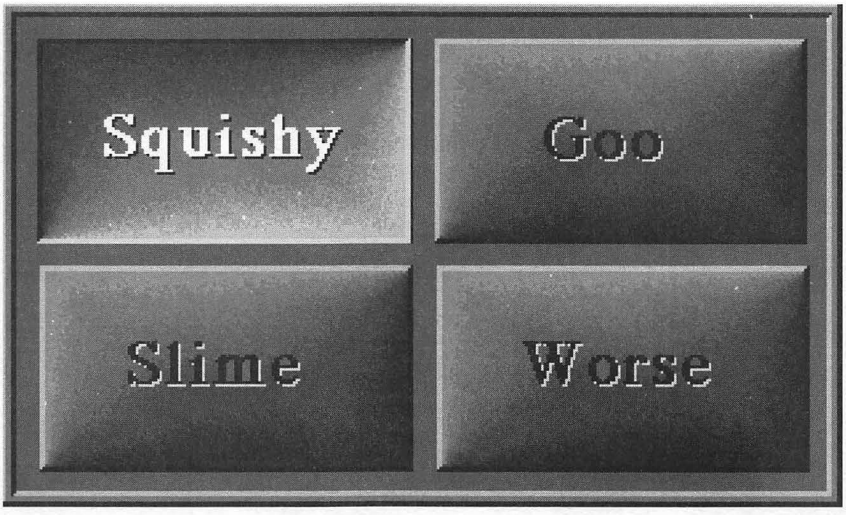


Figure 7-3. Buttons highlighted by color inversion

CP-4E was generated using a foreground color of white ($FFFF, $FFFF, $FFFF) and a background color of gray ($8000, $8000, $8000). This has the effect of inverting all the bits and forcing the high bit of each color component to 1. This image, therefore, contains no black.

CP-4F was generated using a foreground color of gray ($8000, $8000, $8000) and a background color of gray ($7FFF, $7FFF, $7FFF). This inverts only the high bit of each color component. This is an alternative way to highlight. The problem with inverting all the bits is that gray ($8000, $8000, $8000) goes to gray ($7FFF, $7FFF, $7FFF). These two grays are indistinguishable since they are immediately adjacent in color space. By inverting only the high bit, you guarantee that all colors will shift by the same amount in RGB color space.

Color Separation. With foreground and background colorizing, it's possible to separate an image into its red, green, and blue components. The technique is simple: Set the foreground color to black and set the background color to the component you want. To get the blue component, for example, set the background color to pure blue ($0, $0, $FFFF).

An application can also perform CMY color separation by setting the foreground and background color and calling **CopyBits**(). The routine in Listing 7-2 shows an example of this. In this example, a picture with resource ID 1000 is separated into its cyan, magenta, and yellow components.

Listing 7-2. Color separation example

```
#define  PICTResID  1000

void CMYColorSeparation( void )
{
Rect          boundsR, destR;
long          *bitsPtr;
RGBColor      myrgb, savergb;
GDHandle      savedGD;
CGrafPtr      savedPort;
GWorldPtr     myOffGWorld;
PixMapHandle  myPixMapHandle;
PicHandle     myPicHandle;
OSErr         errStat;

  myPicHandle = GetPicture( PICTResID );
  if (myPicHandle == NULL)
    return;          /* failed -> exit */

  boundsR = (*myPicHandle)->picFrame;
  OffsetRect( &boundsR, -boundsR.left, -boundsR.top );
  errStat = NewGWorld( &myOffGWorld, 32, &boundsR, NULL,
    NULL, 0L );
  if (errStat != noErr)
  {
    GetGWorld( &savedPort, &savedGD );
    GetForeColor( &savergb );
    SetGWorld( myOffGWorld, NULL );

    EraseRect( &boundsR );    /* clear the GWorld */

#ifdef _GET_GWPM_FIXED_
    myPixMapHandle = GetGWorldPixMap( myOffGWorld );
#else
    myPixMapHandle = myOffGWorld->portPixMap;
#endif _GET_GWPM_FIXED_

    (void)LockPixels( myPixMapHandle );
    DrawPicture( myPicHandle, &boundsR );

    /* copy the separations into window */
    SetGWorld( savedPort, savedGD );
```

```
        /* separate out the yellow component */
        myrgb.red = 0xFFFF;
        myrgb.green = 0xFFFF;
        myrgb.blue = 0;
        RGBForeColor( &myrgb );
        CopyBits(
          *myPixMapHandle, &(thePort->portBits),
          &boundsR, &destR, srcCopy, NULL );
        OffsetRect( &destR, 220, 0 );

        /* separate out the magenta component */
        myrgb.red = 0xFFFF;
        myrgb.green = 0;
        myrgb.blue = 0xFFFF;
        RGBForeColor( &myrgb );
        CopyBits(
          *myPixMapHandle, &(thePort->portBits),
          &boundsR, &destR, srcCopy, NULL );
        OffsetRect( &dstRect, -220, 220 );

        /* separate out the cyan component */
        myrgb.red = 0;
        myrgb.green = 0xFFFF;
        myrgb.blue = 0xFFFF;
        RGBForeColor( &myrgb );
        CopyBits(
          *myPixMapHandle, &(thePort->portBits),
          &boundsR, &destR, srcCopy, NULL );
        OffsetRect( &destR, 220, 0 );

        RGBForeColor( &savergb );

        DisposGWorld( myOffGWorld );
    }
}
```

Grayscale Remapping. An application can also use a search procedure to add color to a grayscale image. Suppose you have a grayscale image where grays in the range ($6000, $6000, $6000) to ($A000, $A000, $A000) represent a critical area and you would like to remap them to varying levels of red. Since the source image is known to be grayscale (that is, all RGB component values are equal), the search procedure needs to check only one component of the color to determine the intensity of gray. Listing 7-3 illustrates this technique.

Listing 7-3. Grayscale to color remapping example

```
pascal Boolean MapGrayToRed(
  RGBColor *targetRGB,
  long     *result )
{
  if ((targetRGB->red >= 0x6000) &&
     (targetRGB->red < 0xA000))
  {
    targetRGB->red = (targetRGB->red - 0x6000) << 2;
    targetRGB->green = 0;
    targetRGB->blue = 0;
  }
  return FALSE;
}
```

▶ Aliasing

There are basically two kinds of graphics: object graphics and bitmapped graphics. Applications like MacDraw deal with graphics as objects. In such "Draw" applications, a graphic item, such as a circle, is stored either as a position and a radius or simply as a bounding rectangle. The resolution of the object is usually determined at render time: If it's being printed on a LaserWriter, it will appear at 300 dpi. If it's drawn on the screen, it will appear at 72 dpi. However, if the graphic items are copied to the clipboard, they are typically put into a picture and the picture's data is then copied to the System scrap, maintaining its resolution-independent nature.

Painting applications, such as MacPaint and ArtToy (which was described in Chapters 5 and 6) deal predominantly with graphics as pixel images. When a circle is drawn, it's rendered into a PixMap or BitMap and the original definition of the shape is discarded: The circle is nothing more than a pattern of values scattered around the pixel image. The resolution is determined by the destination pixel-image buffer, which is typically 72 dpi. If this pixel image is printed on a graphics device of a different resolution, it will show the effects of aliasing, the manifestation of which is jagged "stair-step" edges where smooth curves or lines are expected.

Figure 7-4 shows the difference between a circle printed on a LaserWriter by a Draw application (hereafter called *Draw*) and one printed by a hypothetical painting application (or *Paint*). This illustrates the differences between rendering a nonrasterized circle (using **FrameOval**()) compared to rendering a rasterized circle (using **CopyBits**()).

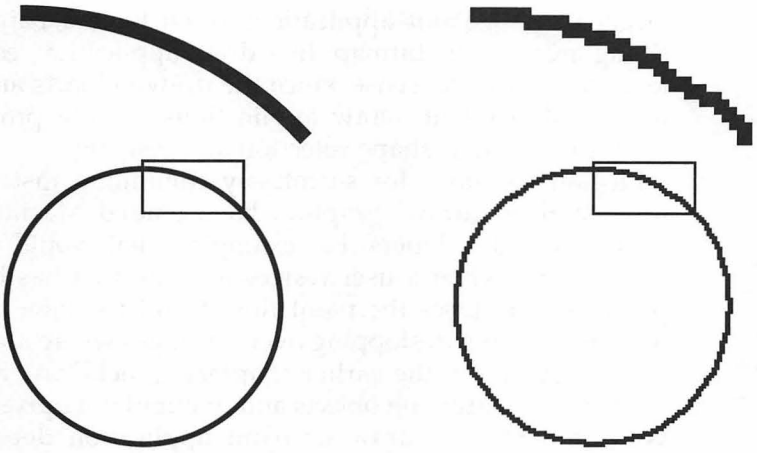


Figure 7-4. Circle printed in "Draw" versus "Paint"

Notice that Draw's circle is smooth, while Paint's circle has aliased edges. The difference stems from the way these programs store the circle: Draw retains the circle's definition (simply its bounds rectangle), whereas Paint images the circle into a PixMap or BitMap and uses **CopyBits**() to "draw" it into the destination graphics device. When you print, Draw renders the circle at the resolution of the destination device. Paint, on the other hand, simply uses **CopyBits**() to scale its pixel image up to (or down to) the printer's resolution. Since the printer has higher than 72–dpi resolution, the "jaggies" in the source pixel image are visible.

Note ▶

Since a LaserWriter is a bitmapped printer, Draw's circle also has aliased edges. These rough edges are much harder to see since they are at the printer's bitmap resolution (300 dpi), rather than the resolution of screen bitmap (72 dpi) as used by the paint application. If Draw's circle were rendered by a plotter, there would be no jagged edges at all. If Paint's application circle were rendered by a plotter, the jagged edges would still be evident.

Because of the way these applications store their images, paint and draw programs provide the user with different tools. For example, a paint application allows the user to modify individual pixels of the

target bitmap. Paint applications often have a paint bucket tool for filling areas of the bitmap. In a draw application, editing at the pixel level doesn't make sense, since the drawn objects are inherently resolution-independent. Draw applications usually provide shape-based operators, such as shape selection and resizing.

Elegant solutions for seamlessly combining rasterized (paint) and nonrasterized (draw) graphics have eluded Macintosh software designers and developers. For example, what would a paint/draw application do when a user resizes a shape that has been filled with a paint bucket? Does the paint flow to fill the new borders or does it remain unchanged, slopping over the edges where a shape was moved?

As discussed in the earlier chapters, QuickDraw has traps for dealing with both drawing objects and manipulating pixel images. Whether you are writing a draw or paint application depends on how the objects are rendered: If they're rendered into a pixel-image map when they are drawn, it's a paint application; if the application maintains a list of the shapes and updates the display by drawing the shape list, it's a draw application.

▶ Rasterizing

As discussed in Chapter 5, rasterization is the process of generating a pixel image from drawing operations. Aliasing is a term taken from signal processing and refers to a situation where a signal is *undersampled*. An important consequence of rasterizing operations is that a continuous graphic shape is inherently sampled in order to determine which pixels to set. Pixels values are, by nature, atomic, discrete, and otherwise indivisible entities. Determining which pixels to change and which values to assign to them is essentially a sampling operation. Aliasing is the most prominent, visible side effect of undersampling, and may be expressed in the geometric (x and y) axes as jagged edges or in the pixel value (z) axis as blotches of color in place of continuously shaded domains.

For example, the triangle wave shown in Figure 7-5, section A, is sampled in locations where the black dots appear. Unless the frequency content of the source signal is known, these samples do not provide enough information to distinguish between the waveform shown in Figure 7-5, section A, and that shown in Figure 7-5, section B. Based on these samples, the waveform shown in Figure 7-5, section B, takes on the identity, or is an alias of, the waveform shown in Figure 7-5, section A.

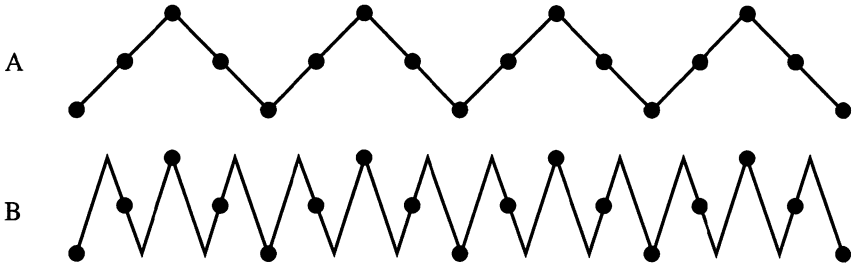


Figure 7-5. Example of aliased waveforms

Figure 7-6 shows how undersampling can manifest itself as aliasing in computer graphics. The rectangular grid represents pixel locations on the screen. The objective is to render the black object as it moves across the grid. Since the resolution of the object is higher than the resolution of the grid, the image is undersampled.

A first cut at a solution might be to turn the pixel element on if the object intersects the top-left corner of the pixel, and to turn the pixel element off otherwise. Using this method requires sampling the source image at points where the grid intersects. Below each grid is a graph showing the source waveform for the scan line marked with an arrow. In Figure 7-6, section A, one pixel would be turned on since the waveform is on at the sample point. In Figure 7-6, section B, the object does not intersect any horizontal grid line and will not appear at all.

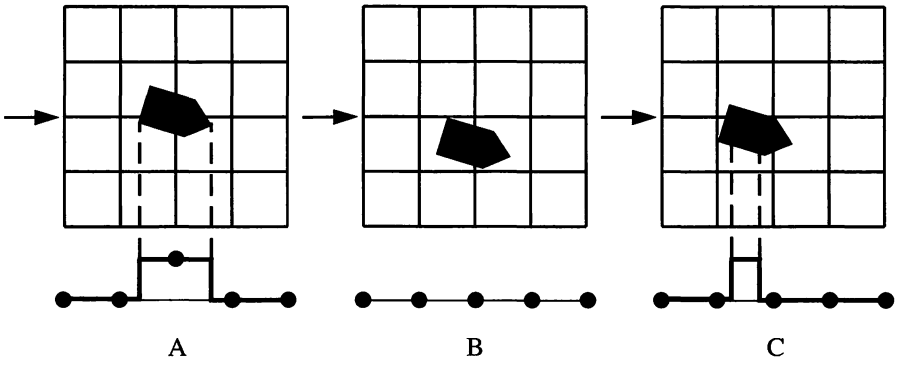


Figure 7-6. Signals and 2-D graphics sampling

In Figure 7-6, section C, the object intersects a horizontal grid line, but not at a location that is sampled. The waveform that is being sampled has a much higher frequency than the sampling rate. The problem is further compounded since the image is reconstructed with pixels that have only two values, on or off. Using this method of sampling and reconstruction, the resulting image flashes as it crosses the screen.

There are two basic strategies for reducing aliasing: increase the sampling rate or decrease the frequency of the source signal. Implementations of these strategies are collectively known as *antialiasing* techniques. The prevailing techniques rely on methods to functionally increase the sampling rate. Let's proceed with an example of oversampling.

Since the resolution of the output device (namely the screen) is fixed, each screen pixel must represent several samples. Suppose the output graphics device has four possible levels (0–3) for each pixel. If the sampling frequency is increased by a factor of two in each direction, the grids in Figure 7-7, section A, become the grids shown in Figure 7-7, section B.

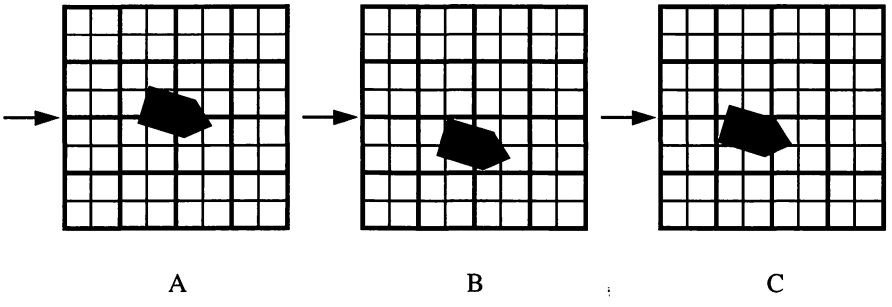


Figure 7-7. Antialiasing by oversampling

There are now four samples taken for each pixel, and the value of the resulting pixel is simply the number of points that intersect the object. Thus, if the object completely covers a pixel, the pixel is turned on (given a value of 3). If the object doesn't intersect the pixel at all, the pixel is turned off (given a value of 0). If the object covers about half the pixel, the pixel is a shade of gray (given a value of 1 or 2). If the object is rendered this way, it appears gray as it crosses the screen.

An even better result can be achieved with a still higher sampling frequency. If each screen pixel can take one of 256 values, the sampling frequency can be increased by a factor of 8 in each direction.

Drawing an object in this way requires knowing the shape of the object at a resolution higher than the screen resolution. Using gray values to account for round-off error where pixels are partially covered creates the illusion of higher resolution by recovering the lack of geometric (x-y) resolution in the pixel value (z) dimension.

▶ Aliased Regions

In addition to pixel images, regions can also be distorted by aliasing effects. Because regions capture the results of drawing commands rather than the shape's characteristic formula (for example, a circle: $x^2 + y^2 = r^2$), both **InsetRgn**() and **MapRgn**() impart distortions to the region's original shape. If a region is treated as a "snapshot" of line-oriented graphics operations, then the possibilities for region aliasing should become clear. A region contains scan line–oriented data, just like a pixel image, and is therefore subject to aliasing whenever it's geometrically scaled or rendered into a graphics device of a different resolution than the one in which the region was created. This means that all lines and curves get distorted when the region's dimensions are scaled in any direction, with the exception of perfect vertical or horizontal lines. Examine the framed regions in Figures 7-8 and 7-9.

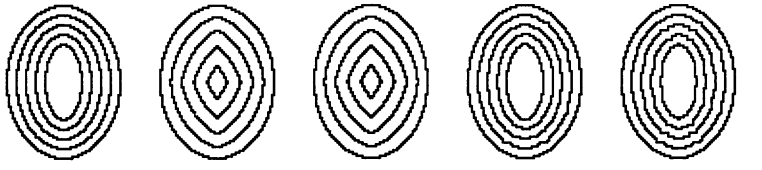


Figure 7-8. Region distortions, inset

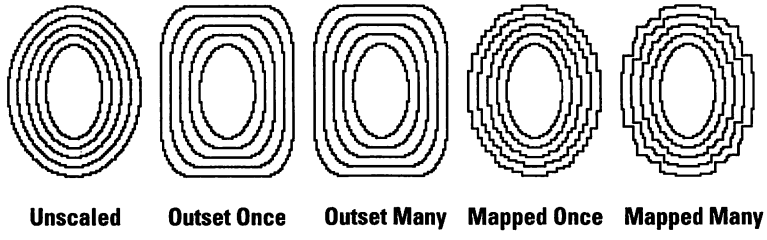


Figure 7-9. Region distortions, outset

The region on the far left of Figure 7-8 was created by capturing a **FrameOval**() trap after the bounding rectangle was scaled inward with **InsetRect**(). All the other regions were produced by capturing the innermost oval into a region and scaling the region itself. The regions in Figure 7-9 are produced identically to those in Figure 7-8, but they start with a smaller shape and scale outward. Because each shape is larger and has more detail than its predecessor, the aliasing effect is much greater than those produced by shrinking operations.

From these examples, we can draw a couple of conclusions about using regions to accurately represent an area:

- A region should be created by line and frame commands that most closely define that area *without* scaling.
- A region should be created at the highest resolution at which it's expected to be accurate.
- Regions shouldn't be relied on as the sole representation of a shape. Whenever possible, keep the original shape data available for future operations.

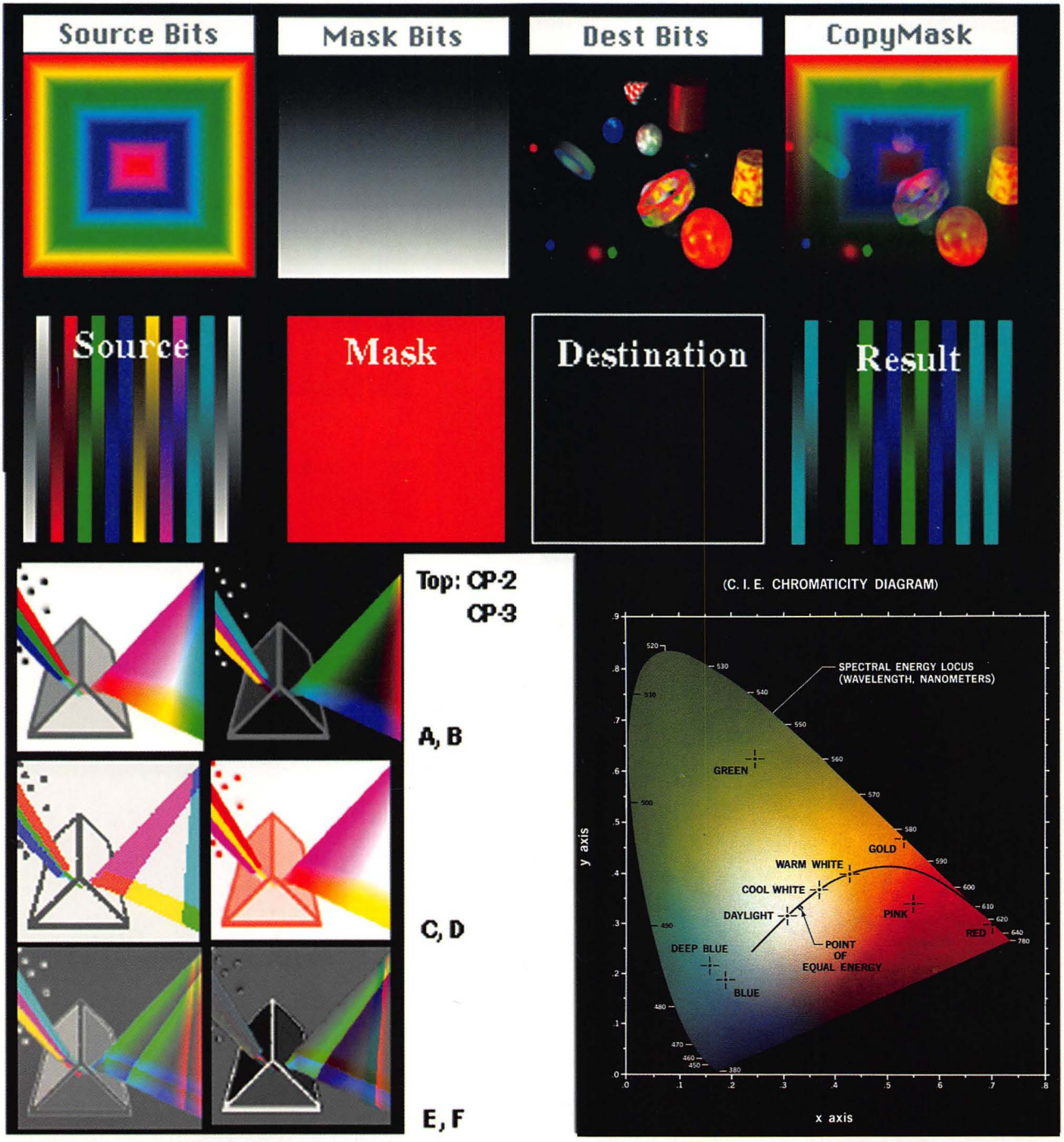
▶ Antialiased Lines

Aliasing is a side-effect of taking discrete samples of a continuous image. In any sampling operation, there is an implied quantization, or round-off, error. Figure 7-10 illustrates this error as manifested in line drawing operations. The problem is determining what to display for the pixels that are only partly covered by the line.

Although the goal may be to render a solid black line, the final output can be improved substantially by using gray values for the pixels that are partially covered. This antialiasing produces the illusion of higher resolution graphics by using sub-pixel techniques to lessen the effects of rounding errors. Figure 7-11 shows a magnified view of a line drawn with and without antialiasing.

Whenever a line is drawn in QuickDraw, it's rendered in one solid color. As with most image processing techniques, there are a number of ways to produce the desired result. In the first issue of *develop* magazine (Apple Computer, January 1990), Bruce Leak describes a simple way to do antialiased drawing. The procedure for achieving the gray pixels is similar to that used in a following sub-pixel stretching section. The following example outlines the technique.

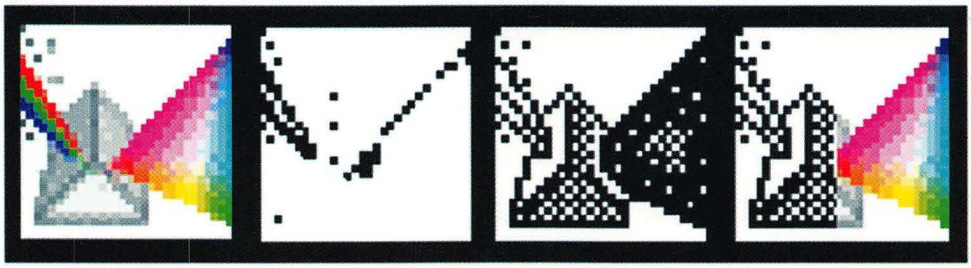
Suppose an application needs to draw an antialiased line on top of a 72-dpi PixMap. First it must determine the bounding rectangle of the



CP-4

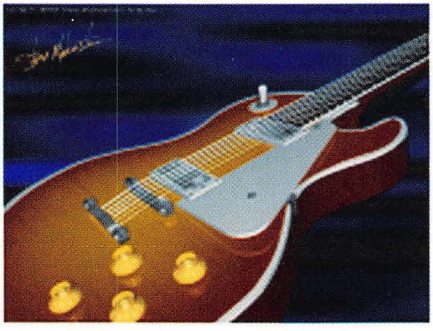
CP-1

The complete CIE color diagram. (Courtesy of GE Lighting.)

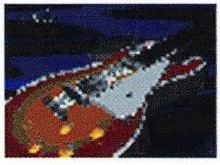


A B C D

CP-5



A

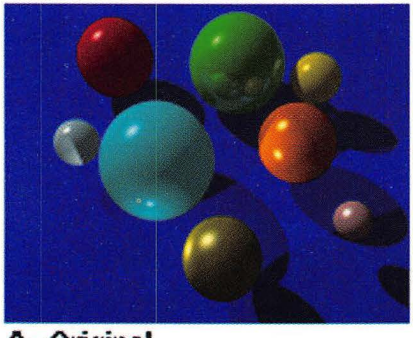


B

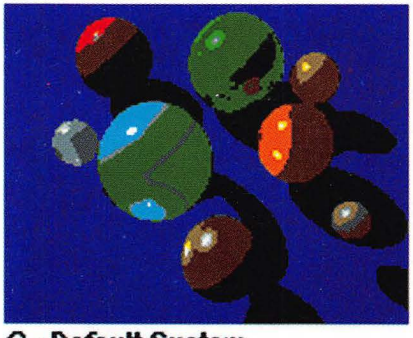


C

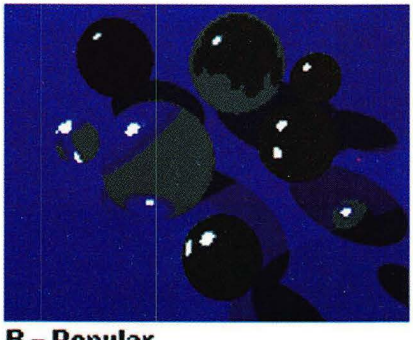
CP-6



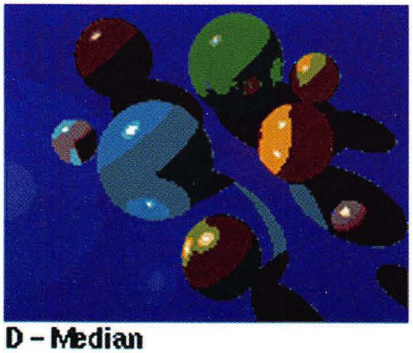
A – Original



C – Default System



B – Popular



D – Median

CP-7

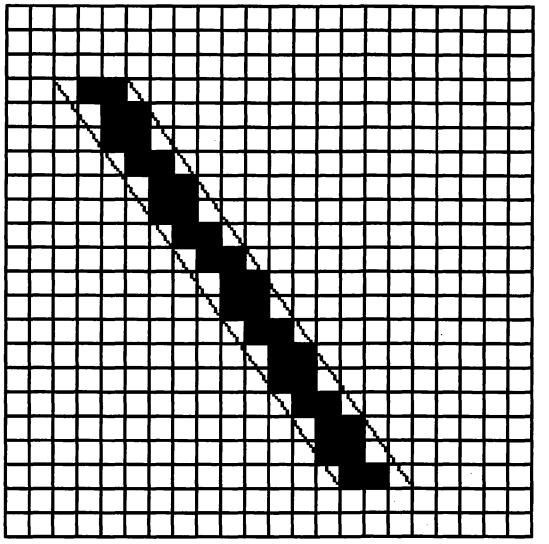


Figure 7-10. Sampling a line

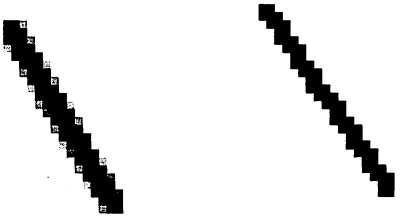


Figure 7-11. Magnified view of a line drawn with and without antialiasing

line. The **CopyBits**() trap can be used to copy the contents of that rectangle from the destination PixMap, scaling it up by some integral factor (four works well). Draw the line in the expanded PixMap. Note that the line has higher resolution than the PixMap by a factor equal to the amount the PixMap was scaled up. Finally, use dithered shrink to shrink it back to its original size.

The result is that the old pixel values will remain unchanged since they were expanded and shrunk by the same amount; an operation which doesn't change them. The line, which was rendered at the higher (effectively) resolution will be averaged with the surrounding pixels when it is shrunk. This produces smoother edges.

Important ▶ The key to antialiased drawing is that the representation of the image is known at a resolution higher than the destination resolution.

▶ Antialiased Text

Aliased text is all too familiar to computer users, largely because most typefaces are pixel images. This was the case with QuickDraw on the Macintosh prior to TrueType (available as an INIT for System 6 users and part of the System 7.0 release). To generate fonts of sizes for which no BitMap exists, QuickDraw simply picked the closest font size and stretched or shrunk the bits. Unfortunately, when a BitMap is resized, the resulting image is often far from pleasing.

TrueType fonts are algorithmically generated from line segment and curve descriptions, rather than produced from individual character bitmaps. This means that QuickDraw knows the shape of the character, much like QuickDraw knows the shape of a circle. Thus, when the character is scaled, its shape, rather than its BitMap, is scaled. This provides an excellent representation of the letter at all scaling factors. Furthermore, since the shape of the letter is known at a resolution (essentially infinite) higher than the display resolution (usually 72 dpi), text is an excellent candidate for antialiasing.

One technique for producing antialiased text is a direct extension of the discussion in the previous sections. The text is drawn at a certain size, for example four times larger than the intended destination. Next this pixel image is shrunk to the destination size using the ditherCopy mode. This produces a 4-bit grayscale PixMap which is solid black in the center of the letters and gray around the edges. When this PixMap is used as a mask argument to **CopyDeepMask**() (System 7.0 and later only) the result is smooth text.

To draw antialiased text using **CopyDeepMask**(), a source and a mask PixMap are required. The source PixMap can be generated as desired, and the CreateTextMask() routine creates the mask PixMap automatically. Figure 7-12 shows how CreateTextMask() generates the soft mask.

Notice in this figure that the edges of the text are gray. (This is particularly visible on the edges of the 'W'.) When this soft edge mask is used by **CopyDeepMask**(), the result is a soft, or antialiased edge.

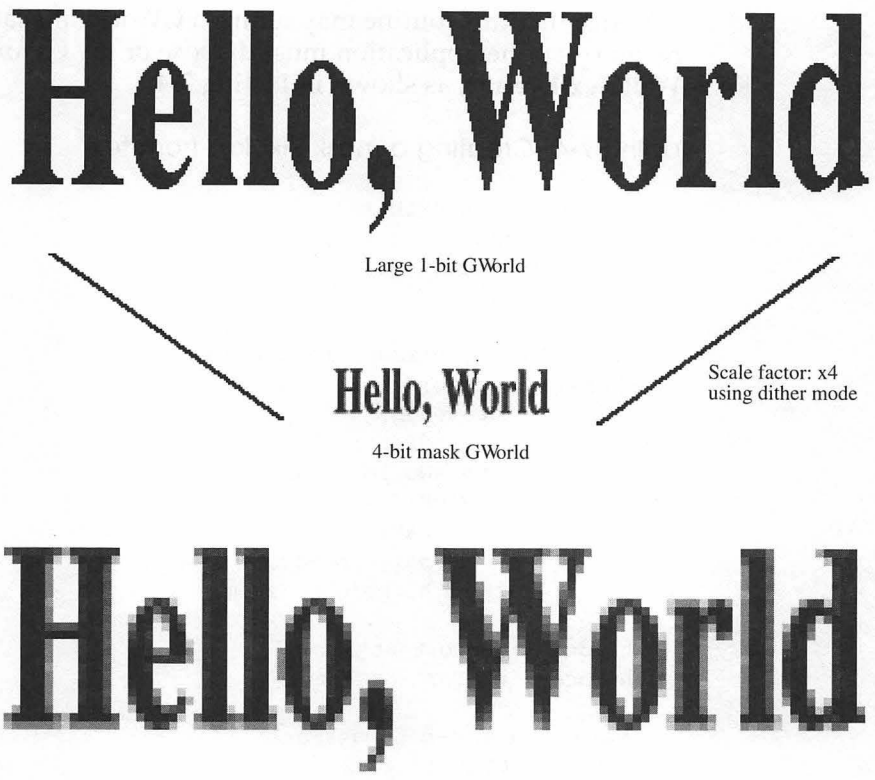


Figure 7-12. Creating a masking pixel image from text

The routine works as follows. First the text is imaged at four times the final resolution into a 1-bit GWorld. If there isn't enough memory in the application heap or temporary memory to allocate a GWorld this big, GWorlds of three times and then two times the final resolution are created. If all of these attempts fail, NULL is returned. Otherwise, the text is imaged into the 1-bit GWorld, scaled by a factor of 2, 3, or 4 (or whatever multiple the 1-bit GWorld is) in each direction. Then the text pixel image is copied and shrunk from large 1-bit GWorld to the 4-bit GWorld. This operation is performed by **CopyBits**(), using the ditherCopy transfer mode (discussed later in this chapter). Since the mask will consist only of grays, the 4-bit GWorld is given a grayscale ColorTable. The 1-bit GWorld has served its purpose, and is disposed. The 4-bit grayscale image just created is returned by CreateTextMask().

Notice that this routine may return a GWorld allocated in temporary memory, so the application must dispose of the GWorld before calling **WaitNextEvent()**, as shown in Listing 7-4.

Listing 7-4. Creating a mask PixMap from text

```
GWorldPtr CreateTextMask(
  char   *text,
  Rect   *boundsR,
  long   textLen )
{
PixMapHandle  bigPixMap, maskPixMap;
CTabHandle    ctab;
GDHandle      savedGD;
CGrafPtr      savedPort;
GWorldPtr     maskWorld, bigWorld;
Rect          aBigR;
OSErr         errStat;
short         theFont, theFace;
short         width, height, zoom;

  theFont = thePort->txFont;
  theFace = thePort->txFace;

  /*create the 4-bit maskWorld */
  ctab = GetCTable( 4+32 );
  errStat = NewGWorld( &maskWorld, 4, boundsR, ctab, NULL,
    useTempMem );
  if (errStat != noErr)
    errStat = NewGWorld( &maskWorld, 4, boundsR, ctab, NULL,
      0L );
  if (errStat != noErr)
  {
    DisposHandle( ctab );
    return NULL;
  }

  /* create the super-sampling bigWorld */
  zoom = 4;
  width = boundsR->right - boundsR->left;
  height = boundsR->bottom - boundsR->top;
  while (TRUE)
  {
```

```
    SetRect( &aBigR, 0, 0, width * zoom, height * zoom );
    errStat = NewGWorld( &bigWorld, 1, &aBigR, NULL, NULL,
      useTempMem );
    if (errStat != noErr)
      errStat = NewGWorld( &bigWorld, 1, &myRectBig, NULL,
        NULL, 0L );
    if (errStat == noErr)
      break;

    zoom--;
    if (zoom == 1)
    {
      DisposGWorld( maskWorld );
      return NULL;
    }
  }

  /* render the text into bigWorld */
  GetGWorld( &savedPort, &savedGD );
  SetGWorld( bigWorld, NULL );
  TextFont( theFont * zoom );
  TextFace( theFace );
  bigPixMap = GetGWorldPixMap( bigWorld )
  LockPixels( bigPixMap );
  TextBox( text, textLen, &aBigR, teJustLeft );

  /* image into destination 4-bit maskWorld */
  maskPixMap = GetGWorldPixMap( maskWorld );
  LockPixels( maskPixMap );
  SetGWorld( maskWorld, NULL );
  CopyBits(
    *bigPixMap, *maskPixMap, &aBigR, boundsR,
    ditherCopy | srcCopy, NULL );
  UnlockPixels( maskPixMap );

  /* clean up */
  DisposHandle( ctab );
  DisposGWorld( bigWorld );
  SetGWorld( savedPort, savedGD );

  return maskWorld;
}
```

Using this technique for drawing antialiased text works great for drawing to deep PixMaps, but is less than optimal in many other situations, such as printing. If you maintain the text rather than image it into bits, the print driver can do its best to produce a high resolution printout. Most printers provide the option of substituting an outline font for a bitmap font at print time if an outline version of the requested font isn't available.

▶ Review of Key Points

The pixel processing traps **CopyBits()**, **CopyMask()**, and **CopyDeepMask()** provide great flexibility in transferring images. When you understand how the explicit trap arguments and implicit QuickDraw state affect the outcome of the trap call, you have a powerful tool at your disposal for performing a number of standard, and nonstandard, image processing functions.

System 7.0 provides a considerable amount of new functionality to these traps. Averaged shrink, colorizing, search procedures, and the ditherCopy mode provide simple and flexible ways of producing varied, high-quality images.

▶ Depth Conversion and Dithering

Depending on the operation, there is quite a bit of processing that can occur inside the Stretch traps. If the source and destination PixMaps are of different depths, then the Stretch traps must perform a depth conversion. A depth conversion involves finding an appropriate mapping between the color values in the source and those available on the current graphics device.

▶ Depth Conversion Cases

There are four distinct types of depth conversion:

- indexed to indexed
- direct to indexed
- indexed to direct
- direct to direct

Depth conversion to an indexed destination can happen in two different ways, depending on whether or not the dither flag is set. The most basic form of depth conversion replaces each source pixel with its closest representation in the destination. This is how the Stretch traps produce destination pixel values when the dither flag isn't set.

Indexed to Indexed

For indexed-to-indexed depth conversions, Stretch creates a translation table. For example, if a 16-color (4-bit) PixMap is copied into a 256-color (8-bit) PixMap, the Stretch traps create a color table with sixteen entries, one for each possible value in the source. Once the table is built, depth conversion occurs with a simple table lookup. If a search procedure is installed, it's called for each entry in the table.

Indexed to Direct

Indexed-to-direct depth conversion is straightforward: Since the maximum number of unique source pixel values is 256, a color table is created to perform the conversion via lookup. For 1-bit to N-bit depth conversions, the Stretch traps are particularly fast, since there are only two possible output values. As before, any installed search procedure is called once for each entry in the color table.

Direct to Indexed

Setting up a mapping table does not work for direct-to-indexed depth conversions because of the prohibitively large number of possible source (direct) pixel values. For indexed sources, the largest possible translation table is 256 bytes; for 32-bit direct color (8-8-8), a translation table of 16.8 million entries would be necessary, not counting the high byte. For execution time and memory limitations, this simply isn't practical.

Direct-to-indexed depth conversion is accomplished by calling the search procedure (if one is installed) or using inverse table lookup for every source pixel, as described in Chapter 3. Unlike **Color2Index**(), which accommodates hidden colors, the Stretch traps use the inverse table without the hidden color mechanism for performance reasons. Thus, depth conversion from direct-to-indexed pixel images reduces to a simple lookup in the inverse table.

If a search procedure is present, it's called for every pixel in the source. For indexed sources, the search procedure (on each intersecting graphics device) is called a maximum of 258 times: once for every

color in the source PixMap (256 maximum), and once for the foreground and background colors. For direct sources, the search procedure is called for *every pixel* in the source. For a typical screen-sized (640-by-480) pixmap, the search procedure is called 307,202 times: once for every pixel in the source, 307,200 times, and then once for the foreground and background colors.

Note ▶

The table lookup associated with direct-to-indexed depth conversion is slightly more complicated than the table lookup necessary for indexed-to-indexed depth conversion. The reason is that the index for the inverse table lookup is calculated using a certain number of high bits (specified by the inverse table resolution) from each color component (R, G, and B) of the source pixel. In the indexed-to-indexed case, the source pixel value itself is used as the index.

Direct to Direct

There are two possible cases, 32-bit–to–16-bit and 16-bit–to–32-bit depth conversion. In the direct-to-direct case, table lookup is again impractical, due to the potentially large number of unique source pixel values. The 32- to 16-bit conversion is accomplished by truncating the 32-bit value (8-8-8) to a 16-bit value (5-5-5).

The 16- to 32-bit conversion is accomplished by replicating bits. Figure 7-13 shows a 5-bits-per-component replication converted into an 8-bits-per-component replication, as well as a 5-bits-per-component replication to a 16-bits-per-component replication.

This figure shows that the 5 source bits of the color component are used as the top 5 bits of the destination. The low 3 bits of the destination are the top 3 bits of the source. In the 5-to-16 case, the 5 source bits appear in the result three times, and the top source bit appears four times. At first it may seem that simply copying the top 5 bits is good enough, until you realize that it is impossible to get pure white ($FF hexadecimal) using this technique since the low bits are always 0. By replicating the top bits into the low bits, the 5-bit source pixels map linearly to the entire 8-bit- or 16-bit-per-color component destination.

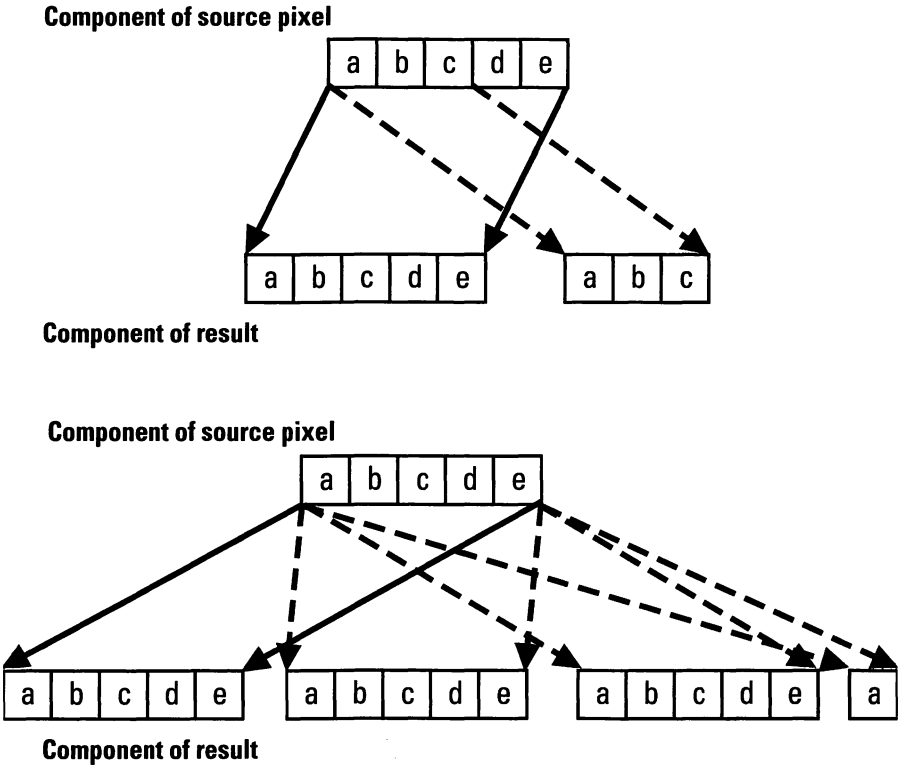


Figure 7-13. 5-to-8-bit and 5-to-16-bit component conversion using pixel replication

▶ What Are Dithering and Error Diffusion?

When copying into a pixel image whose pixel size is less than that of the source image, some amount of image data is lost in the depth conversion process. This loss of image data is a form of aliasing and tends to produce blotchy areas in the destination where smooth shaded areas existed in the source. The difference between source and destination color values is a quantization (or roundoff) error. If this difference error is summed up and redistributed over neighboring pixels, then the resulting image appears substantially similar to the source image. This technique is called *dithering* or *error diffusion* and can be invoked when calling the stretch traps. The sixth bit in the transfer mode is known as the dither flag and can be set by adding or bit-oring the ditherCopy value of 64 into the transfer mode, as shown in Listing 7-5.

Listing 7-5. Setting the dither flag

```
void UseDitherMode( void )
{
short  newTransferMode;

   newTransferMode = srcCopy | ditherCopy;
   CopyBits(
      *srcPM, *destPM, &srcR, &destR,
      newTransferMode, (RgnHandle)NULL );
}
```

▶ How CopyBits() Dithers

There is a major shortcoming with the depth-conversion techniques just discussed. The problem is most pronounced when converting a deep source pixmap to a destination that has few colors. For example, converting a 50 percent gray pixmap to a 1-bit destination results in solid white. A more desirable result can be achieved through dithering the destination by alternating black and white pixel values, thus preserving the source image's overall luminance content.

The same applies to color transfers. If the source contains a smooth red ramp and the destination has only a few different values of red, the image resulting from the depth conversion will have a blotchy appearance rather than the smooth transition in the source, as shown in Figure 7-14.

Dithering is a process that preserves the color content of the source PixMap. There are a number of different algorithms for producing dither effects. QuickDraw's algorithm was chosen on the basis of speed (it has to be fast) and memory (it should use a minimal amount of memory). The basic theory of QuickDraw's dithering algorithm is that if a requested color is not available on the destination, the closest approximation is used and the difference between the color used and the requested color is kept in an error variable. This error is added to surrounding pixel values so that the color content of an area is maintained. The process is then repeated.

Depending on the QuickDraw version, the error propagation is performed in different ways. In Color QuickDraw, half the error is diffused (spread around) to those pixels to the right and left, and the other half is spread downward. The error is spread to the right on even scan lines and to the left on odd scan lines.

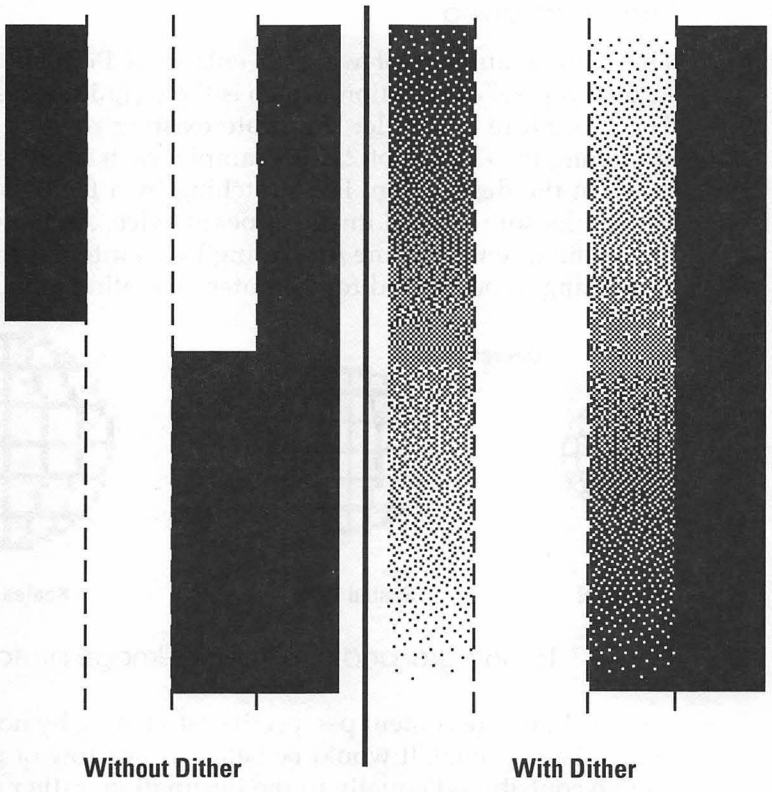


Figure 7-14. Example of dithering

Because of the extra processing that dithering requires, dither-copying an image is considerably slower than a non-dithered copy operation. The trade-off between image quality and execution time is significant and should be considered when designing image-intensive applications.

▶ Image Resizing

When the source and destination rectangles are different sizes (in x and y, not pixel depth), the Stretch trap stretches or shrinks the source PixMap to fill the destination. Since these operations are independent in the horizontal and vertical dimensions (for example, shrinking in one dimension operates independently of scaling, if any, in the other dimension), only the vertical case is discussed. How this operation is performed and the effect of the dither flag (yes, the dither flag affects shrinking) is discussed in this section.

Image Stretching

There are a number of ways to enlarge a PixMap. The easiest (and fastest) is pixel duplication, which is the technique used by QuickDraw. The procedure is simple: Duplicate existing rows of pixel values. For stretching by a factor of 2, for example, each source scan line appears twice in the destination. For stretching by a factor of 1.5, every other row in the source pixel image appears twice. As shown in Figure 7-15, this technique works fine for scaling by an integral amount, but leaves something to be desired for nonintegral scaling amounts.

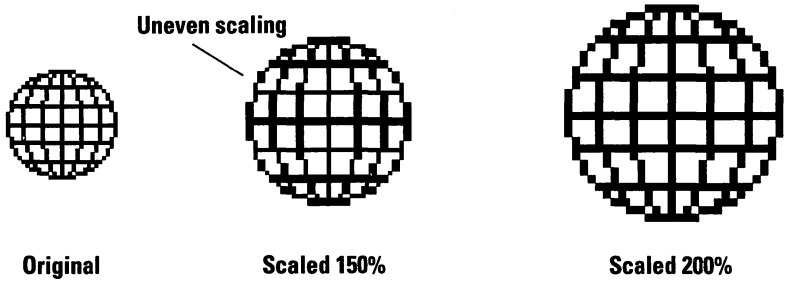


Figure 7-15. Integral and non-integral image stretching

From an image content perspective, stretching by noninteger amounts is less than optimal. It would be better if each row of pixel values in the source contributed equally to the destination, rather than (as in Figure 7-15) having some rows contribute twice as much as others. There is a way to achieve an equal contribution when stretching, but you'll have to wait until we're done discussing shrinking to find out how.

Image Shrinking

No Dithering. As with stretching, there are a number of different ways to shrink an image. It may seem reasonable to simply drop rows of pixel values when shrinking — basically the opposite operation of stretching, where rows of pixels are replicated. This is a bad idea because images could disappear completely. What if you shrink a pattern of alternating black-and-white horizontal lines by a factor of 2? You would get either an all-black or an all-white image, depending on the alignment of the lines.

Since most images are drawn on a white background, and it's obviously undesirable to have an image disappear when shrinking, QuickDraw preserves the black content of an image when shrinking without dither. The way this occurs depends on the depth of the source PixMap.

1-Bit Images. To shrink a 1-bit image, QuickDraw uses the Boolean OR operator to merge rows of pixel values. For example, to shrink vertically by a factor of 2, every two scan lines are OR'd together. Since black pixels are represented by a bit being turned on, the result moves toward black. Figure 7-16 shows a portion of an image processed in this way. In particular, notice that the gray (black-and-white dithered) background ends up black.

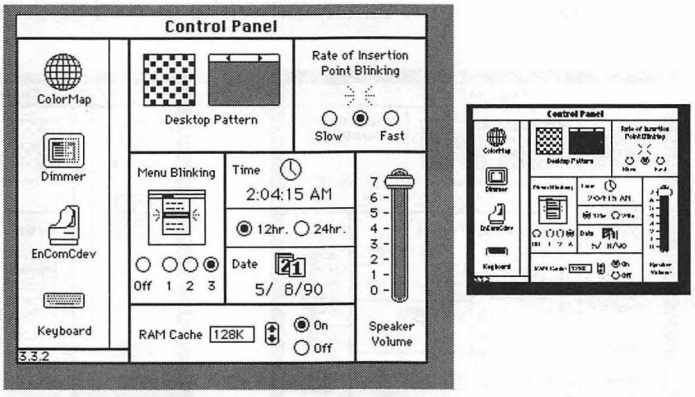


Figure 7-16. 1-bit shrinking by 50 percent

Indexed Images. For these source depths, OR-ing the indexes together does not make sense: Colors can occur in the result that weren't in the source. This is obviously undesirable. Again, QuickDraw's goal is to preserve the darker colors. Unfortunately, there is no fast way to do this, and the shortcut QuickDraw takes is to simply use the largest pixel index value. For a grayscale color table, where $0 is white and $FF is black, this achieves the desired result. In other cases, the result depends on the ordering of colors in the destination GDevice's ColorTable. While this technique is far from producing an optimal shrink, it's very fast and produces adequate results.

Direct Images. As of System 7.0, both 16- and 32-bit images are averaged when shrunk. Prior to the 7.0 version of QuickDraw, 16-bit images were not averaged when shrunk. Since direct color images deal directly in color space rather than in index space, it's relatively easy and fast to average neighboring colors. Thus, shrinking an image that consists of alternating red and blue lines by a factor of 2 will produce an image result of solid purple.

Dithered Shrinking. It may come as a surprise, but setting the dither flag affects the way images are shrunk. Setting the dither flag directs QuickDraw to convert the source to 32-bit data, thus causing the pixel values to be averaged when shrinking. Figure 7-17 shows the results of shrinking a 1-bit image by a factor of 2 with and without dithering: In both cases the destination pixel image was 32 bits deep. The dithered version contains black and white as well as three different grays.

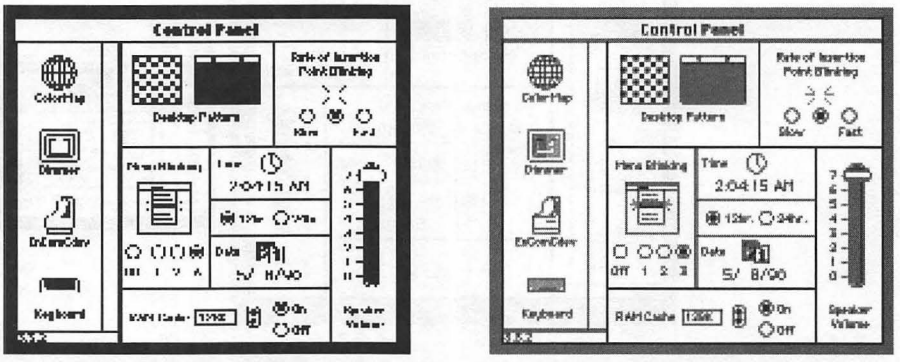


Figure 7-17. Shrunken 1-bit source (left: without dither; right: with dither)

CP-6 shows an 8-bit image drawn with (CP-6B) and without (CP-6C) the dither flag set. While the difference between the images is not as drastic as in the 1-bit case, the dithered version is clearly superior to the nondithered version. In the dither case, the black is averaged with the surrounding colors, producing an image that resembles the source much more closely.

Setting the dither flag when shrinking only affects the way indexed sources (1-, 2-, 4-, and 8-bit) are treated; direct sources are already averaged when they are shrunk.

Averaging when shrinking is considerably slower than shrinking without setting the dither flag. You decide what the trade-off should be: fast drawing with lower quality or higher-quality images that take longer to render.

You will recall from the earlier section on depth conversion that the dither flag has the same effect: Setting it produces a higher-quality image but it takes longer to render. In that case, dithering affected how the result is imaged into indexed destinations. In the case of shrinking, the dither flag describes how the source should be treated. This meaning of the dither flag in the transfer mode is the result of the structure of the underlying code and the history of its development. We agree that its operation is a bit bizarre.

Dithered Smoothing

In a previous section, we discussed the fact that images are stretched via pixel replication. This can produce suboptimal results for noninteger scaling since parts of the source image affect the result more than other parts. This effect can be minimized, or even eliminated. The procedure is simple, and the following outlines an example of dithered smoothing.

Suppose you want to scale an image by a factor of 1.5. If a Stretch trap is called with a destination rectangle argument 1.5 times larger than the source rectangle, every other line is replicated.

Ideally, every 1.5 scan lines would be replicated. Stretching in such a manner is called *subpixel stretching,* since fractional parts of source pixels are used to determine the color of each pixel in the result. While QuickDraw does not provide a way to do this directly, you can achieve the same result by first scaling the image up (by a factor of 3 in this case), and then shrinking it down with dither (by a factor of 2 in this case). The factor of 3 scaling produces an image where each source scan line contributes to three lines in the result. The factor of 2 shrinking (with dithering) produces a result where two source scan lines are averaged to produce one scan in the result. Thus, each line in the final image is generated by 3/2, or 1.5, scans of the source image, as desired. Figure 7-18 shows the same image scaled up vertically by a factor of 1.5 with and without subpixel stretching.



Figure 7-18. 1-bit image stretched by a factor of 1.5, with and without subpixel stretching

▶ Dithered Patterns

Drawing shapes with a solid color directs the Color Manager to determine the closest available color before rendering the shape; if the nearest available color is significantly different from the requested one, then the shape's appearance will be appreciably different from the intended one. To remedy this shortcoming, QuickDraw allows for the creation and usage of dithered, RGB-relative patterns. Applications can use dithered patterns to approximate colors that aren't available on the destination graphics device. The **MakeRGBPat**() trap creates a pattern that approximates a given RGBColor, and has the following interface:

```
void MakeRGBPat(
  PixPatHandle     thePixPat,
  RGBColor         *targetRGB );
```

As with other PixPat traps, the pattern must be previously allocated using **NewPixPat**(). The pattern generated by this trap will produce a close match to the specified color at the time the shape is rendered. If a change in the GDevice's color mapping occurs, then the pattern will automatically compensate for it when the shape is rendered, without having to recalculate it.

▶ Using Deep PixMaps on Classic QuickDraw Machines

A classic problem (no pun intended) that developers face is figuring out which version of QuickDraw to develop for. Applications that run on the Macintosh Classic family have a much greater potential market but can't use the features of Color QuickDraw. The result is three types of programs: Those that run on all machines but use only Classic QuickDraw traps, those that run only on Color QuickDraw machines, and those that run better on color machines but also do a pretty good job on Classic QuickDraw machines. Obviously, it's most desirable to fall into the last category.

For example, if you are writing a scanner application you need the feature set of Color QuickDraw, but you would like it to run on a Classic QuickDraw machine as well. While the job of reimplementing all of Color QuickDraw for 68000-based machines is a formidable task (even Apple has avoided it so far), there are some tricks you can use to get acceptable results on Classic QuickDraw machines with a minimum of effort.

One of the most common problems is displaying a PixMap on a Classic QuickDraw machine, since there is no provision to deal with anything other than BitMaps. Fortunately, there is one exception to this rule: Classic QuickDraw can display pictures containing PixMaps of depths up to 8 bits, and as of System 7.0, any depth. Thus, you can construct a picture containing deeper PixMaps manually and then display the picture using **DrawPicture**().

Note ▶

Pictures containing bad data can crash the Macintosh. An application must ensure that the pictures it creates work on both Classic and Color QuickDraw machines. It's essential that you understand the picture format as described in *Inside Macintosh*, Volume V, before attempting this technique.

A scanning application is an instance of when this technique might be useful. This would allow a user to create color pictures using only a black-and-white Macintosh. Obviously, the resulting scans cannot be viewed in color, but a black-and-white rendition may be adequate until the picture is exported to a paint program on a Color QuickDraw machine. The **DrawPicture**() trap on Classic machines handles all the color-to-monochrome translation for you.

Chapter 8 gives an example of creating a picture containing a PixMap. You can use this routine (or a similar one) on a Classic QuickDraw machine to display deep PixMaps using the **DrawPicture**() trap.

▶ The BitMapToRgn() Trap

The first release of 32-bit QuickDraw included the **BitMapToRgn**() trap. This useful trap creates regions from BitMaps and 1-bit PixMaps, that describe the area equal to the black pixel values in the source pixel image. This capability can be quite useful for painting and other pixel image editing applications. The interface to this trap is as follows:

```
OSErr BitMapToRgn(
  BitMap      *sourceBitMap,
  RgnHandle   destRgn );
```

Using this trap is simple; however, a few rules have to be observed. As with other traps that manipulate regions, the region must be a valid region, preferably created via a call to **NewRgn**(). Secondly, because of the size and complexity limitations on regions (as discussed in Chapter 2), the trap's return value should always be checked to determine that the resulting region is in fact valid. Lastly, the System 6 versions of this trap can generate corrupted regions *without* reporting the error if the destination region would be larger than 32K.

Listing 7-6 shows an example of using this trap to produce a region equal in area to the pixels affected by text-drawing traps.

Listing 7-6. Example of using BitMapToRgn()

```
RgnHandle RegionFromText(
  char   *theText,
  long   textLen,
  Rect   *boundsR,
  short  fontID,
  short  fontSize,
```

```
  short  textStyles,
  short  textJust )
{
BitMap      offBits, savedBits;
GrafPort    portMem;
GrafPtr     textPort, savedPort;
RgnHandle   theRgn;
OSErr       errStat;

  /* check argument validity as much as possible */
  if ((theText == NULL) || (textLen <= 0L) ||
       (boundsR == NULL) || EmptyRect( boundsR ))
    return NULL;

  /* create a BitMap to image the text into */
  offBits.bounds = *boundsR;
  errStat = BitMap_Create( &offBits );
  if (errStat != NULL)
    return NULL;

  /* create an offscreen port and install the BitMap */
  GetPort( &savedPort );
  textPort = &portMem;
  OpenPort( textPort );
  savedBits = thePort->portBits;
  SetPortBits( &offBits );
  ClipRect( boundsR );

  /* set up text parameters and render the text */
  TextFont( fontID );
  TextSize( fontSize );
  TextFace( textStyles );
  TextBox( theText, textLen, boundsR, textJust );

  /* (hopefully) derive a region from the text bit image */
  theRgn = NewRgn();
  errStat = BitMapToRegion( theRgn, &offBits );
  if (errStat != NULL)
  {
    DisposeRgn( theRgn );
    theRgn = NULL;
  }

  /* clean up */
  SetPortBits( &savedBits );
  ClosePort( textPort );
```

```
    SetPort( savedPort );
    BitMap_Dispose( &offBits );

    return theRgn;
}
```

▶ Device Loops

A device loop is the code that performs some drawing operation for all the screen devices the drawing intersects. For example, when Color QuickDraw draws a rectangle on the screen, it first checks which screen devices the rectangle will intersect. It then issues a drawing call to each of the affected devices. There are two device loops internal to QuickDraw (one for objects and one for PixMaps), and one that can be used by applications. The application-accessible version is contained in the **DeviceLoop**() trap and is available on all machines as of System 7.0.

It's important to understand that the **DeviceLoop**() trap is an exception to the QuickDraw graphics model where an application makes drawing calls and QuickDraw determines how those calls will be instantiated on the available hardware. Rather, **DeviceLoop**() provides a way for an application to determine how drawing will occur to devices of different depths. The rationale for this is that you may not be happy with the way QuickDraw maps an image to a particular depth. For example, if you are drawing a picture that contains primarily shades of blue, QuickDraw may draw an entirely black rendition (or a dithered version that is unacceptable to you) on a 1-bit device. With **DeviceLoop**(), you can tell QuickDraw how to draw to each destination depth.

Color plates CP-5A through CP-5D show four magnified versions of the 32-bit QuickDraw icon. CP-5A is the icon displayed on an 8-bit screen. This rendition is fine. CP-5B is the icon displayed on a 1-bit screen. Because QuickDraw maps most of the colors in the icon to white, the 1-bit rendition looks awful. CP-5C shows the desired 1-bit icon. With the **DeviceLoop**() trap, an application can draw the 1-bit version to 1-bit screens (and maybe 2- and 4-bit screens), and the 8-bit version to the rest. Thus, if the icon spans an 8-bit and a 1-bit device, it will appear as in CP-5D.

Note ▶

As of System 6.0.5, the **PlotCIcon**() trap automatically does this for icons. In System 7.0, an application can achieve the same results with the **DeviceLoop**() trap.

▶ The DeviceLoop() Trap

The **DeviceLoop**() trap calls the drawing procedure for each active screen device that intersects the drawing region. When the drawing procedure is called, the current port's *visRgn* will be the intersection of the original *visRgn* and the intersecting portions of the device.

The interface for this trap is as follows:

```
void DeviceLoop(
  RgnHandle          drawingRgn,
  ProcPtr            drawingProc,
  long               userData,
  DeviceLoopFlags    flags );
```

The drawing procedure is declared as:

```
pascal void DrawingProc(
  short      depth,
  short      deviceFlags,
  GDHandle   targetGDH,
  long       userData );
```

where the depth is the pixel size of the device and the flags are the GDevice's *gdFlags*. The "userData" is the userData that was passed to **DeviceLoop**().

Note ▶

Apple is striving to have a consistent calling interface across the Macintosh product line. To this end, the **DeviceLoop**() trap is available on both the Classic and the Color QuickDraw machines. When the drawing procedure is called on a non-Color QuickDraw machine, the "targetDevice" argument is set to NULL.

The "flags" argument passed to **DeviceLoop**() specifies how the screen devices should be treated. You can select what constitutes similar devices and whether or not similar devices should be grouped together. Similar GDevices have:

- the same setting of the "gdDevType" bit in their *gdFlags* field (monochrome/color),
- the same pixel depth, and
- optionally, the same color table seed.

When similar GDevices are grouped, the first one found is assumed to be representative of the rest of the GDevices and is passed to the supplied drawing routine as the "targetDevice" argument.

Setting the "singleDevices" flag directs **DeviceLoop()** not to group similar GDevices. When this flag is set, the routine pointed to by "drawingProc" will be called once for each GDevice that intersects the "drawingRgn."

The "dontMatchSeeds" flag tells **DeviceLoop()** not to consider color table seeds when comparing GDevices for similarity. This flag is ignored if the "singleDevices" flag is set.

Finally, by setting the "allDevices" flag, **DeviceLoop()** will ignore "drawingRgn" and call the supplied drawing routine for all active screen GDevices. In this case, the port's *visRgn* is not changed between calls to "DrawingProc." Either the "singleDevices" or the "dontMatchSeeds" flags can be used in conjunction with the "allDevices" flag.

▶ Using DeviceLoop(): A Source Code Example

The purpose of the **DeviceLoop()** trap is to provide a way for applications to perform custom drawing to screens of different depths. For example, suppose a paint application has a color wheel in one of its windows. The color wheel is a 32-bit image stored in a picture. The problem is that QuickDraw's rendition of the picture on 1-, 2-, and 4-bit screens is less than ideal. **DeviceLoop()** provides an opportunity for you to supply a custom picture if the color wheel intersects screens of these depths. On 8-, 16-, and 32-bit screens, QuickDraw does an adequate job with the 32-bit picture.

Listing 7-7 shows an example of how to use **DeviceLoop()** to implement the update procedure for such a window. Note that the size of the window is fixed and is kept in the global variable "gColorWheelRect."

Listing 7-7. DeviceLoop() example

```
void UpdateColorWheel(
     RgnHandle   invalidRgn )
{
  BeginUpdate();
    DeviceLoop(
      invalidRgn, (ProcPtr)&DrawColorWheel,
      0, 0 );
  EndUpdate();
}
```

The drawing routine might look something like the one shown in Listing 7-8.

Listing 7-8. DeviceLoop() custom drawing routine

```
pascal void DrawColorWheel(
  short       depth,
  short       deviceFlags,
  GDHandle    targetGDH,
  long        userData )
{
PicHandlemyColorWheel;
short  resID;

  resID = depth;
  if (depth > 4)
    resID = 32;
  myColorWheel = (PicHandle)GetResource( 'CLRW', resID );

  if (myColorWheel != NULL)
    DrawPicture( myColorWheel, &gColorWheelRect );
}
```

▶ Summary

The **CopyBits**() trap provides a simple mechanism for performing standard image processing operations such as depth conversion, switching color spaces, image resizing, and colorizing. Furthermore, search procedures provide an easy mechanism for extending color operations. For example, an application can employ search procedures to perform color separations in a variety of colorspaces, cyan-magenta-yellow-key (CMYK, for example). The "key" is typically a binary or monochrome black-and-white overlay image.

The **CopyDeepMask**() trap provided in System 7.0 extends QuickDraw's image processing capabilities still further. **CopyDeepMask**() provides great flexibility for combining images. While these traps need to be supplemented if you want to develop a Hollywood-style special effects program, they provide an excellent toolbox for basic image manipulation.

The best way to perform custom image processing is to allocate an offscreen GWorld and then operate on those pixels directly. You can then use **CopyBits**() to transfer the image to the screen. For an even

higher degree of control for drawing to the screen, you can use the **DeviceLoop**() trap to perform drawing operations in a manner specific to a destination pixel depth. While your application may prefer a certain pixel depth over others, it should appear sensible and convey meaning, regardless of pixel depths or color/grayscale mapping.

8 ▶ All About Pictures

Pictures and the PICT format have been part of QuickDraw since the introduction of the Macintosh in 1984. The design goal of the original picture implementation (now called PICT version 1) was that pictures had to be compact. Due to the memory limitations of a 128K machine, this format utilized 1-byte opcodes, leaving little or no room for expansion.

The introduction of the Macintosh II and Color QuickDraw necessitated the expansion of the PICT data format. Thus, the version 2 format picture was introduced. This format facilitates future expansion and has been extended several times with subsequent System software releases.

When version 2 pictures were introduced, Classic QuickDraw was revised so that these new pictures could be displayed on older machines. While classic QuickDraw could display both version 1 and version 2 pictures, only version 1 (PICT) pictures could be created. Color QuickDraw, on the other hand, created a version 1 picture if the picture was created in a monochrome GrafPort, and a version 2 picture if the picture was created in a CGrafPort. In System 7.0, the **OpenCPicture**() trap allows "Classic QuickDraw only" machines to create version 2 pictures. Regardless of how the picture was created or what version it is, the **DrawPicture**() trap can be used to display it.

This chapter is all about pictures. It discusses how pictures are created, how they can be analyzed using the new Picture Utilities, and other advanced topics related to the use and handling of pictures.

This chapter also includes code for creating a picture that contains a deep PixMap manually (without the **OpenPicture**() trap). This is useful on Classic QuickDraw machines since it allows PixMaps to be displayed using **DrawPicture**().

▶ Picture Analysis with the Picture Utilities

Chapter 2 introduced the standard QuickDraw method of data interchange, the picture, or PICT. That chapter described a picture as a container for almost any QuickDraw drawing commands, much as a polygon is a container of line drawing commands. This section expands on that discussion and describes a set of traps, the Picture Utilities, for analyzing pictures.

When a user imports a picture into an application, it can be treated in a number of ways. Prior to System 7.0, the standard way was to image the picture into a PixMap and treat it as pixel data. The problem with this approach is that you are forced to treat the picture as a black box and can only guess its contents. For example, it's difficult to determine what depth PixMap to image a picture into. If the PICT contains only black-and-white data, a 1-bit deep PixMap will do; if it contains a color scan, a 32-bit PixMap may be required.

The alternative is to examine a picture's contents before rendering it. The problem with this approach is that it goes out of date as the PICT format is updated, and it requires a substantial amount of supporting code.

The Picture Utilities are a new set of routines in the System 7.0 versions of Classic and Color QuickDraw. They provide a standard methodology for picture content analyses, returning information about the number and types of graphic objects (including text) contained in a picture. The Picture Utilities also provide color analysis of pictures and PixMaps. These tools provide applications with the capabilities necessary to accurately display graphic data imported from other applications and environments.

For example, an application could obtain a picture's font information, and if some of the fonts weren't available, the application could query the user for a suitable substitute.

While all other QuickDraw routines are in memory at all times, the Picture Utilities are implemented as a package, contained in the System code resource of type 'PACK' and ID #15. The 'PACK' resource type is a code resource accessed by the System via the Package Manager. The Standard File routines also reside in a 'PACK' resource. The Package Manager is described in *Inside Macintosh*, Volume I, and

provides a way to access disk-based System code. This is useful for large routines that are not speed-critical. They are loaded when they are called and are automatically purged if the memory is needed for other processes.

▶ Taking Pictures, in Review

As described in Chapter 2, creating a picture is simple. The **OpenPicture**() trap is called to initiate Picture recording, and **ClosePicture**() is called to end recording. All QuickDraw trap calls that are processed through the bottlenecks or that affect the state of the current port are saved into the picture. The **DrawPicture**() trap is used to display the picture. Pictures can also be saved to disk or exported to other applications, indirectly via the Clipboard or directly using System 7's Edition, High-Level Event, or Process managers.

Note ▶

Picture recording is implemented via the QuickDraw bottleneck procedures. The standard bottlenecks check if a picture is currently being recorded; if it is, then the parameters of the trap call are deposited into the open picture.

Printing also relies on QuickDraw's bottleneck mechanism. In this case, the print drivers determine what to image by replacing the bottlenecks with a printing version. Two QuickDraw traps don't pass through the bottlenecks. As a consequence, they cannot be directly recorded into a picture, nor can they be directly printed. In particular, the **CopyMask**() and **CopyDeepMask**() traps aren't bottlenecked.

But what does "recording into a picture" mean? To answer this question, we first have to look at what we are trying to accomplish with a picture. The goal is to store drawing calls so that they can be recreated later. As earlier chapters have shown, drawing depends on a lot more than just the drawing trap calls invoked: The status of the current port and the destination GDevice also affect drawing. So how does a picture account for and store GrafPort (or CGrafPort) and GDevice information?

While it's true that both the current port and the current GDevice affect drawing, they do so in different ways. The port affects the operation itself: The desired color, location, and pen mode of the drawing are stored in the port. Thus, the state of the port must be

restored after calling **DrawPicture**() to render a picture. QuickDraw accomplishes this by assuming that the port starts in a standard state and then records any changes to the port's state into the picture. Thus, pictures store the tokenized commands necessary to recreate the environment and the trap calls that were made while the picture was being recorded.

Important ▶

GrafPort state information is critical for recreating the drawing environment. GDevice information isn't.

The GDevice controls how drawing will be represented on the destination device. Since the destination is determined when the picture is drawn, the GDevice information when the picture is created isn't pertinent to the picture's data, and consequently, isn't stored in PICTs. If a search procedure is attached to the GDevice (discussed in Chapter 3) to perform a darkening operation when the picture is recorded, the darkening won't occur when the picture is played back unless the target GDevice has the search procedure attached.

Important ▶

A picture is essentially a list of QuickDraw calls and the arguments to those calls. When a picture is played back (in other words, drawn with a call to **DrawPicture**()), the environment that existed when each call was recorded is recreated.

There are actually two different versions of the PICT data format. The PICT version 1 existed in the first version of QuickDraw. It used 1-byte tokens, or *pict opcodes*, to represent each QuickDraw call. Color QuickDraw introduced the PICT version 2 format, which used the 2-byte opcodes described in *Inside Macintosh*, Volume V. Realize that pictures of both versions can be saved in resources of type 'PICT'. Fortunately, QuickDraw can distinguish the two versions from each other and render either one. This is possible because all pictures begin with a 1-byte version opcode that indicates the opcode format of the remaining picture data.

A version 2 picture can be created only if opened in a CGrafPort. Pictures created on color machines in a GrafPort are version 1 pictures.

One of the major advantages of Macintosh is compatibility, making data interchange between Classic and Color QuickDraw critical. Thus, although Classic QuickDraw could only create version 1 pictures, it was updated to include the ability to display version 2 pictures.

Important ▶

At any point in time, current versions of Classic QuickDraw and Color QuickDraw can display each other's PICT versions. There was actually a brief period of time (after Color QuickDraw 1.0 was introduced and before System 7.0 was released) that Classic QuickDraw could not display all Color QuickDraw pictures. Specifically, pictures that contained 16-bit and 32-bit data couldn't be displayed on Classic machines because Color QuickDraw 1.0 wasn't released for Classic QuickDraw machines; the next QuickDraw revision for Classic QuickDraw machines did not come until System 7.0.

On System 6.0.5 for Color QuickDraw machines and System 7.0 for Classic QuickDraw machines, a new picture trap, **OpenCPicture**(), was added. This call is similar to **OpenPicture**(), except it stores resolution information in the picture. Furthermore, it always creates version 2 pictures, even on Classic QuickDraw machines.

▶ Using the Picture Utilities

The Picture Utilities allow applications to profile a number of pictures at one time. This is useful if you are displaying several pictures at the same time. Using the Picture Utilities in this way is similar to using a bank account: The account number data structure is called a PictInfo. As with GWorlds, the PictInfoID structure is considered private; the fields are accessible only through Picture Utility traps. The PictInfo data structure contains the actual data associated with the account; its definition is shown in Listing 8-1.

Listing 8-1. PictInfo data structure

```
typedef struct
{
  short           version;
  long            uniqueColors;
  PaletteHandle   thePalette;
```

```
  CTabHandle          theColorTable;
  Fixed               hRes;
  Fixed               vRes;
  short               depth;
  Rect                sourceRect;
  long                textCount;
  long                lineCount;
  long                rectCount;
  long                rRectCount;
  long                ovalCount;
  long                arcCount;
  long                polyCount;
  long                regionCount;
  long                bitMapCount;
  long                pixMapCount;
  long                commentCount;
  long                uniqueComments;
  CommentSpecHandle   commentHandle;
  long                uniqueFonts;
  FontSpecHandle      fontHandle;
  Handle              fontNamesHandle;
  long                reserved1;
  long                reserved2;
}
PictInfo, *PictInfoPtr, **PictInfoHandle;
```

A Picture account can be opened with **NewPictInfo**(), make deposits with **RecordPictInfo**() or **RecordPixMapInfo**(), request account balance information with **RetrievePictInfo**(), and close the account with **DisposePictInfo**(); unfortunately, there is no way to withdraw money from a Picture account.

NewPictInfo() opens a Picture account. The trap is defined as:

```
OSErr NewPictInfo(
  PictInfoID    *pictInfoID,
  short         verb,
  short         colorsRequested,
  short         colorPickMethod,
  short         version );
```

The "verb" argument specifies the kind of information you are interested in: font, comment, and/or color. The "colorsRequested" argument is the maximum number of colors to be requested with a call to the **RetrievePictInfo**() trap. Some color-sampling methods (described in a following section) use this information to determine

how much memory they need to allocate. In System 7.0, "colorPickMethod" can be one of three predefined types, or a resource ID (in the range [128..32767]) of a custom color-sampling method. The "version" argument should be set to zero for the System 7.0 Picture Utilities.

The **NewPictInfo**() trap returns a new PictInfoID in the location pointed to by the "pictInfoID" argument. This is analogous to a bank account number, and you must use this value anytime you access this account, much like a File System refNum value.

There are two traps that allow information to be added into the account: **RecordPixMapInfo**() and **RecordPictInfo**(). They are defined as:

```
OSErr RecordPixMapInfo(
  PictInfoID    pictInfoID,
  PixMapHandle  thePixMapHdl );

OSErr RecordPictInfo(
  PictInfoID    pictInfoID,
  PicHandle     thePictHdl );
```

The "pictInfoID" argument is the account number returned by **NewPictInfo**() and the PixMapHandle or PicHandle is the reference to the PixMap or picture that is being added. The information is added to a Picture account each time one of these traps is called.

You can request the information about the current account balance using the **RetrievePictInfo**() trap. This trap can be made after each item is added, or once after all recording is finished, depending on your needs. The trap is defined as:

```
OSErr RetrievePictInfo(
  PictInfoID    pictInfoID,
  PictInfo      *thePictInfo,
  short         colorsRequested );
```

The results of this trap are returned in the "PictInfo" argument. The "pictInfoID" argument contains the account number returned by **NewPictInfo**(), as always. The "colorsRequested" argument is the number of colors desired in the ColorTable or palette. This number must be equal to or smaller than the value passed for the "colorsRequested" argument to **NewPictInfo**().

When an application no longer needs a Picture account, it should close it using the **DisposePictInfo**() trap, defined as:

```
OSErr DisposePictInfo(
  PictInfoID    thePictInfoID );
```

This disposes of all the memory used by the Picture account. It doesn't dispose of memory returned by calls to **RetrievePictInfo()**.

There are also two traps that open an account, record information for one picture or PixMap, retrieve the requested values, and then dispose of the account. One trap is for pictures, the other for PixMaps. These are useful if you want to profile only one picture or PixMap. They are defined as follows:

```
OSErr GetPixMapInfo(
  PixMapHandle  thePixMapHdl,
  PictInfo      *thePictInfo,
  short         verb,
  short         colorsRequested,
  short         colorPickMethod,
  short         version );

OSErr GetPictInfo(
  PicHandle     thePictHdl,
  PictInfo      *thePictInfo,
  short         verb,
  short         colorsRequested,
  short         colorPickMethod,
  short         version );
```

From the definitions you can see that the only difference between the traps is that one takes a "PicHandle" argument while the other takes a PixMapHandle argument. The arguments to these traps are the same as the arguments with the same name used by the previously described Picture Utilities traps.

▶ Count of Basic QuickDraw Objects

The Picture Utilities always return information about the different QuickDraw objects contained in a picture. For example, this information might be useful for a Draw-type application. You could determine whether the picture contains anything other than BitMaps and PixMaps and then allocate memory for the objects and extract them individually.

Getting information about the types of objects in a picture using the Picture Utilities is easy. The routine shown in Listing 8-2 gets a PictInfoID from the Picture Utilities using the **NewPictInfo()** trap. Then it loads two pictures from resources and adds them to the Picture account (using **RecordPictInfo()**), retrieving cumulative picture information after loading each Picture (using **RetrievePictInfo()**).

The information is displayed using another routine (not shown here) called PrintPictInfo(). This routine would display the contents of the PictInfo structure in a manner deemed appropriate to the application, perhaps as fields in a simple dialog.

If an error occurs anywhere along the way, the code jumps to **DisposPictInfo()** to clean up the memory allocated by the initial **NewPictInfo()** trap.

Finally, the PictInfoID is disposed of using the **DisposPictInfo()** trap. Note that the pictures themselves are not disposed of since they come from resources and the memory is the property of the Resource Manager. In a commercial application, the pictures would probably come from disk or the Scrapbook, rather than a resource.

Listing 8-2. Determining numbers of picture-based objects

```
/*
** Display information from consecutively numbered
** PICT resources.
*/
OSErr PrintObjectCounts(
  short     pictRsrcID )
{
PicHandle     myPicHandle;
PictInfoID    myPictInfoID;
PictInfo      myPictInfo1, myPictInfoBoth;
OSErr         errStat;

  errStat = NewPictInfo( &myPictInfoID, 0, 0, 0, 0 );
  if (errStat != noErr)
    return errStat;

  /* load Picture from designated resource */
  myPicHandle = GetPicture( pictRsrcID );
  if (myPicHandle == NULL)
  {
    /* failed -> exit */
    errStat = resNotFound;
    DisposPictInfo( myPictInfoID );
    return errStat;
  }

  errStat = RecordPictInfo( myPictInfoID, myPicHandle );
  if (errStat != noErr)
```

```
    {
      /* failed -> exit */
      DisposPictInfo( myPictInfoID );
      return errStat;
    }

    RetrievePictInfo( myPictInfoID, &myPictInfo1, 0 );
    PrintPictInfo( &myPictInfo1, 10, 10 );

    /* Do the next PICT resource */
    myPicHandle = GetPicture( pictRsrcID + 1 );
    errStat = RecordPictInfo( myPictInfoID, myPicHandle );
    if (errStat != noErr)
    {
      /* failed -> exit */
      DisposPictInfo( myPictInfoID );
      return errStat;
    }

    RetrievePictInfo( myPictInfoID, &myPictInfoBoth, 0 );
    PrintPictInfo( &myPictInfoBoth, 100, 10 );

    DisposPictInfo( myPictInfoID );
    return errStat;
}
```

▶ Color Analysis

Probably the most useful feature of the Picture Utilities is the ability to determine the best set of colors for displaying a picture. For example, if you are writing a scanner application that scans in images at 32 bits per pixel but displays them as 8-bit images, you are faced with the problem of determining which 256 colors to use to display the image. Or, if the image spans several screens of varying characteristics, you can use the Picture Utilities to get the best color set for each screen, and then display the result using the **DeviceLoop()** trap (described in Chapter 7).

There are many ways determine the best colors for a given image. The Picture Utilities currently support two methods: **Popular** and **Median Cut**. You can also choose the **System** method, which will itself choose the best method available. Currently, it always selects the Median Cut method, but that may change: If better color selection methods are available in the future, the System method will choose the best one available. If you are undecided about which method to use, use the System method.

Color Plates CP-7A through CP-7D show 32-bit source image rendered using the various Color pick methodologies. CP-7A shows the original source image. CP-7B displays the image with the standard 16-entry ColorTable. CP-7C displays the image with the 16 most popular colors. CP-7D displays the image with the 16 colors generated using the Median Cut method.

All images were drawn without dithering. It's important to understand that the colors returned by the Picture Utilities are generated by statistics described in the following section; this doesn't imply that the colors are guaranteed to make a picture look good on a GDevice with a limited number of entries. For most pictures, the colors returned by the Picture Utilities are a substantially closer match than the standard 16-entry ColorTable.

Popular

The popular method of color selection chooses the most frequently occurring colors. The Picture Utilities accomplish this by producing a histogram of color usage. When profiling a picture containing a single indexed PixMap, the histogram is calculated to the resolution of the PixMap's ColorTable. If the picture contains direct PixMaps or multiple indexed PixMaps, the histogram is calculated to the most significant 5 bits of each RGB component.

When a fixed number of colors is requested, the histogram is searched for the colors most frequently used until the requested number of colors is found. If more colors are requested than occur in the picture, the remaining entries are filled with black.

Median Cut

The median cut method returns a well-distributed set of colors. The Picture Utilities accomplish this by grouping all picture colors into the smallest RGB box that holds them. The length of each side of the box is determined by the distance between the smallest and largest color value in that axis. For example, if the color with the largest red component has a red value of 63488, and the color with the smallest red component has a red value of zero, the length of the red side of the RGB box is 63488 – 0, or 63488.

After the length of each side of the box is determined, the box is split in two along the longest edge. The box is split so that half the colors fall in each of the new boxes. This process is repeated, always splitting along the longest edge of all the RGB boxes, until there are as many boxes as there are colors requested.

Once the desired number of boxes is obtained, the average of all the colors in each box is used to represent that box.

Listing 8-3 shows an example routine that sets the palette of the frontmost window to the number of colors requested, using the selected pick method. Note that the "suppressBlackAndWhite" flag is set since black and white are always available. For a 16-color display, an application would call the AttachPalette() routine with a "numColors" argument value of 14.

Listing 8-3. Creating a palette using Picture Utilities

```
OSErr AttachPalette(
  short         numColors,
  PicHandle     myPicHandle,
  short         method )
{
WindowPtr       curFrontWindow;
PaletteHandle   oldPaletteHdl;
PictInfoID      myPictInfoID;
PictInfo        myPictInfo;
PictInfo        myPictInfoBoth;
Rect            boundsR;
OSErr           errStat;

  /* test to make sure that there is a front window! */
  curFrontWindow = FrontWindow();
  if (curFrontWindow == NULL)
    return (OSErr)(-1);

  /* Palettes already have black and white entries, */
  /* so there is no need to return it */
  errStat =
    NewPictInfo(
      &myPictInfoID, returnPalette, numColors,
      method | suppressBlackAndWhite, 0 );
  if (errStat != noErr)
    return errStat;

  errStat = RecordPictInfo( myPictInfoID, myPicHandle );
  if (errStat != noErr)
  {
    /* failed -> exit */
    DisposPictInfo( myPictInfoID );
    return errStat;
  }
```

```
  RetrievePictInfo( myPictInfoID, &myPictInfo, numColors );

  /* Dispose of old palette */
  oldPaletteHdl = GetPalette( curFrontWindow );
  if (oldPaletteHdl != NULL)
    DisposePalette( oldPaletteHdl );

  /* Attach new palette and get color updates */
  SetPalette( curFrontWindow, myPictInfo.thePalette, TRUE );
  boundsR = (**myPicHandle).picFrame;
  OffsetRect( &boundsR, -boundsR.left, -boundsR.top );
  DrawPicture( myPicHandle, &boundsR );

  DisposPictInfo( myPictInfoID );
  return errStat;
}
```

The palette returned by the Picture Utilities has tolerant entries with a tolerance value of zero. See Chapter 4 and *Inside Macintosh,* Volume VI, for more information about the Palette Manager.

Custom

The two standard color pick methods produce good results for a large class of images. But there are certainly a number of other techniques for color selection. The Picture Utilities allow you to write a custom pick method, much as you can write your own 'WDEF' or 'MDEF'. Custom pick methods are kept in resources of type 'CPMT' and have resource ID numbers in the 128 to 32767 range, inclusive. To select a custom method, simply pass the resource ID of the method you want to use for "method" argument when calling the Picture Utilities.

Writing custom pick methods requires some knowledge of assembly language. The following pick method is written in Think C, which readily allows mixing assembly and C. Modifying this sample method to meet your own needs should be straightforward. This example returns the first *n* unique colors (defined by a globally accessible value; in this case, a global variable "gColorCount") encountered in the picture and is given for illustrative purposes; it is not generally useful. The first thing in a 'CPMT' resource is the main entry point. This routine will be called with a selector in register D0. Listing 8-4 defines the four possible messages.

Listing 8-4. PickColors selector values

```
typedef enum
{
   InitColors          = 0,
   RecordColors        = 1
   CalculateColors     = 2
   KillColors          = 3
}
PickColors_Selectors;
```

The Dispatch routine and the routines associated with the four selector values have the following prototypes:

```
pascal void Dispatch( void );

pascal OSErr InitColors(
   short     colorsRequested,
   long      *dataRef,
   short     *colorBankType );

pascal OSErr RecordColors(
   long             dataRef,
   RGBColor         *colors,
   unsigned long    colorCount,
   long             *uniqueColorsPtr );

pascal OSErr CalculateColors(
   long        dataRef,
   short       colorsRequested,
   short       *colorBankPtr,
   ColorSpec   *resultPtr );

pascal OSErr KillColors(
   long   dataRef );
```

The Dispatch() routine is organized to maximize performance. Since color recording is the most common message the color pick method will receive, this is the first thing checked. The Dispatch() routine is shown in Listing 8-5.

Listing 8-5. Custom color pick dispatch routine

```
/* Dispatch() routine */
pascal void main( void )
{
  asm {
    cmpi.w    #1,D0
    beq       recordColors

    cmpi.w    #2,D0
    beq       calcColors

    move.w    D0,D0
    beq       init

    cmpi.w    #3,D0
    beq       kill
  }
}
```

The first message sent to the color pick method is Init, which is associated with some InitColors() routines. This routine is responsible for allocating memory the color method requires. Since code that resides in a resource does not typically have its own A5 world and thus does not have global variables, the memory reference is passed via the "dataRef" argument. All of the color pick method routines take the dataRef as a parameter. In this example the memory is large enough for "gColorsCount": colors+1. The extra entry is used to keep a count of the number of colors in the RGBColor array.

The second task that the InitColors() routine must perform is to return the type of color bank the pick method will use. There are two default color banks, or you can specify a custom color bank. They are specified by the values in Listing 8-6.

Listing 8-6. ColorBank values

```
typedef enum
{
  ColorBankIsCustom          = -1,
  ColorBankIsExactAnd555     = 0,
  ColorBankIs555             = 1
}
ColorBank_Types;
```

The "ColorBankIs555" value indicates that the Picture Utilities should keep a histogram of colors accurate to 5 bits each of red, green, and blue.

The "ColorBankIsExactAnd555" value indicates that the Picture Utilities should keep an exact color bank, unless either one of the following conditions is met:

- more than one indexed PixMap is being profiled, or
- a direct PixMap is being profiled.

If one of these two cases is true, the color bank is converted to a 5-5-5 histogram and the results are the same as if "ColorBankIs555" were selected.

If either "ColorBankIs555" or "ColorBankIsExactAnd555" is specified, the Picture Utilities will record the colors and the 'CPMT' will not be called with the "recordColors" message.

This sample color pick method simply records the first unique gColorCount colors encountered. Thus, it uses a custom color bank. Listing 8-7 shows an example InitColors() routine.

Listing 8-7. Example InitColors() routine

```
/* application global value */
short    gColorCount;

pascal OSErr Init(
  short  colorsRequested,
  long   *dataRef,
  short  *colorBankType )
{
Handle cSpecHdl;

  *colorBankType = ColorBankIsCustom;
  *dataRef = (long)0;

  cSpecHdl =
    NewHandleClear(
      sizeof(RGBColor) * (gColorCount + 1) );
  errStat = MemError();
  if (errStat != noErr)
    return errStat;

  /* first free index */
  (*(RGBColor**)cSpecHdl)[0].red = 1;
  *dataRef = (long)cSpecHdl;
```

```
  return noErr;
}
```

Since this pick method uses a custom color bank, it needs a RecordColors() routine. This routine takes the following four arguments:

- data reference (which was determined by the Init routine),
- a pointer to an array of RGBColors,
- a count of colors in the RGBColor array, and
- a pointer to a long integer variable that will be used to hold the number of unique colors.

Since the color banks used by the Picture Utilities are only accurate to the most significant 5 bits each of RGB component, unique colors are distinguished only to that resolution. With a custom color bank, you can use whatever method you like to count the unique colors in the picture or PixMap.

The RecordColors() routine is called for every pixel in the picture. To minimize the effects of execution overhead, the Picture Utilities passes an array of RGBColors to the RecordColors() routine. The number of colors in the array is indicated by the value of the "colorCount" argument.

Listing 8-8 shows an example of a RecordColors() routine. It compares each of the colors with the colors in the array to determine if it's unique. The first "colorCount" unique colors are put in the myColors array, which is pointed to by the "dataRef" argument.

Listing 8-8. Example RecordColors() routine

```
pascal OSErr RecordColors(
  long            dataRef,
  RGBColor        *colors,
  unsigned long   colorCount,
  long            *uniqueColorsPtr )
{
RGBColor          *myColors;
RGBColor          anRGB;
unsigned long     arrayIndex;
short             index, i;
Boolean           gotUniqueColor;

  myColors = *(RGBColor**)dataRef;
  index = myColors[0].red;
```

```
    arrayIndex = 0;
    while (index <= COLOR_COUNT && arrayIndex < colorCount)
    {
      gotUniqueColor = FALSE;

      while (arrayIndex < colorCount)
      {
        gotUniqueColor = TRUE;
        anRGB = colors[arrayIndex];
        for (i=1; (i<=index) && gotUniqueColor; i++)
        {
          if ((anRGB.red == myColors[i].red) &&
               (anRGB.green == myColors[i].green) &&
               (anRGB.blue == myColors[i].blue))
            gotUniqueColor = FALSE;
        }
        if (gotUniqueColor)
          break;
        else
          arrayIndex++;
      }

      if (gotUniqueColor)
      {
        myColors[index] = colors[arrayIndex];
        index++;
        (*uniqueColorsPtr) += 1;
      }
      else
        break;
    }

    myColors[0].red = index;

    return noErr;
}
```

The Picture Utilities will call the CalcColors() routine to get the best colors. The "dataRef" argument is the same as that returned by the Init() routine; the "colorsRequested" argument specifies the number of colors to return; the "colorBankPtr" argument is a pointer to the color bank if the Picture Utilities kept the color bank; the "resultPtr" argument contains the address of a 16-bit (short) integer variable where the 'CPMT' should put the result.

Listing 8-9 shows an example implementation of a custom histogram calculation method. The CalculateColors() routine returns the unique colors stored in the color bank by the RecordColors() routine. If less than the number of colors requested are stored in the color bank, the extra entries are cleared to zero.

Listing 8-9. Example CalculateColors() routine

```
pascal OSErr CalculateColors(
  long                dataRef,
  short               colorsRequested,
  register short      *colorBankPtr,
  register ColorSpec  *resultPtr )
{
register RGBColor  **colors;
register short     i;
short              maxValue;
short              colorsStored;

  maxValue = colorsRequested;
  colors = (RGBColor**)dataRef;
  colorsStored = (*colors)[0].red - 1;

  if (colorsRequested > colorsStored)
    maxValue = colorsStored;

  for (i=0; i<maxValue; i++)
  {
    resultPtr[i].rgb = (*colors)[i + 1];
    resultPtr[i].value = i;
  }

  /* fill in the great unwashed masses */
  for (i=maxValue; i<colorsRequested; i++)
  {
    resultPtr[i].rgb.red =
    resultPtr[i].rgb.green =
    resultPtr[i].rgb.blue = 0;
    resultPtr[i].value = i;
  }

  return noErr;
}
```

The KillColors() routine is called when the Picture Utilities are done with the color pick method when the application calls **DisposPictInfo**(). It's responsible for disposing of any memory that the color pick method allocated. Listing 8-10 shows an example of a KillColors() routine.

Listing 8-10. Example KillColors() routine

```
pascal OSErr KillColors(
  long dataRef )
{
OSErr    errStat;

  errStat = noErr;
  if (dataRef != NULL)
    DisposHandle( (Handle)dataRef );

  return errStat;
}
```

Note ▶

The Picture Utilities keep a color bank accurate to 5 bits each of RGB. The InverseTable on a standard indexed GDevice has a resolution of only 4 bits. This means that seven-eighths of the colors returned by the Picture Utilities might be beyond the resolution of the GDevice's InverseTable. You may need to change the InverseTable resolution to 5 bits for optimal results, using the **MakeITable**() trap.

▶ Using Deep PixMaps on Classic QuickDraw Machines

A major deficiency of Classic QuickDraw is the ability to manipulate deep PixMaps. Anyone who has attempted to develop an application that runs on both Classic QuickDraw and Color QuickDraw machines has run into this problem. While the ultimate solution (implementing Color QuickDraw on 68000 machines) has not yet taken place, this section presents a simple idea that can assist with displaying PixMaps on Classic QuickDraw machines.

The idea is to take advantage of the fact that Classic QuickDraw can display version 2 pictures. Thus, to display a PixMap on a Classic QuickDraw machine, you simply need to create a picture that contains the PixMap and then call **DrawPicture**(). The routine in Listing 8-11 creates a version 2 picture containing a given PixMap.

Listing 8-11. CreatePICT2() routine

```
#define CLIPSIZE            12
#define PIXMAPRECSIZE       50
#define HEADERSIZE          40
#define MAXCOLORTABLESIZE   256*8+8   /* size of 256 entry */
                                      /* ColorTable */
#define OPCODEMISCSIZE      2+8+8+2   /* opcode+srcRect+ */
                                      /* dstRect+mode */
#define ENDOFPICTSIZE       2
#define PICSIZE \
         PIXMAPRECSIZE + HEADERSIZE + MAXCOLORTABLESIZE + \
         ENDOFPICTSIZE + OPCODEMISCSIZE + CLIPSIZE

PicHandle CreatePICT2(
  PixMap    *srcBits,
  Rect      *srcRect,
  Rect      *dstRect,
  short     mode )
{
PicHandle   thePictHdl;
long        curPictSize;
short       *picPtr, *ctPtr, *pixMapPtr;
short       myRowBytes, picPosition, iii;

  /* determine rowbytes value with flag bits stripped */
  myRowBytes = srcBits->rowBytes & 0x3fff;

  /* allocate worst case memory scenario */
  curPictSize =
    PICSIZE + ((myRowBytes / 127) + 2 + myRowBytes)
    * (srcBits->bounds.bottom - srcBits->bounds.top);
  thePictHdl = (PicHandle)NewHandle( curPictSize );
  if (thePictHdl == NULL)
    return NULL;

  /* skip picSize and put out picFrame (10 bytes) */
  picPtr = (short*)(((char*)*thePictHdl) + 2);
  *picPtr++ = srcRect->top;
  *picPtr++ = srcRect->left;
  *picPtr++ = srcRect->bottom;
  *picPtr++ = srcRect->right;

  /* put out header (30 bytes), this could be done from a */
  /* resource, or taken from an existing picture */
  *picPtr++ = 0x0011;  /* version opcode */
```

```
*picPtr++ = 0x02ff;  /* version number */
*picPtr++ = 0x0C00;  /* header opcode */

/* the rest of the header is ignored, 0 it out */
for (iii=12; iii>0; iii--)
  *picPtr++ = 0;   /* write out 24 bytes of 0 */

/* put out a clip region: containing a bounds rectangle */
/* and zero bytes of scan line data */
*picPtr++ = 0x0001;
*picPtr++ = 0x000A;
*picPtr++ = srcBits->bounds.top;
*picPtr++ = srcBits->bounds.left;
*picPtr++ = srcBits->bounds.bottom;
*picPtr++ = srcBits->bounds.right;

/* put out "packBitsRect" opcode: $0098 */
*picPtr++ = 0x0098;

/* put out PixMap */
*picPtr++ = srcBits->rowBytes | 0x8000; /* always make */
                                        /* PixMaps */
*picPtr++ = srcBits->bounds.top;        /* emit bounds rect */
*picPtr++ = srcBits->bounds.left;
*picPtr++ = srcBits->bounds.bottom;
*picPtr++ = srcBits->bounds.right;
*picPtr++ = 0;                          /* version number */

/* put out PixMap's data pack information */
if (myRowBytes < 8)
  *picPtr++ = 1;     /* unpacked format */
else
  *picPtr++ = 0;     /* standard format */

*picPtr++ = 0;       /* packed size */
*picPtr++ = 0;
*picPtr++ = 0x0048;  /* horizontal resolution: $0048 0000 */
*picPtr++ = 0;
*picPtr++ = 0x0048;  /* vertical resolution: $0048 0000 */
*picPtr++ = 0;
*picPtr++ = 0;       /* pixel type: chunky */

/* these fields are different for BitMap/PixMap */
if ((srcBits->rowBytes & 0x8000) != 0)
{
  /* do PixMap */
```

```
    *picPtr++ = srcBits->pixelSize;  /* pixel size */
    *picPtr++ = srcBits->cmpCount;   /* number of components */
    *picPtr++ = srcBits->cmpSize;    /* size of each component */
  }
  else
  {
    /* do BitMap */
    *picPtr++ = 1;                   /* pixel size */
    *picPtr++ = 1;                   /* number of components */
    *picPtr++ = 1;                   /* size of each component */
  }

  /* the remainder are the same */
  *picPtr++ = 0;                     /* offset to next plane */
  *picPtr++ = 0;
  *picPtr++ = 0;                     /* ColorTable */
  *picPtr++ = 0;
  *picPtr++ = 0;                     /* reserved */
  *picPtr++ = 0;

  /* emit ColorTable */
  if ((srcBits->rowBytes < 0) && (srcBits->pmTable != NULL))
  {
    /* obtain pointer to ColorTable */
    ctPtr = (short*)*(srcBits->pmTable);

    *picPtr++ = *ctPtr++;            /* copy ctSeed */
    *picPtr++ = *ctPtr++;
    *picPtr++ = *ctPtr++;            /* copy ctFlags */

    iii = *ctPtr;
    *picPtr++ = *ctPtr++;            /* copy ctSize */
    for ( ; iii >= 0; iii--)
    {
      /* put out all entries */
      *picPtr++ = *ctPtr++;          /* pixel value */
      *picPtr++ = *ctPtr++;          /* red */
      *picPtr++ = *ctPtr++;          /* green */
      *picPtr++ = *ctPtr++;          /* blue */
    }
  }
  else
    /* put out an empty ColorTable: 8 words of 0 */
    for (iii=8; iii>0; iii--)
      *picPtr++ = 0;
```

```
/* put out srcrect, dstrect, and mode */
*picPtr++ = srcRect->top;
*picPtr++ = srcRect->left;
*picPtr++ = srcRect->bottom;
*picPtr++ = srcRect->right;

*picPtr++ = dstRect->top;
*picPtr++ = dstRect->left;
*picPtr++ = dstRect->bottom;
*picPtr++ = dstRect->right;

*picPtr++ = mode;

/* put out PixData */
if (myRowBytes < 8)
{
  /* no packing */
  pixMapPtr = (short*)(srcBits->baseAddr);
  iii =
    (myRowBytes *
    (srcBits->bounds.bottom - srcBits->bounds.top)) / 2;
  for ( ; iii > 0; iii--)
    *picPtr++ = *pixMapPtr++;
}
else
{
/* use packbits to compress the data */
Ptr    srcPtr, dstPtr, packBuf, tempPicPtr;
short  packedSize, jjj;

  tempPicPtr = (char*)picPtr;
  srcPtr = srcBits->baseAddr;
  HLock( (Handle)thePictHdl );
  packBuf = NewPtr( myRowBytes * 2 );
  if (packBuf == NULL)
  {
    DisposHandle( thePictHdl );
    return NULL;
  }

  /* Must use only byte accesses to avoid */
  /* address errors on machines with 68000 CPUs */
  iii = srcBits->bounds.bottom - srcBits->bounds.top;
  for ( ; iii > 0; iii--)
  {
    dstPtr = packBuf;
```

```
        PackMoreBits( &srcPtr, &dstPtr, myRowBytes );
        packedSize = (long)dstPtr - (long)packBuf;
        if (packedSize < 250)
          /* put a byte to the picture */
          *tempPicPtr++ = (unsigned char)packedSize;
        else
        {
          /* put a word to the picture */
          *tempPicPtr++ = (unsigned char)(packedSize >> 8);
          *tempPicPtr++ = (unsigned char)packedSize;
        }

        /* put the packed data out */
        dstPtr = packBuf;
        for (jjj = packedSize; jjj > 0; jjj--)
          *tempPicPtr++ = *dstPtr++;
      }
      DisposPtr( packBuf );
      HUnlock( (Handle)thePictHdl );

      /* perform long alignment */
      if ((long)tempPicPtr & (long)0x0001)
        tempPicPtr++;
      picPtr = (short *)tempPicPtr;
    }

    /* All done! Put out "end of picture" opcode: $00FF */
    *picPtr++ = 0x00FF;

    /* resize handle down to the amount actually used */
    curPictSize = (long)picPtr - (long)*thePictHdl;
    SetHandleSize( thePictHdl, curPictSize );
    *((short*)*thePictHdl) = (short)curPictSize;

    /* mission accomplished: return the pict to the caller */
    return thePictHdl;
}


void PackMoreBits(
    Ptr     *srcPtr,
    Ptr     *dstPtr,
    short   myRowBytes )
{
short     smallRowBytes, iii;
```

```
    /* first do the odd (remainder) amount */
    smallRowBytes = myRowBytes % 127;
    for (iii = myRowBytes/127; iii >= 0; iii--)
    {
      PackBits( srcPtr, dstPtr, smallRowBytes );
      smallRowBytes = 127;
    }
}
```

This routine has a calling interface similar to **StdBits**() except that it doesn't take a region parameter and it returns a PicHandle. If your application needs to use regions, you can modify the code to use the "PickBitsRgn" opcode $99 and to put the region into the picture just before the pixel data.

Another limitation is that this routine only accepts BitMaps and PixMaps of depths 8-bits or less. For direct PixMaps you need to modify the routine to use the "DirectPackBitsRect" opcode $009A and change the packing as described in *Inside Macintosh*, Volume VI. If you decide to do this, be sure to assign the *baseAddr* field — it's not used for opcodes $0098 and $0099 — with value $000000FF. See *Inside Macintosh*, Volume VI. Creating direct PixMaps to pass to the CreatePICT2() routine is much easier than creating indexed PixMaps since you don't need to generate a ColorTable. But Classic QuickDraw machines before System 7.0 can't display pictures containing direct PixMaps. And that is why this approach was chosen here.

The first section of this routine makes a worst-case memory guess. It assumes a worst-case packing scenario that can slightly increase the picture size. This memory management strategy is used because it's much faster to allocate a large Handle to begin with than to attempt to continually adjust the size of a Handle that is too small. Furthermore, it greatly simplifies the code that adds data to the picture since it doesn't constantly have to check if there's enough space. A possible negative side effect is that there might not be enough memory for the requested block, although there is enough memory for the picture. You could rewrite the code to make a different tradeoff if you wish. Another improvement would be writing the PICT directly to disk. You could then spool the picture from disk when playing it back as described in *Inside Macintosh*, Volume V. While this is slightly slower due to disk access time, it uses only a minimal amount of memory.

The routine puts out the picture's *picFrame* next. The picPtr variable is used to put data into the picture. The Picture's *picSize* field, which is the first word of the picture data, is skipped since the final size of the picture is not known yet. It will be the last thing written to the picture.

In this case, the *picFrame* field is simply the srcRect passed to the routine. When pictures are created via QuickDraw, the *picFrame* is passed as an argument to **OpenPicture**(). Next the routine puts out a standard PICT2 header as described in the Color QuickDraw chapter of *Inside Macintosh*, Volume V, followed by a clipping region the size of the bounds rectangle. Setting a clipping region avoids a bug in Classic QuickDraw that prevents pictures from drawing in certain circumstances. Specifically, when a wide-open region is scaled or translated, the region operations overflow and the resulting region is empty.

The PackBitsRect opcode, $0098, follows the clipping region. The opcode is followed by the PixMap record. Note that the *baseAddr* field is not stored in the picture for opcodes $0098 and $0099. For opcode $009A and $009B (used for direct PixMaps) you must set the PixMap's *baseAddr* field to $000000FF. This allows machines that are not aware of the $009A and $009B opcodes to terminate picture playback, rather than crash. These opcodes are further described in *Inside Macintosh*, Volume VI. The CreatePICT2() routine can take either a BitMap or a PixMap as the "srcBits" argument. If a BitMap is passed, it's added to the picture data as though it were a 1-bit/pixel PixMap.

After the PixMap record and ColorTable are put to the picture, the srcRect, dstRect, and transfer mode are put out. Like the PixMap, these arguments are passed to CreatePICT2().

The next section of code puts out the actual PixMap data and Handles packing. QuickDraw uses the **PackBits**() trap to pack each scanline of pixel data when rowBytes is greater than eight. If the rowBytes value is less than eight, the data is put into the picture without packing. On Classic QuickDraw machines, the **PackBits**() trap can only pack a maximum of 127 bytes at a time. The PackMoreBits() routine takes care of larger packing runs by calling the standard **PackBits**() trap multiple times. Packing in groups a maximum of 127 bytes in length is slightly less efficient than allowing arbitrarily sized groups. The initial worst-case memory scenario takes this into account. Finally an EndOfPicture opcode, $00FF, is put to the picture. The size of the picture is calculated, and the Handle is resized; the Handle is always made smaller since the original allocation assumed worst case.

Once the picture is created, it can be treated like any other picture: It can be displayed using **DrawPicture**(), or saved to disk and displayed in full color on a Color QuickDraw machine.

Generating the PixMap to pass to CreatePICT2() can be a little tricky on Classic QuickDraw machines. The easiest way to do this is by putting a generic PixMap in a resource and then modifying relevant

fields to meet your needs. The ColorTable can be handled in the same fashion. Alternatively, the PixMap and ColorTable can be constructed manually if desired.

One problem you will encounter is mapping colors to indices. Recall that QuickDraw does this by using an inverse lookup table. Since none of the inverse table functionality exists on Classic QuickDraw machines, you're on your own. You may simply want to do an exhaustive search of the ColorTable for the closest match for each pixel. Slow, but simple and functional.

If you are running on a System 7.0 version of Classic QuickDraw you can use the Picture Utilities to help with the ColorTable construction. The previous section in this chapter as well as *Inside Macintosh*, Volume VI, describes how to do this.

Or, if you are running System 7.0 you can create pictures with direct PixMaps. Direct PixMaps are much easier to create since you don't have to map colors back into a ColorTable.

This routine also provides a solid starting point for implementing a version of **CopyBits**() which accepts any depth PixMap as the source for Classic QuickDraw machines. You could patch **StdBits**() and look at the depth of the source PixMap. If it is 1 bit, then the **StdBits**() trap is called. If it's deeper than 1 bit, create a picture as just discussed, draw the picture, and then throw the picture away.

▶ Summary

This chapter described many of the inner workings of pictures and how to access the information contained within them, using the Picture Utilities package new to System 7.0. The Picture Utilities provide an easy way to profile the contents of a picture. This makes it easier for an application to intelligently deal with the contents of a picture. Perhaps the most useful aspect of the Picture Utilities is that they will return the best color set for displaying the picture. For example, if you want to display a 32-bit PixMap on a 16-bit screen, the Picture Utilities can give you the best palette to do so. There are two standard color pick methods. If neither is suitable for your application, you can write your own.

Finally, this chapter presented code for creating a picture containing a deep PixMap. This allows an application to display PixMaps on Classic QuickDraw machines using **DrawPicture**().

9 ▶ Compatibility and Configuration Issues

One of the greatest virtues of commercial microcomputer architectures is openness, that is, the ability to add or change components freely in order to suit the user and/or owner. Unfortunately, this virtue is arguably the biggest headache for microcomputer applications developers. A launched application is thrust into an uncertain environment whose features and services vary from machine to machine and may have even been changed since the last time the application was launched. In order to meet this challenge, applications must test for all operating system services that aren't guaranteed to be present on launch, and they must take appropriate action should those services be absent or defective. As a developer, if you ignore Memory Manager errors or if you access QuickDraw traps that are not present on the current machine, a system error is the usual result. Failure to follow such programming essentials should be considered unacceptable application behavior under virtually all circumstances.

Striving to meet the goal of wide-range compatibility can involve a significant amount of additional special-case coding and careful structuring. You should consider both your programming resources and market forces when making application compatibility decisions. What percentage of the targeted users are running the required System software configuration? How much development time is available? What are the best case and worst case scenarios?

▶ Hardware Configuration

Apple has sold upwards of ten different Macintosh CPU "boxes" under the Macintosh name. When cross-combined with only the Apple hardware options (hard disk drives, video cards, display monitors, and so forth), the real number of different Macintosh hardware configurations is huge.

It's a tribute to Apple System Software that the details of a particular hardware configuration are almost completely transparent to the application. For example, there are many different kinds of disk drives that can be connected to the Macintosh — from 400K floppy disks to magneto-optical disks with over 1 gigabyte of storage — but the operating system manages the differences between these devices and applications need deal only with a standard interface provided by (in this case) the File Manager. The same is true for video cards and monitors. Macintosh displays range in size from 512-by-342 monochrome to 1152-by-870 (or even larger) in 32-bit color. Fortunately QuickDraw handles all of the device dependent aspects of drawing and presents a standard graphics environment to the application.

It's important for applications to follow the well-behaved application rules described in the following section to reap the benefits of device independence as Apple expands its System Software. If your application is dependent on a new piece of hardware (such as a frame grabber) for which there is no standard interface in the Toolbox, you will need to write custom routines. This is what developers did in the interim between the advent of 32-bit display cards and direct pixel support in QuickDraw.

This scenario appears again and again. As of this writing, there is no support in the Toolbox to support video overlay or frame grabbing, yet these cards are becoming increasingly popular. To support such special video modes, developers are currently forced to write their own graphics routines. It's only a matter of time before Apple provides guidelines for producing such hardware and a standard software interface for dealing with the new capabilities.

Once these standard guidelines are published, it is important that your application abides by them. Any rules your application breaks increases the probability that it will not be compatible with a future version of System Software.

▶ The Well-Behaved Application

With respect to graphics, Apple has developed a set of behavioral guidelines that describe the well-behaved application. The rules for well-behaved applications aren't really that restrictive, and the benefits include not merely future compatibility, but also improved performance on systems with graphics acceleration hardware that conforms to these guidelines.

The performance of QuickDraw graphics has enormous potential for performance enhancement; a well-behaved application may be able to realize a more than 100-fold improvement in its graphics-intensive code over that of unaccelerated performance. The fundamental criterion is that performance improvements must be accessible to applications within the Macintosh graphics environment, and that they require no additional programming beyond the techniques prescribed by the well-behaved application guidelines described in this section.

There are two strategies that can be undertaken by a hardware manufacturer to speed up Macintosh system graphics:

- Achieve faster execution cycle time by incorporating faster dedicated graphics processing units.
- Minimize NuBus traffic via cached data structures. When graphics data are transferred from application RAM to the video memory resident on display cards, they travel across the NuBus. These transfers are significantly slower than those in main memory. By keeping up-to-date copies of large graphics data structures on the video card itself, accelerators can realize significant performance improvements.

From a software developer's perspective, hardware accelerators shouldn't be a concern: Either the user's machine has the hardware or it doesn't. Graphics hardware accelerators function in a transparent, device-independent fashion if you follow the well-behaved application rules.

Accelerators achieve performance improvements to all applications (for example, faster line and polygon drawing), by patching QuickDraw traps when the system boots up. Additional graphics capabilities that aren't standard in QuickDraw (for example, video frame capture or specialized image processing operations) are implemented by defining new device driver calls. Of course, these features are accessible to only those applications that "understand" them and have tested for the presence of that specific hardware.

The primary focus of Apple's "good behavior" guidelines is to provide developers with rules that guarantee compatibility with accelerators and future versions of System Software. To allow accelerators to maintain accurate, effective graphics data caching, applications must carefully limit the direct modification of the following important Graphics Toolbox data structures:

- GDevice
- GrafPort and CGrafPort
- ColorTable
- PixPat

If an application needs to modify one of these data structures, it can inform the Graphics Toolbox via one of the appropriate traps listed here:

- **GDeviceChanged()**
- **PortChanged()**
- **CTabChanged()**
- **PixPatChanged()**

Caching these data structures allows the Graphics Toolbox to maintain them on the display card itself, rather than having to copy them across the NuBus for every graphics operation. Calling one of these new "DataStructureChanged" traps directs the Graphics Toolbox to reestablish consistency within the data structure and to update any cache that the data structure may be residing in.

Apple has decreed direct data structure modifications, other than these four data structures, as threatening to future compatibility. Fortunately, most cases where an application might consider direct data structure modification can be addressed with a trap call. On two different occasions, Apple has published a list of traps that should be called in such circumstances, shown here in Listing 9-1. The first section of code is from System 7.0 ßeta2 CD and *develop* magazine; the second is from System 7.0 ßeta2 CD; and the third is from *develop* magazine.

Listing 9-1. Well-behaved application traps

```
AddComp()           AddSearch()          BackColor()
BackPat()           BackPixPat()         ClipRect()
ColorBit()          CopyPixPat()         DelComp()
DelSearch()         ForeColor()          HidePen()
HiliteColor()       MakeRGBPat()         MovePortTo()
OpColor()           PenMode()            PenNormal()
PenPat()            PenPixPat()          PenSize()
PortSize()          RGBBackColor()       RGBForeColor()
SetClip()           SetOrigin()          SetPenState()
SetPortBits()       SetPortPix()         ShowPen()

CharExtra()         GrafDevice()         HideCursor()
Move()              MoveTo()             ObscureCursor()
SetCCursor()        SetClientID()        SetDeviceAttribute()
SetCursor()         SetGDevice()         SetPort()
ShowCursor()        SpaceExtra()         TextFace()
TextFont()          TextMode()           TextSize()

PMgrDispatch        GWorldDispatch
InitGDevice()       DisposGDevice()
DisposPixPat()      CopyBits()           CopyPixMap()
DrawPicture()       OpenPicture()        OpenPoly()
StdLine()           StdBits()            StdText()
StdRgn()            StdArc()             StdRRect()
StdOval()           StdPoly()            StdRect()
OpenRgn()           CopyRgn()            DiffRgn()
XorRgn()            RectRgn()            SectRgn()
UnionRgn()          OffsetRgn()          InsetRgn()
```

If we examine these traps in terms of the data types that they operate on, the list of traps transforms into a prescription for dealing with Macintosh data structures. The data structures and the prescription are as follows:

- **GrafPorts, CGrafPorts, PixPats, ColorTables** — Protected, but accessible. Use the appropriate "Changed" trap after direct modifications are performed.
- **GWorlds, Regions, Pictures, Palettes, Polygons** — Absolutely protected. Don't change them in any way, except via the traps provided.

Important ▶

These lists and guidelines also imply that applications which rely on custom bottlenecks that don't call through to the standard procs installed in onscreen (C)GrafPorts or Color GrafPorts cannot be accelerated.

We must also realize that there is a physical limit to the number of data structures that can be cached at one time. Developers should assume a "most recently used" caching strategy. The best way to account for this in application code is to group drawing operations so that operations within each GrafPort and GDevice are performed together. Examples of good and bad technique are shown in Listing 9-2.

Listing 9-2. Efficient and inefficient graphics code

```
/* global variables */
/* previously assigned */
PicHandle       gBackgroundPict;
Rect            gBasicR;

void Doit_Good(
  GWorldPtr     gwArray[],
  short         gwCount )
{
GDHandle        savedGDev;
GrafPtr         savedPort;
Rect            pictBoundsR, currentR;
short           i, j, k;

  GetGWorld( &savedGDev, &savedPort );

  pictBoundsR = (**gBackgroundPict).picFrame;
  OffsetRect( &pictBoundsR, -pictBoundsR.left, -pictBoundsR.top );
  for (i=0; i<gwCount; i++)
  {
    SetGWorld( gwArray[i], NULL );

    /* draw background picture */
    DrawPicture( gBackgroundPict, &pictBoundsR );

    /* draw some lines */
    for (j=0; j<10; j++)
    {
```

```
        MoveTo( j*10 + 30, j*10 + 30 );
        Line( 20, 10 );
      }

      /* fill some rectangles */
      currentR = gBasicR;
      for (k=0; k<5; j++)
      {
        PaintRect( &currentR );
        OffsetRect( &currentR, 10, 10 );
      }
    }

    SetGWorld( savedGDev, savedPort );
}

void Doit_Bad( void )
    GWorldPtr   gwArray[],
    short       gwCount )
{
GDHandle    savedGDev;
GrafPtr     savedPort;
Rect        pictBoundsR, currentR;
short       i, j, k;

    GetGWorld( &savedGDev, &savedPort );

    pictBoundsR = (**gBackgroundPict).picFrame;
    OffsetRect( &pictBoundsR, -pictBoundsR.left, -pictBoundsR.top );

    for (i=0; i<gwCount; i++)
    {
      SetGWorld( gwArray[i], NULL );

      /* draw background picture */
      DrawPicture( gBackgroundPict, &pictBoundsR );
    }

    for (i=0; i<gwCount; i++)
    {
      SetGWorld( gwArray[i], NULL );

      /* draw some lines */
      for (j=0; j<10; j++)
      {
```

```
            MoveTo( j*10 + 30, j*10 + 30 );
            Line( 20, 10 );
        }
    }

    for (i=0; i<gwCount; i++)
    {
        SetGWorld( gwArray[i], NULL );

        /* fill some rectangles */
        currentR = gBasicR;
        for (k=0; k<5; j++)
        {
            PaintRect( &currentR );
            OffsetRect( &currentR, 10, 10 );
        }
    }

    SetGWorld( savedGDev, savedPort );
}
```

To summarize, graphics-intensive applications should be able to take maximum advantage of the available graphics hardware. This advantage can be realized only when applications, System Software, and graphics hardware cooperate with one another. Within the context of this cooperation, applications are responsible for the maintenance of graphics data structures. They shouldn't perform direct modifications to them, particularly when those modifications can be performed by a trap. If such modifications are absolutely necessary, then the application must inform the Graphics Toolbox. For the highest level of Toolbox software and graphics hardware support and cooperation, applications should group graphics operations by destination and functionality.

Note ▶

There's always great demand for "faster" graphics; indeed, most graphics-intensive applications and their users are the most direct benefactors of advancing graphics technologies. It's important for software developers to remember, however, that the best opportunities to improve graphics performance lie within the graphics applications themselves. Minimizing pixel value recalculations and onscreen drawing can improve the performance and appearance of a clunky application to a far greater degree than a $3000 graphics hardware hotrod kit!

▶ Software Compatibility

Apple has published guidelines on how to write applications that will stay compatible with future System software and Macintosh family hardware. These guidelines are essentially an open agreement between Apple and its developers stating what actions developers and Apple must undertake to maintain future compatibility. The guidelines aren't published collectively, but are available separately in a number of *Inside Macintosh* chapters and Tech Notes. Many of the guidelines pertinent to graphics programming don't specifically focus on graphic issues; as a consequence, many graphics developers aren't familiar with the guidelines and the rules that they prescribe.

In the course of graphic software component design, software developers should imagine the likely configuration of graphics hardware that their user's systems will have. They should also be concerned with the minimum configuration, not to mention illegal configurations. Accounting for ideal system configurations is perhaps a luxury, but it is important to consider nonetheless.

This variety of display setups usually causes developers to react in a number of ways. They create software that can run on:

- **One system configuration only, but the application crashes or behaves weirdly on most other systems** — This scenario often results when lone developers throw up their hands and say, "I give up. So it'll only run on my system!"
- **One configuration only, but it exits from other systems on initialization with a nice "Bye-bye, no can do!" alert** — This scenario is essentially Case #1, except that the programmer has learned how to use **SysEnvirons**() or **Gestalt**() to check for nominal System Software and hardware features.
- **Many different systems, but the application causes other applications to visually degrade and behave strangely** — The programmers have some nifty tricks up their sleeves, but their applications prefer to take direct control of display GDevices.
- **Every known configuration and a few impossible ones (such as a 512K machine with a 68882 math chip) to boot** — The software team implemented graphics-intensive routines to support 1-, 2-, 4-, 8-, 16-, and 32-bit depths and they developed color-matching strategies to make 32-bit images appear almost realistic on monochrome machines. The drawback: The application took forever to write, ran as slow as molasses, and took up over 2000K on disk.

The point here is that most applications need to meet some basic goals. They need to:

- **Look really good on an ideal system** — A 24-bit scanning application looks best on a high-resolution display driven by a 24-bit video controller.
- **Look pretty good on most systems** — If graphics aren't a major part of a program, then it should be easy to use QuickDraw techniques that require a minimum of colors in the most flexible way.
- **Exit gracefully on some systems** — Maybe a floral–arrangement design application doesn't need to work on a Macintosh Plus or a Classic...

Developers strive for the most direct means to meet these goals, while at the same time avoid implementing parallel versions of individual graphics algorithms.

▶ Detecting and Handling Limited System Software

One of the difficulties involved with Macintosh applications development is handling the wide range of System software configurations that an application can encounter. It's desirable to develop applications that can run under the largest number of reasonable System software configurations. We say "reasonable" because there are people out there who are still running System 3.3 and Finder 4.2, and there's no way that a color application is going to operate under that configuration short of implementing its own version of Color QuickDraw. There is also a large number of systems out there on which users have loaded numerous INITs and perhaps even some viruses that may prevent applications from behaving properly or in a manner that makes the danger undetectable to applications.

No Color or 32-bit QuickDraw

A lack of Color QuickDraw is typically intolerable for graphics-intensive applications. This means that such applications won't run on any machine in the current Macintosh Classic family. You should consider whether your application is truly graphics-intensive before you decide to turn your back on this large installed base. If it can make do with eight colors, you may consider selecting colors using the Classic color model discussed in Chapter 3. An example of such testing is shown in Listing 9-3.

Listing 9-3. Drawing under black-and-white and Color QuickDraw

```
#define PORT_IS_COLOR(A) \
   ((((CGrafPtr)(A))->portVersion & 0xC000) == 0xC000)

void SafeDrawingRoutine( void )
{
RGBColor    curPenRGB;
Rect        boxR;
Boolean     isColorPort;

   /* determine whether the current port is */
   /* a CGrafPort or GrafPort */
   isColorPort = PORT_IS_COLOR( thePort );

   if (isColorPort)
   {
      /* set a cyan pen color */
      curPenRGB.red = 0x0000;
      curPenRGB.green = 0xFFFF;
      curPenRGB.red = 0xFFFF;
      RGBForeColor( &curPenRGB );
   }
   else
   {
      /* set a cyan pen color, use a pen pattern to */
      /* distinguish it further from solid black and white */
      ForeColor( cyanColor );
      PenPat( ltGray );
   }

   /* draw using traps available to all QuickDraw versions */
   SetRect( &boxR, 0, 0, 40, 40 );
   OffsetRect( &boxR, 80, 30 );
   FillRect( &boxR );
}
```

Notice the use of the pen pattern in the black-and-white condition in this code fragment. You can often achieve adequate results when the display's pixel depth is less than ideal.

Using 32-bit QuickDraw is another matter; its reliance on 68020 code means that Apple isn't likely to develop a version that runs on 68000 CPUs. The tendency is for applications that are graphics-intensive to require 32-bit QuickDraw in order to run. If the application requires the Direct color model, then it requires 32-bit QuickDraw and must test for it on startup.

Old System Software

This is a more general version of the 32-bit QuickDraw present condition, except that in addition to QuickDraw, the developer has to consider whether the functionality of the trap patches found in a particular system service are adequate. Some of the more important features that have changed over minor System Software releases include the following:

- Color and grayscale printing require LaserWriter 6.0, which in turn requires System 6.0.3.
- The Palette Manager has had a number of significant enhancements up through version 6.0.5. In particular, the Palette Manager from Systems 6.0.3 and earlier wouldn't always release entries back to the System, even after the owning application quits.

▶ Imposing Configuration Requirements

In the final analysis, you may conclude that there are certain software and hardware resources that your application requires and others that are desirable but whose absence can be tolerated.

Minimum RAM

An applications developer should determine as accurately as possible the recommended and minimum heap size necessary for the application to operate meaningfully and set the values in the 'SIZE' resources numbered 0 (recommended size) and –1 (minimum size). In particular, the maximum and minimum number of offscreen image buffers should be accounted for in these size requirements because they are by far the largest consumers of heap memory. The main stumbling block here is that this number changes drastically for different pixel depths. For example, imagine an application with two to eight offscreen image buffers, each with dimensions of 512 pixels wide by 512 tall (see Table 9-1).

Table 9-1. Heap memory requirements

	4 bits per pixel	*8 bits per pixel*	*32 bits per pixel*
2 buffers	256K	512K	2M
8 buffers	1M	2M	8M

As you can see, such an application could perform well under most circumstances with a heap size of 2 megabytes. If the user wanted to work with 32-bit images, the application should probably instruct the user to set the size resource directly, via the entry in the application file's Finder Get Info window. The Finder's About... memory status dialog can be used to verify these memory usage calculations.

Minimum System Software

As stated earlier, Apple considers it perfectly acceptable to check for and, if necessary, require the user to install a system version greater than or equal to the one that the application deems necessary for survival. This has the added benefit of encouraging users to keep their System Software current.

Monitor Configurations

Applications (with the possible exception of palette animators) should never require the user to set a particular pixel depth, and they should never use **SetDepth()** to change display depths without the user's explicit permission to do so. In the past, when GDevices weren't well understood and GWorlds weren't available, a number of graphics applications commandeered the current GDevice (usually the main GDevice) and changed its characteristics to suit its needs. There is no longer any excuse for that kind of hardware control strategy. Create a GWorld if your application requires precise graphics environment control. Don't directly mess with public display GDevices. Enough said.

▶ 32-Bit Addressing Issues

The Macintosh runs in two modes, 24 bit and 32 bit. In 24-bit mode only 24 address bits are used, while in 32-bit mode all 32-bit address bits are used. Thus, in 32-bit mode, the address space is 256 times larger than it is in 24-bit mode.

The problem is that the memory map changes considerably between the two modes, and the Toolbox must know which addressing mode its parameters are in. For compatibility reasons, the assumption is that calls to all Toolbox traps are made in the mode the Macintosh started up in and that addresses passed into Toolbox traps are valid in that addressing mode.

One difficulty with this technique is that if the Macintosh was booted in 24-bit mode the base addresses of PixMaps can be in only

24-bit space, while video cards may require 32-bit addressing to access all of their memory. As a result, there are several memory-addressing environments that QuickDraw must deal with.

- If the base address is the same as the QuickDraw application global variable "scrnBase," which it will be for all onscreen ports and PixMaps, the base address is assumed to be 32 bit. 32-bit QuickDraw and later versions always enter 32-bit mode before drawing to the screen.
- Any other base address is assumed to be valid in the processor mode on boot unless Bit 2 of a PixMap's *pmVersion* field is set. If that bit is set, then the address is 32 bit and QuickDraw will enter 32-bit mode before accessing the pixel-image memory pointed to by the PixMap's *baseAddr* field.

The offscreen GWorld traps also use the *pmVersion* field to keep state information about the *baseAddr* for internal use. Thus, the application should set bit 2 *pmVersion* only for those PixMaps that it creates. Applications should *never* change the *pmVersion* value for a PixMap belonging to an offscreen GWorld.

▶ A Short History

The 68000 memory addressing architecture used in the Classic Macintosh family maintains a 32-bit address internal to the processor but has only 24 physical address lines actually connected to motherboard RAM. These 24 lines can address a range from $0 to $00FFFFFF. To conserve memory, many older Toolbox traps (mostly those described in *Inside Macintosh*, Volumes I, II, and III) use the high byte of the address to keep additional information.

Later processors in the 68000 family use all 32 processor address lines. This gives the processor an address range from $0 to $FFFFFFFF hexadecimal. The A/UX operating system was the first environment that required Macintosh applications to be 32-bit clean. For the A/UX release, the portions of the Toolbox that use the high byte of the address were revised. Unfortunately, some applications assumed they could still use the high byte of the address. These applications don't run under A/UX.

In System 7.0, the user can use the Memory Control Panel Device (cdev) to set whether 24-bit (which uses the 24-bit Memory Manager) or 32-bit (which uses a 32-bit clean Memory Manager) addressing mode should be used. Users with more than 8 megabytes of RAM need to use 32-bit addressing mode.

▶ Addressing Video Display Memory

As if the existence of two different addressing modes doesn't complicate matters enough, it turns out that slot space, where video memory usually resides, sometimes requires 32-bit addressing.

This information can be extracted from the video cards associated with GDevice driver, in the manner shown in Listing 9-4.

Listing 9-4. Determining a video card's slot number

```
unsigned short GDev_GetSlotID(
  GDHandle    theGDev )
{
AuxDCEHandle  theDrvrHdl;
short         slotNum;

  if (theGDev != NULL)
  {
    theDrvrHdl =
      (AuxDCEHandle)GetDCtlEntry( (**theGDev).gdRefNum );
    if (theDrvrHdl != NULL)
      return (**theDrvrHdl).dCtlSlot;
  }

  return 0x0000;
}
```

In 24-bit addressing mode, slot space ranges from $900000 to $EFFFFF, with one megabyte per slot. Valid 24-bit addresses are within the range $s00000 to $sFFFFF, where "s" is a valid slot number: $9 to $E hexadecimal.

In 32-bit addressing mode, slot space ranges from $F9000000 to $FEFFFFFF, with 16 addressable megabytes per slot; in this mode, valid slot addresses range between $Fs000000 and $FsFFFFFF, where "s" is a valid slot number.

However, there is also what's known as *super slot space*. The super slot space is accessible only in 32-bit addressing mode and ranges from $90000000 to $EFFFFFFF, with 256 megabytes per slot: valid slot addresses range between $s0000000 to $sFFFFFFF, where "s" is a valid slot number.

When 32-bit QuickDraw (or System 7.0) is not around, 24-bit slot space is used (old QuickDraw always accesses the screen in 24-bit addressing mode). However, 24-bit slot space doesn't permit access to

more than 1 megabyte of video memory, preventing the support of 32 bits per pixel. So 32-bit QuickDraw always accesses the screen in 32-bit addressing mode, using either the 32-bit slot space or the super slot space base address as defined by the video display card ROM's initialization code.

For example, the original Macintosh High-Resolution Video Card uses 24-bit slot space, with a slot base address of $Fss00000. In 24-bit addressing mode, the stripped address is $s00000, which maps to slot "s" in 24-bit slot space. But that address still works with 32-bit QuickDraw because $Fss00000, if used in 32-bit addressing mode, happens to map into 32-bit slot space as well.

The 8•24 card uses $Fss00000 (24-bit slot space) when 32-bit QuickDraw isn't installed, in which case 32-bit-per-pixel mode is not available, and $Fs000000 (super slot space) when 32-bit QuickDraw is available.

When an address is passed to a QuickDraw trap, it's assumed to be a valid 24-bit mode address. In order to pass a PixMap reference that contains a 32-bit *baseAddr* value into **CopyBits**() or some other QuickDraw trap, an application must set bit 2 of the PixMap's *pmVersion* field (pmVersion = 4) to inform the QuickDraw trap that the address is a 32-bit address and not a 24-bit one.

▶ Handling 32-Bit Addressing Mode

If you have read this far in this book, you are surely aware that your application should never write directly to the screen. Thus, you might think you never need to worry about whether you are in 32-bit mode or not. This is true as long as your application is 32-bit clean, with one exception.

The exception occurs when applications use the GWorld traps to create a GWorld and then access the GWorld's pixels directly. To get the *baseAddr* of such a PixMap, applications must call **GetPixBaseAddr**() rather than copying the embedded PixMap's *baseAddr* field value. This trap returns an address that is valid only in 32-bit addressing mode. This causes problems if the application tries to use the address in 24-bit mode. The bug will not appear until the application encounters an accelerator card which implements GWorlds on the card's memory such as Apple's 8•24 GC card with a 2 MB DRAM upgrade kit.

To avoid these problems, applications should use the following strategy:

1. Call the **GetPixBaseAddr**() trap to get the PixMap's address valid only in 32-bit addressing mode.
2. Switch to 32-bit addressing mode with the **SwapMMUMode**() trap.
3. Perform the intended task. You must call **StripAddress**() trap on any singly dereferenced Handle (master pointer) value.
4. Call **SwapMMUMode**() to restore the previous addressing mode.

Furthermore, an application cannot make any other system calls after it has switched to 32-bit mode since traps expect to be called in whatever mode the Macintosh was started in. Applications that make other trap calls after switching to 32-bit mode are likely to crash if the machine was started in 24-bit mode. The routine in Listing 9-5 shows how these calls should be used.

Listing 9-5. Using CopyBits() for colorizing

```
OSErr DoColorizedCopyBits( void )
{
Rect            srcRect, dstRect;
ColorSpec       savedFgColor;
RGBColor        coloredRGB;
GDHandle        oldGD;
GWorldPtr       oldGW, myOffGWorld;
PixMapHandle    myPixMapHandle;
OSErr           errStat;
short           iii;
long            jjj;
long            *bitsPtr;
unsigned short  myRowBytes;
char            mmuMode;

  SetRect( &srcRect, 0, 0, 1, 256 );
  errStat = NewGWorld( &myOffGWorld, 32, &srcRect, NULL, NULL, 0L );
  if (errStat != noErr)
    return errStat;

#ifdef _GET_GWPM_FIXED_
  myPixMapHandle = GetGWorldPixMap( myOffGWorld );
#else
  myPixMapHandle = myOffGWorld->portPixMap;
#endif _GET_GWPM_FIXED_
```

```
    /* get baseAddr good in 32-bit mode */
    LockPixels( myPixMapHandle );
    bitsPtr = (long*)GetPixBaseAddr( myPixMapHandle );
    myRowBytes = (**myPixMapHandle).rowBytes & 0x3fff;

    /* switch to 32-bit addressing mode to access pixel values */
    /* mmuMode contains previous mode */
    mmuMode = true32b;
    SwapMMUMode( &mmuMode );

    for (jjj = 256-1; jjj >= 0; jjj--)
    {
      *bitsPtr = jjj | (jjj<<8) | (jjj<<16);
      bitsPtr = (long *)((char *)bitsPtr + myRowBytes);
    }

    /* switch back to previous addressing mode: 24- or 32-bit */
    SwapMMUMode( &mmuMode );
    SaveFore( &savedFgColor );
    coloredRGB.red = 0xFFFF;
    coloredRGB.green = 0;
    coloredRGB.blue = 0;
    RGBForeColor( &coloredRGB );
    CopyBits(
      *myPixMapHandle, &(thePort->portBits),
      &srcRect, &(thePort->portRect), srcCopy, NULL );
    RestoreFore( &savedFgColor );
    UnlockPixels( myPixMapHandle );

    DisposGWorld( myOffGWorld );

    return errStat;
}
```

Important ▶

Applications must call **GetPixBaseAddr**() and switch to 32-bit mode in order to directly access the pixel values of an offscreen GWorld. Be sure to test your application with a card that requires 32-bit addressing to access its memory, such as the 8•24 GC card with a 2MB DRAM upgrade kit.

▶ Other Issues

There are a number of other small but significant facts that graphics developers might want to consider when designing the software architecture of their next smash application:

- **Don't overload the Memory Manager** — The performance of software that relies on a large number of small, dynamically allocated data objects can degrade significantly when its heap becomes full. The penalty is usually paid by the application when it tries to allocate large relocatable blocks or make the existing one larger.

  This has a number of implications for graphics applications design, especially when object-oriented software methodologies are employed. Many object-oriented development systems provide Handle-based support for individual objects.
- **Get others to test your work *during* development** — The only way to guarantee any degree of functionality across most of the Macintosh family is to test your software regularly and thoroughly on many different configurations. While it's unfortunate that it's so easy to write software that performs differently under even slightly differing configurations, denying this fact only leads to rude awakenings near project's end — even the most experienced Macintosh programming teams can become afflicted with compatibility shock. Begin the testing phase of software development as soon as possible so that you and your team can find and correct incompatible implementations.
- **Make the application fit the user** — Consistent with the Macintosh spirit is the idea that the best applications are designed and developed by people who understand their customers and appreciate those customers' needs and desires. Graphics-intensive applications get the most "oohs" and "ahhs" — people expect a bigger bang out of 3-D rendering and animation programs than they do out of word processors and spreadsheets, despite all advertising to the contrary. Computer graphics users tend to place a higher value on aesthetic components than do business computer users. A powerful interface is more important to a radically new visualization tool than it is to familiar tools such as databases and calculators. This doesn't mean that you need to go crazy with ornamentation: showy color, pop-up controls, and animation effects don't constitute a solid user interface, they merely augment one. Do your interface homework.

Developing applications that will maintain their functionality over the numerous Macintosh configurations is a daunting task, but the rewards to the customer are numerous. Most important, customers should not encounter problems because they are running on configurations that differ from the developer's. A second developer mandate is to hold support questions and maintenance releases to a minimum.

▶ Summary

This chapter presented a number of guidelines that Apple has put forth in order to help developers solve compatibility problems and deal with related issues effectively. The primary issues covered were determining reliance on System Software and display hardware, accessing system graphics data structures properly, and handling the differences between 24-bit and 32-bit addressing modes.

Bibliography

Graphics-intensive programming requires an understanding of a number of computer disciplines. In addition, many non-computer approaches to visual presentation can provide other insights into computational graphics. Here are a few of our favorite books on graphics and related topics.

▶ Graphics

Foley, James and van Dam, Andries 1990. *Fundamentals of Interactive Computer Graphics*, Second Edition. Reading, MA: Addison-Wesley Publishing Company. This book is one of the few good "standard" computer graphics textbooks and one of the most comprehensive overviews of fundamental computer graphics concepts. Presentation is well balanced, including "Pascal-ish" source code examples, illustrations, and mathematical formulas.

Gonzales, Rafael and Wintz, Paul 1987. *Digital Image Processing*, Second Edition. Reading, MA: Addison-Wesley Publishing Company. This textbook covers most of the fundamental techniques associated with pixel-based image collection and processing. Emphasis is on signal processing from a mathematic perspective; many readers may find it slow going.

Wolberg, George 1990. *Digital Image Warping*. IEEE Computer Society Press. A strong, thorough investigation into the craft of distorting pixel images. It strikes a good balance between mathematics, code samples, and color plates.

Vince, John 1990. *The Language of Computer Graphics.* Van Norstrand Reinhold. A reasonably comprehensive of computer graphics terminology.

Glassner, Andrew S. 1989. *3D Computer Graphics: A User's Guide for Artists and Designers.* Design Press. A thorough, yet plain-language look at the world of computer graphics. Although this book is targeted for non-technical people, most programmers will find it technically substantial.

Friedhoff, Richard Mark and Benzon, William 1989. *Visualization: The Second Computer Revolution.* Abrams. An excellent technically oriented survey of intermediate and advanced computer imaging techniques. Descriptions of terminology and usage of examples is outstanding. Chock full of lovely color plates.

Rivlin, Robert 1986. *The Algorithmic Image: Graphic Visions of the Computer Age.* Bellevue, WA: Microsoft Press. Another good image-rendering survey. Clear description of image processing algorithms, along with numerous helpful graphics and color plate examples. Solid historical perspective, too.

▶ Object Programming

Stroustrup, Bjarne 1987. *The C++ Programming Language.* Reading, MA: Addison-Wesley Publishing Company. The first chapter presents a number of design objectives for which C++ was designed, which should be of interest to OOPS programmers.

Wilson, David A. 1990. *Programming with MacApp.* Reading, MA: Addison-Wesley Publishing Company. A clear description of MacApp and its design objectives is presented. Good discussion of the issues surrounding views and view hierarchies.

▶ Text Programming

Knuth, Donald E. 1986. *The MetaFont Book: Computers and Typesetting.* Reading, MA: Addison-Wesley Publishing Company. THE standard text for text computation discussion.

Adobe Systems, Inc., 1991. *PostScript Language Reference Manual,* Second Edition. Reading, MA: Addison-Wesley Publishing Company. If you're into printing, this book is a must. Even if you're not, it's rich in rendering theory and technique.

Nelson, Ted 1981. *Literary Machines,* Second Edition. Xanadu Press. An alternative, irreverent, thoroughly enjoyable look at text and information processing. Read it if you can find it.

▶ Applied Graphics

Nelson, Ted 1974 (revised 1987). *Computer Lib/Dream Machines.* Tempus/Microsoft Press. A searching, yet mirthful look into the past and future of computing. Nelson's unique mix-and-match multimedia approach makes the serious business of analyzing the use of computers seem like a hayride.

Ambron, Susan and Hooper, Kristina, Editors 1988. *Interactive Multimedia.* Apple/Microsoft Press.

Laurel, Brenda, Editor 1990. *The Art of Human-Computer Interface Design.* Reading, MA: Addison-Wesley Publishing Company. A good survey of the issues surrounding software interface design.

Tufte, Edward R. 1983. *The Visual Display of Quantitative Information.* Graphics Press. A detailed and enjoyable investigation into the art and craft of displaying graphics-oriented data creatively and expressively.

▶ Reference

Tech Notes, Macintosh Developer Technical Support, Apple Computer, Inc.

Required Reading
#120: Drawing into an Offscreen Pixel Map
#163: Adding Color with CopyBits()
#211: Palette Manager Changes in System 6.0.2
#276: Of Time, Space and CopyBits()

Useful Reading
#181: Every PicComment Tells a Story, Don't It?
#277: Gimme Depth or Death

Inside Macintosh, Volume I, Apple Computer, Addison-Wesley Publishing Company

Required Reading
QuickDraw
Window Manager

Useful Reading
Font Manager

Inside Macintosh, Volume IV, Apple Computer, Addison-Wesley Publishing Company

Useful Reading
Font Manager

Inside Macintosh, Volume V, Apple Computer, Addison-Wesley Publishing Company

Required Reading
Color QuickDraw
Graphics Devices
Color Manager
Palette Manager
Color Picker Package

Useful Reading
Window Manager
Menu Manager
Control Manager
Dialog Manager

Inside Macintosh, Volume VI, Apple Computer, Addison-Wesley Publishing Company

Required Reading
Graphics Overview
Color QuickDraw
Palette Manager
Graphics Devices Manager
Memory Management

Useful Reading
User Interface Guidelines
Font Manager
Picture Utilities Package
Color Picker Package

Designing Cards and Drivers for the Macintosh Family, Second Edition Apple Computer, Addison-Wesley Publishing Company

Standard Apple Numerics Manual, Second Edition Apple Computer, Addison-Wesley Publishing Company. This is the only place where the Standard Apple Numerics Environment (SANE) is discussed in any detail. It's also a pretty decent reference to basic computer number theory.

32-bit QuickDraw Technical Reference, Apple Computer, Apple Programmer's and Developer's Association (APDA)

▶ Magazines

MacTutor
develop
MacTech Quarterly
APDALog

▶ Specific Articles

Using Model-View-Controller in MacApp, Robin Alger, *FrameWorks*
Palette Manager, Dave Van Brink, *develop* magazine V1, #1
Braving Offscreen Worlds, Guerremo Ortiz, *develop* magazine V1, #1

▶ HyperCard Stacks

Macintosh Graphics Primer
TechNotes Stack
Macintosh Q&A Stack
Human Interface Notes Stack

Index

A

Abstract differentiation
    irregular shapes, 270-284
    overview, 256
    regular shapes, 256-270
    selection tools, 284-309
Abstract superclass, 234, 254
Activate events, 221-222
Additive color, 94
addPin / addOver mode, 128
adMax / adMin mode, 128-129
aGDevice, 175
Alias
    overview, 326-328
    rasterizing, 328-331
    regions, 331-332
    waveforms, 329
alignPix, 176
Altering images with CopyBits()
    color separation, 320
    colorizing, 320-321, 322-326
    posterizing, 322
Anchored-line drawing tool, 265-270
Anchored-line sketching, 267
And-intersect, 32
Animating palette entry, 186-187, 194
Animation, 195-196
Ant pattern logic, 298
Ant patterns, 295
Antialiased lines, 332-334
Antialiased text, 334-338
Antialiasing by oversampling, 330
Application traps, 389
Arcs and wedges, 55-57
Arithmetic drawing. *See* adMax / adMin
Arithmetic modes, 127-133
ArtToy, 205-207
Atomic data types, 20
Attributes
    computer-generated shapes, 29
    font, 64
    GDevice, 109, 113
AugmentWindow(), 245

B

baseAddr, 28, 29
BasicMouseTool class, 253
BitMap data structure, 28

Bitmap font, 64
BitMaps, 28-29, 63
BitMapToRgn() trap, 350-352
bkColor, 318
Blend mode, 129-130
Blend transfer mode, 132
Blitters, 157
Blitting routine, 157-158
Border, 29
Bottleneck, 77-78, 160-163
Boundaries and coordinates, 73-75
Bounding box, 44
bounds, 28, 29, 137
boundsRect, 175
Brightness, 93

C

'cctb' type, 203
CDEF, 215
'cdev', 215
CGrafPort
    associating palettes with, 192
    data structure, 122-124
    PixPats, 147
Chroma, 93
Chromaticity and primary colors, 94-96
Chunky, 139
CIE chromaticity diagram, 94
Class family tree, 266
Classic color model, 89, 92
Classic QuickDraw, 84
Clip, 176
Clipboard operations, 230
Clipping, 75-77, 211, 319
clipRgn, 76, 210, 317
CLUT color model, 90
cmpCount, 138
cmpSize, 138
CMY colorspace, 104
'CNTL' resource, 203
Collected points, 270-284
Color
    addition, 128
    animation, 186-187
    blending, 130
    chromaticity, 95
    contention, 181
    cycling, 195
    display limits, 99-100
    mapping, 88
    requests, 182-183
    separation, 320, 323
    software components of, 82-89
    subtraction, 128
    theory, 93-100
    thresholding, 129
    updates, 184
Color analysis
    custom, 369-376
    median cut, 367-369
    popular, 367
Color-inhibited, 187
Color LookUp Table, 90
Color Manager, 87
Color mapping
    GDevice ColorTables, 113-114
    inverse tables, 114-119
    overview, 113
    search and complement procedures, 119-122
Color models
    classic, 92
    direct, 90-91
    fixed, 92
    indexed, 89-90
    overview, 89
Color patterns
    CGrafPort PixPats, 147
    custom PixPats, 148
    overview, 146
    PixPats, 147
Color Picker, 87, 100-104
Color QuickDraw, 84, 86

ColorBank values, 371
Colorizing
  overview, 320-321
  techniques, 322-326
  with CopyBits(), 401
Colorspace, 100
ColorTable
  ctSeed ID numbers, 201-202
  data structure, 104-105
  GDevice, 203
  specialized, 201-204
  user interface, 203-204
colrBit, 318
Comments, 13
Compatibility
  hardware configuration, 386-392
  overview, 385
  software, 393-397
  summary, 404
  32-bit addressing issues, 397-402
Configuration
  hardware, 386-392
  overview, 385
  requirements, 396-397
  software compatibility, 393-397
  summary, 404
  32-bit addressing issues, 397-402
Content region, 210
Control Manager, 20
Control Panel, 217
Control points, 29
Conversions
  CTable, 189
  Depth, 338-349
  palette, 189
  PostScript conversion, 21-22
  QuickDraw-to-PostScript, 21-22
Coordinate systems, 73-75
CopyBits(), 149-150, 152-156
CopyBits() spatial scaling, 151
CopyDeepMask(), 314
CopyMask(), 314
Courteous colors, 186, 193
'CPMT' resource, 369
Create(), 238-239
CreateRes(), 238-239
cTable argument, 175
CTable conversions, 189
ctFlags, 105
ctSeed, 105
ctSize, 105
ctTable, 105
Current GDevice, 109
Custom color selection, 369-376
Custom PixPats, 148
CWindowRecord data structure, 208-209
Cyan-magenta-yellow colorspace, 104

D

D/A, 166
Data structures
  BitMap, 28
  CGrafPort, 122-124
  ColorTable, 104
  CWindowRecord, 208-209
  GDevice, 107
  GrafPort, 70-71, 122-124
  InverseTable, 114
  pattern, 34
  PictInfo, 361
  picture, 65
  PixMap, 136-137
  PixPat, 146
  polygon, 53
  print, 232-233
  Rect, 40
  Region, 43
  RGBColor, 97
  versus objects, 16-20

Data types, 20
Debugging, 14
Depth conversion and dithering
  CopyBits(), 342-343
  depth conversion cases, 338-341
  dithered patterns, 348-349
  error diffusion, 341-342
  image resizing, 343-348
  overview, 338
Desktop interactions
  display boundaries, 224-226
  windows, palettes, and monitors, 226-227
Destination ColorTable, 319
destRect, 315
Device loop traps, 352-355, 366
Device loops, 352-355
Device-relative, 147
Dialog Manager, 21
Dialogs, 213
Difference, 32
Difference image, 135
Difference pattern, 298
DiffRgn, 47-48
Digital images, 26, 135-136
Digital-to-analog hardware, 166
Direct color, 90, 99
Direct color model, 89-91
Direct GDevices, 193-194
Direct images, 346
Direct depth conversion, 340
Dispatch() routine, 371
Display boundaries, 224-226
Display control
  application display, 178-180
  ColorTables, 201-204
  GDevice settings, 178-179
  GWorlds, 170-178
  HasDepth() and SetDepth(), 179-180
  overview, 165-170
  Palette Manager, 180-201
Dispose(), 238
Dither
  CopyBits(), 342-343
  error diffusion, 341-342
  flag bits, 176
  image resizing, 343-348
  overview, 338
  patterns, 348-349
  shrinking, 346
DoClipCopy(), 240-241
Document organization
  rasterizing, 228-229
  rendering techniques, 229-230
Document windows, 213
DoEvent(), 241-242
DoPrint(), 240-241
DoUpdate(), 240-241
Drag region, 209
Dragging
  outline, 300
  selected pixels, 304-305
  selected region, 300-307
DRAM, 400, 402
DrawChar(), 58-59
Drawing, color
  ColorTables, 104-105
  digital images, 135-136
  GrafPorts, 122-135
  graphic devices, 106-122
  models, 89-92
  overview, 81,
  patterns, 146-148
  picker, 100-104
  pixels, 148-163
  PixMaps, 136-146
  software components, 82-89
  summary, 163-164
  theory, 93-100
Drawing technique
  basic operations, 29
  GrafPort, 69-78
  overview, 23

raster graphics, 24-29
summary, 78
using shapes, 29-69
Drawing tools
irregular shapes, 270-284
overview, 256
regular shapes, 256-270
selection tools, 284-309
DrawPicture() trap, 349-350
DrawString(), 58-59
DrawText(), 59
dstMap, 314-315

E

Empty rects, 52
Encapsulation, 227
Entry
color, 185-186
stealing, 183-184
tolerance, 187
usage, 186-187
Erase (GrafVerb), 30-31
EraseRgn, 44
Error diffusion, 341-342
Error reporting, 88
Error traps, 88
Explicit arguments
image buffers, 314-315
mask region, 317
rectangles, 315
transfer mode, 315-317
Explicit color, 186, 194
Extended, 20

F

False-color images, 136
fgColor, 318
Fill (GrafVerb), 30-31
Fillets, 42
Filling pattern, 29
FillRgn, 44
FindWindowEvent(), 241-242
Fixed, 20
Fixed color model, 89, 92
Flicker, 166-168
float, 20
Focused views, 220
Font, 60
Font attributes, 64
Font Manager, 21
Frame, 166
Frame (GrafVerb), 30-31
FrameRgn, 44
Framing, 42
Free-line drawing, 279-281

G

GDevice. *See* Graphics Devices
gdFlags, 113, 175
gdPMap, 108
gdRect, 108
gdType, 108
GetGWWorld(), 247
GetRenderBounds(), 239-240
Global arguments
"theGDevice", 319
"thePort", 317-319
Go away box, 209
GrafPort
bottleneck procedures, 77-78
boundaries and coordinates, 73-75
clipping, 76-77
color, 122-135
data structure of, 70-71, 122-124
drawing technique, 69-78
overview, 69-73
regions, 212
visible area, 75
GrafVerbs, 30-31
Graphics devices
accessing the GDevice list, 111-112
attributes, 109, 113

color mapping, 113-122
ColorTables, 203
ColorTables and color mapping, 113-114
creation and initialization, 109-111
data structure, 107
list, 109, 111-112
Manager, 86-87, 107-113
overview, 106
settings, 178-179
Graphics environments, 2, 4-12
Graphics, mouse-based
drawing tools and abstract differentiation, 256-309
overview, 249-250
sketching, 250-256
summary, 310
tools, 250
Graphics programming, 1-22
Graphics techniques
ArtToy, 205-207
document organization, 227-230
GWorld window object, 243-247
overview, 205
printing, 230-234
summary, 247
window fundamentals, 207-227
window object, 234-242
Graphics Toolbox, 82-88
grayishTextOr, 130
Grayscale-inhibited, 187
Grayscale remapping, 325
Grow icon, 209
GWorld
components, 171-173
errors, testing for, 173-174
NewGWorld() and UpdateGWorld(), 175-176
overview, 170
PixMap alignment, 176-177
PixMapHandle, 172-173
purgeable and locked PixMaps, 177-178
stretch, clip, and dither, 176
techniques, 173-178
usage summary, 178
using, 170-173
window object, 243-247

H

Handle-based access, 108
Handles, 43
Hardware configuration
overview, 386
well-behaved application traps, 389
well-behaved applications, 387-392
HasDepth(), 179-180
hblank, 166
Heap memory requirements, 396
Hidden colors, 115, 117
HidePen, 46
Highlighting color, 127-128
Hilite, 130
Horizontal and vertical scaling, 151
Horizontal blanking interval, 166
hRes, 138, 160
HSL colorspace, 103
HSV colorspace, 103
Hue, 93

I

Image buffer, 137
Image buffers: srcMap, maskMap, dstMap, 314-315
Image compositing, 314
Image processing
aliasing, 326-332

antialiased lines, 332-334
antialiased text, 334-338
deep PixMaps on Classic machines, 349-350
depth conversion and dithering, 338-349
device loops, 352-355
overview, 311-312
pixel-processing traps, 312-326
summary, 355
Image resizing
dithered smoothing, 347-348
image shrinking, 344-347
image stretching, 344
Index space, 134-135
Indexed color model, 89-91
Indexed depth conversion, 339
Indexed images, 345
Indexed PixMap, 152
Inherit, 243
Init(), 238
Inset, 32
InsetRgn, 49
Instance variables, 16
Integer, 20
International Utilities, 22
Inverse tables
index, 116
index values, 117
installing, 118
resolution, 115
using, 118
InverseTable data structure, 114
Invert (GrafVerb), 30-31
Inverting, 322
InvertRgn, 44
Irregular shapes
free-line drawing, 279-281
overview, 270
polygon drawing, 281-284
IsEmptyShape, 32
IsWindowEvent(), 241-242

L

Lasso
ArtToy's, 292-294
deep pixels, 289-292
monochrome, 288-289
Lightness, 103
Line-drawing traps, 38-39
Lines, 38-39
Location (pen), 33
Locked PixMaps, 177-178
Logical transfer modes, 133-135
long, 20
long fixed point, 20
long floating point, 20
LongInt, 20
long integer, 20

M

Macintosh graphics environments, 4-9
Main GDevice, 109
Managers
Color, 87
Control, 20
Dialog, 21
Font, 21
Graphic Device, 86-87, 107-113
International Utilities, 22
Memory, 21
Palette, 87, 180-201
Print, 21-22
Process, 6
Resource, 21
Slot, 86
TextEdit, 21
Toolbox, 9, 20-22
Window, 20
Window trap, 234
Mapped color model, 90
MapRgn, 49
Marquee, 286-287
Mask region, 317

maskMap, 314-315
maskRect, 315
Median cut color selection, 367-369
Memory management, 88
Memory Manager, 21
Methods, 16
Modal control panels, 214
Modeless control panels, 214
Monitor configurations, 397
Mouse-based graphics
  drawing tools, 256-309
  overview, 249-250
  sketching, 250-256
  tools, 250
Mouse-based objects, 253-256
Multi-bit pixels, 88-89
MultiFinder see: Process Manager

N

Naming conventions, 14-15
Nested structures, 88
NewGWorld(), 175-176
NewPtr(), 28
NewRgn, 47
NotPatBic, 36
notPatCopy, 36
NotPatOr, 36
NotPatXor, 36
NotSrcBic, 36
NotSrcCopy, 36
NotSrcOr, 36
NotSrcXor, 36
NULL, 13
NULL pointer references, 88

O

Object-oriented programming software, 15-20
Objects
  GWorld window, 243-247
  mouse-based, 253-256
  RectangleTool, 262-265
  versus data structures, 16-20
  window, 234-242
Offscreen drawing
  application display, 178-180
  basic procedures, 168-170
  ColorTables, 201-204
  flicker, 166-168
  GWorlds, 170-178
  overview, 165-170
  Palette Manager, 180-201
  summary, 204
  video display, 166
Offscreen PixMaps, 142-146
offscreenGWorld, 175
Offset, 32
OffsetRgn, 49
1-bit images, 345
OOPS, 15-20
OpColor(), 132
Operand color, 127
Or-union, 32
Outline font, 64
Oval shapes, 42
OvalTool class, 260-261
OvalTool Draw() methods, 261
Override, 243

P

'PACK' resource, 358
packType, 139
Paint (GrafVerb), 30-31
Paint application, ArtToy, 205-207
PaintRgn, 44
Palette Manager
  basic concepts, 181-187
  earlier versions, 190-191
  overview, 180-181
  traps, 187-189
  using palettes, 191-201
Palettes
  activation, 189
  animation, 195-196

assignment, 188
associating with windows, 191-192
CGrafPorts, 192
conversions, 189
creating, 187
data structure, 185
direct GDevices, 193-194
disposing, 187
earlier versions, 190
entry control, 189
'pltt' resources, 194-195
usages and tolerances, 192
windows, 191-192
patBic, 36
patCopy, 36
patOr, 36
Pattern (pen), 33, 34
Pattern data structure, 34
patXor, 36
Pen
color control, 126
color traps, 124
sizes, 33
transfer modes, 36
Pens and PenStates, 33-38
PenState
location, 33
pattern, 33, 34
size, 33
transfer mode, 35-36
visibility, 33
picFrame, 383
PicHandle argument, 364
picSave, 318
picSize field, 67
PICT data format, 360
Pict opcodes, 360
PictInfo data structure, 361
Picture data structure, 65
Picture Utilities, 87, 358-376
Pictures
deep PixMaps on Classic machines, 376-384
limits, 68-69
overview, 65-67, 357-358
picture utilities, 358-376
resources, 68
summary, 384
Pixel image buffer, 137
Pixel index value, 114
Pixel-processing traps
altering images with CopyBits(), 320-326
CopyDeepMask() and CopyMask(), 314
explicit arguments, 314-317
global arguments, 317-319
overview, 312-313
Pixel value mapping, 88-89
pixelDepth, 175
Pixels, 26
Pixels, moving
CopyBits(), 149-150, 152-156
performance, 157-160
PixMap resolution, 160-163
scaling, 151
StdBits() bottleneck, 160-163
pixelSize, 138
pixelType, 139
PixMap
alignment, 176-177
creating, 142-143
data structure, 136-140
drawing, in color, 136-146
GWorld PixMapHandle, 172
indexed, 152
management, 140-141
memory diagram, 141
offscreen, 142-146
on Classic machines, 349-350, 376-384

overview, 136-137
purgeable and locked, 177-178
resolution, 160-163
traps, 140-141
PixPat data structure, 146
PixPat traps, 147
PixPats, custom, 148
planeBytes, 139
'pltt' resources, 194-195
pmReserved, 140
pmTable, 140
pmVersion, 140
PointInShape, 32
Points, 26-27
Polygon
data structure, 53
drawing, 281-284
traps, 54-55
Popular color selection, 366-367
Port-relative, 147
Posterizing, 322
PostScript translation, 21-22, 63
PrCloseDoc(), 233
PrClosePage(), 234
PrDefault(), 233
PrGeneral(), 234
Print Manager, 21-22
Print
data structures, 232-233
overview, 230-232
rendered documents, 230
traps, 233-234
Private GDevice, 109
PrJobDlog(), 233
Process Manager, 6
Programming
atomic data types, 20
naming conventions, 14-15
object-oriented, 15-19
practical guidelines, 12-20
PrOpenDoc(), 233
PrOpenPage(), 234
PrStlDlog(), 233
PrValidate(), 233
PtInRect(), 52
PtInRgn(), 52
Public GDevice, 108
Public methods
AugmentWindow(), 245
GetGWWorld(), 247
Render(), 245-246
SetOffGWorld(), 247
SetScreenGWorld(), 247
Purgeable PixMaps, 177-178

Q

qdGrafVars, 78
qdProcsPtr, 77
QuickDraw-to-PostScript translation, 21-22, 63
QuickDraw versions, 84-86

R

RAM, 396-397
Raster graphics
BitMaps, 28-29
digital images, 26
overview, 24-25
points, 26-27
Raster-scan displays, 24
Rasterizing, 328-331
Real, 20
Rect data structure, 40
Rectangle-based shapes
emptiness, 52
overview, 41-42
region calculation traps, 49-50
region problems, 53
regions, 43-46
regions and set theory traps, 47-48
using, 50-51
Rectangles: srcRect, maskRect, destRect, 315

RectangleTool class, 260-261
RectangleTool Draw() methods, 261
RectangleTool object, 262-265
Rectangular marquee, 286-287
Rectangular shapes, 258-262
RectMouseTool sketch() method, 259-260
Region
  aliased, 331-332
  bounding box, 43
  calculation traps, 49-50
  clipping, 211, 319
  content, 210
  creating, 45
  data structure, 43
  drag, 209
  dragging, 300-307
  GrafPort, 212
  mask, 317
  rectangle-based, 43-50
  structure, 209
  theRgn instance variable, 285, 287
  update, 211-213
  visible, 210, 319
  window, 209-213
RegularMouseTool class, 257-258
Regular shapes
  anchored-line drawing tool, 265-270
  RectangleTool, 262-265
  rectangular, 258-262
Render(), 239-240, 245-246
Resolution, 27, 160-163
Resource Manager, 21
rgbBkColor, 318
RGB color
  addition, 128
  blending, 130
  cube, 97-98
  overview, 93-94
  subtraction, 128
  thresholding, 129
  values, 97-99
RGBColor data structure, 97
rgbFgColor, 318
rgnBBox, 43
rgnData, 45
RgnHandles, 43
Rounded rectangles, 41
Routine, 24
rowBytes, 28, 137-138
Running ants, 295-299

S

Saturation, 93
Scaling, 151
Scan line, 166
Screen font, 63
Search and complement procedures, 119-122
Search procedure
  drawing and, 120
  example, 121-122
  techniques, 121
  traps, 121
SearchProcs, 120
SectRgn, 47-48
Selection tools
  dragging a selected region, 300-307
  lasso, 288-295
  rectangular marquee, 286-287
  running ants, 295-299
  using, 308-309
Set theory traps, 47-48
SetDepth(), 179-180
SetOffGWorld(), 247
SetScreenGWorld(), 247
Shapes
  arcs and wedges, 55-57
  attributes, 29
  GrafVerbs, 30-32
  irregular, 270-284

lines, 38-39
pens and PenStates, 33-38
Pictures, 65-69
polygons, 53-55
rectangle-based, 41-53
rectangles, 40-41
regular, 256-270
text, 57-65
short, 20
short fixed point, 20
short floating point, 20
ShowPen(), 46
Shrinking, 344-347
Size (pen), 33
Sketching, 250-256
Slot Manager, 86
SmallFract, 20
Smoothing, 347-348
Software compatibility
detecting System Software versions, 394-396
imposing configuration requirements, 396-397
overview, 393-394
srcBic, 36
srcCopy, 36, 62
srcMap, 314-315
srcOr, 36, 62
srcRect, 315
srcXor, 36
StdBits() bottleneck, 160-163
StdText() bottleneck, 132
Stretch, 176
Stretch traps, 313, 338
Stretching, 344
Structure region, 209
subPin / subOver, 128
Subpixel stretching, 347
Subtractive color, 94
Super slot space, 399
Support, 1
System (color selection) method, 366
System 7 QuickDraw, 86
System Software versions
detecting, 394-396
no color or 32-bit QuickDraw, 394-395
old, 396
overview, 394

T

Text
mixed typefaces, 60-63
overview, 57-58
traps, 58-59, 61
typeface management, 63-65
TextBox(), 59
TextEdit, 21
"theGDevice", 319
"thePort", 317-319
theRgn instance variable, 285, 287
32-bit addressing
history, 398
mode, 400-402
overview, 397-398
video display memory, 399-400
32-bit QuickDraw, 86
Tolerant colors, 186, 193
Tool palettes, 213
Toolbox Managers, 9, 20-22
Tools
anchored-line drawing, 265-270
drawing, 256-309
Graphics Toolbox, 82-88
mouse-based, 250
palettes, 213
selection, 284-309
Toolbox, 9, 20-22
TPrInfo, 232

TPrint, 232
TPrPort, 233
Transfer mode
  argument, 315-317
  arithmetic, 127-133
  logical, 133-135
  overview, 127
  pen, 35-36
Translation, 21-22, 27
Transparent, 130
Trap, 24
Traps
  application, 389
  BitMapToRgn(), 350-352
  device loop, 352-355, 366
  DrawPicture(), 349-350
  error, 88
  line-drawing, 38-39
  Palette Manager, 187-190
  pen color, 124
  pixel-processing, 312-326, 338
  PixMaps management, 140-141
  PixPat, 147
  polygon, 54-55
  print, 233-234
  region calculation, 49-50
  search procedure, 121
  set theory, 47-48
  stretch, 313, 338
  text, 58-59, 61
  well-behaved application, 389
  window, 210
  Window Manager, 234
True color, 98
24-bit color, 98
Type size, 60
Type style, 60
Typeface control, 62
Typefaces, 60-65

U

Undersampled, 328
Unfocused views, 220
UnionRgn, 47-48
Update events, 222-224, 229-230
Update region, 211-213, 223
UpdateGWorld(), 175-176
User alerts, 214
User interface ColorTables, 203-204

V

vblank, 166
Vertical blanking interval, 166
Video display, 166
Video display memory, 399-400
Video display scan pattern, 166
Video RAM, 107
Views, 215
Virtual method, 238, 239
Visibility (pen), 33
Visibility, 75
Visible area, 75
Visible region, 210, 319
visRgn, 75, 210, 223, 317
VRAM, 107
vRes, 138, 160

W

WaitMouseUp() loop, 251-252
WDEF, 223
Wedges, 55-57
Where field, 26
Window fundamentals
  color requests, 182-183
  desktop interactions, 224-227
  event-handling techniques, 221-224
  focusing, 218-221
  graphic foundations, 208-209

Manager, 20
Manager traps, 234
overview, 207-208
regions, 209-213
traps, 210
types and presentation, 213
views, 214-217
Window object
class definition, 235-237
objectives, 234
public methods, 238-242

X

Xor, 32
XorRgn, 47-48

Z

Zoom/unzoom box, 209

Titles in the Macintosh Inside Out Series

- **Extending the Macintosh® Toolbox**
  Programming Menus, Windows, Dialogs, and More
  John C. May and Judy B. Whittle
  A complete guide to programming the Macintosh interface.
  352 pages, $24.95, paperback, order #57722

- **Programming QuickDraw™**
  Includes Color QuickDraw and 32-Bit QuickDraw
  David A. Surovell, Fred M. Hall, and Konstantin Othmer
  The first in-depth reference to the Macintosh graphics system.
  352 pages, $24.95, paperback, order #57019

- **Programming for System 7**
  Gary Little and Tim Swihart
  A complete programmer's handbook to the newest version of the Macintosh system software.
  400 pages, $26.95, paperback, order #56770

- **Programming with AppleTalk®**
  Michael Peirce
  An accessible guide to creating applications that run with AppleTalk.
  352 pages, $24.95, paperback, order #57780

- **The A/UX® 2.0 Handbook**
  Jan L. Harrington
  A complete and up-to-date introduction to UNIX on the Macintosh.
  448 pages, $26.95, paperback, order #56784

- **System 7 Revealed**
  Anthony Meadow
  A first look inside the important new Macintosh system software from Apple.
  368 pages, $22.95, paperback, order #55040

- **ResEdit™ Complete**
  Peter Alley and Carolyn Strange
  Contains the popular ResEdit software and complete information on how to use it.
  576 pages, $29.95, book/disk, order #55075

- **The Complete Book of HyperTalk® 2**
  Dan Shafer
  Practical guide to HyperTalk 2.0 commands, operators, and functions.
  480 pages, $24.95, paperback, order #57082

- **Programming the LaserWriter®**
  David A. Holzgang
  Now Macintosh programmers can unlock the full power of the LaserWriter.
  480 pages, $24.95, paperback, order #57068

- **Debugging Macintosh® Software with MacsBug**
  Includes MacsBug 6.2
  Konstantin Othmer and Jim Straus
  Everything a programmer needs to start debugging Macintosh software.
  576 pages, $34.95, book/disk, order #57049

- **Developing Object-Oriented Software for the Macintosh®**
  Analysis, Design, and Programming
  Neal Goldstein and Jeff Alger
  An in-depth look at object-oriented programming on the Macintosh.
  352 pages, $24.95, paperback, order #57065

- **Writing Localizable Software for the Macintosh®**
  Daniel R. Carter
  A step-by-step guide which opens up international markets to Macintosh software developers.
  352 pages, $24.95, paperback, order #57013

- **Programmer's Guide to MPW®, Volume I**
  Exploring the Macintosh® Programmer's Workshop
  Mark Andrews
  Essential guide and reference to the standard Macintosh software development system, MPW.
  608 pages, $26.95, paperback, order #57011

- **Elements of C++ Macintosh® Programming**
  Dan Weston
  Teaches the basic elements of C++ programming, concentrating on object-oriented style and syntax.
  512 pages, $22.95, paperback, order #55025

- **Programming with MacApp®**
  David A. Wilson, Larry S. Rosenstein, and Dan Shafer
  Hands-on tutorial on everything you need to know about MacApp.
  576 pages, $24.95, paperback, order #09784
  576 pages, $34.95, book/disk, order #55062

- **C++ Programming with MacApp®**
  David A. Wilson, Larry S. Rosenstein, and Dan Shafer
  Learn the secrets to unlocking the power of MacApp and C++.
  624 pages, $24.95, paperback, order #57020
  624 pages, $34.95, book/disk, order #57021

Order Number	Quantity	Price	Total
_____	_____	_____	_____
_____	_____	_____	_____
_____	_____	_____	_____
_____	_____	_____	_____
TOTAL ORDER			_____

Shipping and state sales tax will be added automatically.

Credit card orders only please.

Offer good in USA only. Prices and availability subject to change without notice.

Name _____

Address _____

_____

City/State/Zip _____

Signature (required) _____

___Visa ___MasterCard ___AmEx

Account # _____ Exp. Date _____

Addison-Wesley Publishing Company
Order Department
Route 128
Reading, MA 01867
To order by phone, call (617) 944-3700

Programming QuickDraw™ — The Disk

All the programs and routines listed in this book are available on disk, in source code form, directly from the authors.

The disk includes the following materials:

- all source code in the book
- the complete ArtToy paintbox application source and resources, including many familiar sketching and selection tools
- GrafPort and GDevice management routines
- "useful" region generation routines
- regular polygon generation routines
- color pattern ('ppat') resizing and resource I/O routines
- color palette animation examples

To order the disk, simply clip or photocopy this entire page and complete the coupon below. Enclose a check or money order for $25.00 in U.S. funds made payable to David Surovell. Michigan residents add 4% state sales tax.

David Surovell
1322 Miller
Ann Arbor, MI USA 48103

Please send me a copy of the Programming QuickDraw™ disk. I am enclosing the amount of $25.00 in U.S. funds, plus applicable Michigan state sales tax.

Name: _____

Address: _____

City: _____

State/Province: _____

Country: _____

Zip Code: _____

Macintosh Inside Out

>26.95 USA
>34.95 CANADA

Programming QuickDraw™

DAVID A. SUROVELL
FREDERICK M. HALL
KONSTANTIN OTHMER

Here is the first in-depth, practical guide to QuickDraw™, the heart of the Macintosh® graphics system and the foundation of the intuitive Macintosh user interface. Every Macintosh programmer developing applications needs to learn how to program with QuickDraw in order to develop Macintosh software effectively.

Programming QuickDraw takes Macintosh programmers of all levels through the powerful features and capabilities of QuickDraw, including the newest improvements of Color QuickDraw™, 32-bit QuickDraw™, and System 7. The book begins with the basics of fundamental QuickDraw concepts and functions, and then covers more advanced topics in detail, including color mapping, off-screen drawing, palettes, printing, and picture analysis. Special sections also provide useful, hands-on tips and techniques for avoiding potential problems.

You will also learn how to:

- Build color pictures on the Macintosh Classic
- Maintain the highest degree of compatibility for your applications across the Macintosh platform
- Perform sophisticated image processing operations with CopyBits() and CopyDeepMask()
- Enable your applications to take advantage of QuickDraw-compliant graphics hardware and accelerators.

This thorough coverage of vital QuickDraw concepts and features makes **Programming QuickDraw** an essential resource for all Macintosh programmers.



David A. Surovell and **Frederick M. Hall** are expert Macintosh graphics programmers specializing in graphic-intensive software development. **Konstantin Othmer** is a Software Engineer at Apple Computer, Inc., and is the Project Leader for QuickDraw in System 7.

9 780201 570199 52695>

ISBN 0-201-57019-X
57019

Cover design by Ronn Campisi
Addison-Wesley Publishing Company, Inc.