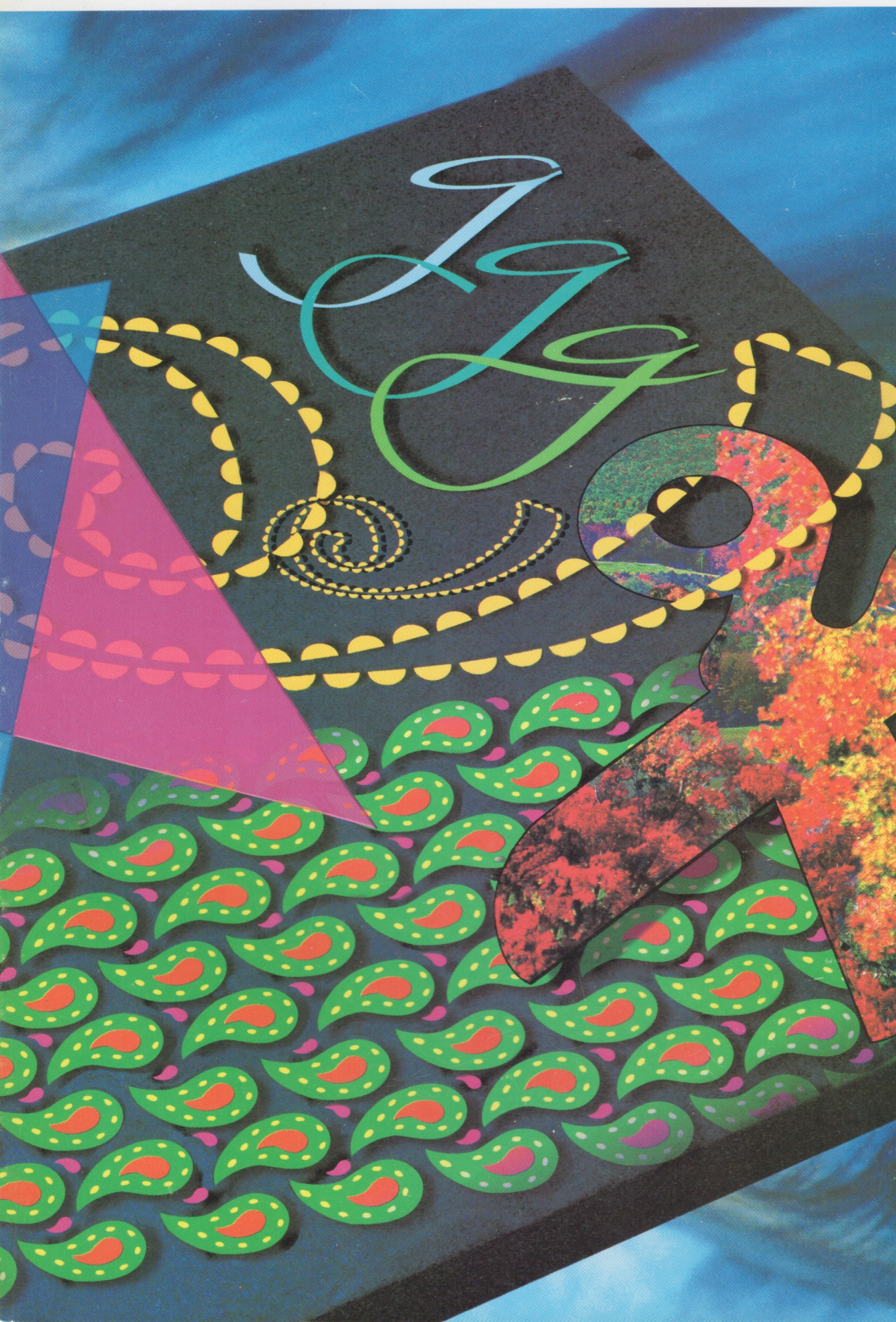


develop

The Apple Technical Journal



**GETTING STARTED
WITH
QUICKDRAW GX**

**DEVELOPING
QUICKDRAW GX
PRINTING
EXTENSIONS**

**QUICKDRAW GX
FOR POSTSCRIPT
PROGRAMMERS**

**MANAGING
COMPONENT
REGISTRATION**

**DYNAMIC
CUSTOMIZATION
OF COMPONENTS**

**FLOATING
WINDOWS:
KEEPING AFLOAT
IN THE WINDOW
MANAGER**

**WORKING IN THE
THIRD DIMENSION**

**KON & BAL'S
PUZZLE PAGE**

**MACINTOSH
Q & A**



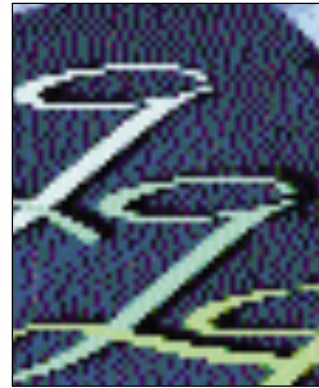
Issue 15 September 1993

EDITORIAL STAFF

Editor-in-Cheek *Caroline Rose*
Technical Buckstopper *Dave Johnson*
Our Boss *Greg Joswiak*
His Boss *Dennis Matthews*
Review Board *Pete (“Luke”) Alexander, Neil Day,
C. K. Haun, Jim Reekes, Bryan K. (“Beaker”) Ressler,
Larry Rosenstein, Andy Shebanow, Gregg Williams*
Managing Editor *Cynthia Jasper*
Contributing Editors *Lorraine Anderson, Philip Borenstein,
Robin Cowan, Toni Haskell, Judy Helfand, Rebecca Pepper,
Rilla Reynolds*
Indexer *Ira Kleinberg*

ART & PRODUCTION

Production/Art Director *Diane Wilcox*
Technical Illustration *Dave Olmos, John Ryan*
Formatting *Forbes Mill Press*
Printing *Wolfer Printing Company, Inc.*
Film Preparation *Aptos Post, Inc.*
Production *PrePress Assembly*
Photography *Sharon Beals*
Online Production *Cassi Carpenter*



Lindsay Marshall of Rucker Huggins illustrated several QuickDraw GX features for this cover. The three g’s are from the “GX-ready” typeface, Columbine, developed by David Siegel.

develop, *The Apple Technical Journal*, a quarterly publication of Apple Computer’s Developer Press group, is published in March, June, September, and December.

The *develop Bookmark* CD (or the *Developer CD Series* disc, Reference Library edition) for September 1993 or later contains this issue and all back issues of *develop* along with the code that the articles describe. The *develop* issues and code are also available on AppleLink and via anonymous ftp on <ftp.apple.com>.

EDITORIAL	Another award for <i>develop!</i> And changes made at your request. 2
LETTERS	Letters from you, plus more from us on asynchronous routines. 3
ARTICLES	Getting Started With QuickDraw GX by Pete (“Luke”) Alexander A brief introduction to QuickDraw GX, and a simple GX-aware sample. 6
	Developing QuickDraw GX Printing Extensions by Sam Weiss All about these nifty new add-ons to QuickDraw GX printing. 34
	QuickDraw GX for PostScript Programmers by Daniel Lipton The two graphics models are compared, along with useful code snippets for each. 51
	Managing Component Registration by Gary Woodcock For those cases where you may need to manage the component registration process, here’s how. 74
	Floating Windows: Keeping Afloat in the Window Manager by Dean Yu A way to implement floating windows without patching traps, and a library you can use in your own application. 89
	Working in the Third Dimension by Jamie Osborne and Deanna Thomas This article shows off a nice 3-D interface and presents a set of MacApp objects you can use to create your own such interface. 103
COLUMNS	The Veteran Neophyte: Through the Looking Glass by Dave Johnson Dave explores the mathematics of symmetry and finds some surprises. 71
	Somewhere in QuickTime: Dynamic Customization of Components by Bill Guschwan A sample derived media handler that “speaks” the text track in a movie. 84
	View From the Ledge by Tao Jones An office survival guide for the socially and politically inept. 115
	KON & BAL’s Puzzle Page: I’m Here to Serve by Konstantin Othmer and Bruce Leak Try your skill (or is it luck?) on yet another puzzle from those masters of Macintosh machinations, KON and BAL. 132
Q & A	Macintosh Q & A Apple’s Developer Support Center answers your product development questions. 117
INDEX	137

© 1993 Apple Computer, Inc. All rights reserved.

Apple, the Apple logo, APDA, AppleLink, AppleShare, AppleTalk, ImageWriter, LaserWriter, LocalTalk, MacApp, Macintosh, MPW, and MultiFinder are trademarks of Apple Computer, Inc., registered in the U.S. and other countries. AppleScript, ColorSync, develop, Finder, KanjiTalk, Macintosh Quadra, MoviePlayer, QuickDraw, QuickTime, Sound Manager, System 7, TrueType, and WorldScript are trademarks of Apple Computer, Inc. HyperCard, HyperTalk, and MacDraw are registered trademarks of Claris Corporation. PostScript is a trademark of Adobe Systems Incorporated, which may be registered in certain jurisdictions. Helvetica is a registered trademark of Linotype Company. UNIX is a registered trademark of UNIX System Laboratories, Inc. All other trademarks are the property of their respective owners.



CAROLINE ROSE

Dear Readers,

develop does it again! We're thrilled to announce that we've won another Excellence award, this time in the 1993 International Technical Publications Competition, sponsored by the Society for Technical Communication.

Some things that are worth noting in this issue:

- We're very happy to finally have a female author. Welcome to Deanna Thomas; may she be the first in a long line.
- We've added a strange new column called "View From the Ledge"; please let us know what you think of this irreverent (or is that "irrelevant"?) addition.
- We're temporarily without a Print Hints or Graphical Truffles column. But there's a lot of information about printing and graphics in our three QuickDraw GX articles.

Finally, here are two changes that have happened as a result of your feedback:

- Tech Notes are numbered again, this time within each category of Note. References in *develop* will include the new number after the category; for example, we might refer you to the Macintosh Technical Note "Fond of FONDS" (Text 21). With this issue we finally stop giving the former number of a Tech Note, as in "(formerly #91)"; those old numbers are long gone.
- New *Inside Macintosh* is now on the *develop Bookmark* CD. It was painfully missing from Issue 14's CD, but we have seen the error of our ways and have quickly rectified the situation.

Please keep letting us know what you want; it pays!

Caroline Rose
Editor

2

CAROLINE ROSE (AppleLink CROSE) has written and edited more technical documentation than she cares to remember. In past work lives, she was also a programmer and (gasp!) a manager. She's worked for Tymshare, Apple, NeXT, and Apple, in that order. But no previous job compares to the variety and fun she enjoys as editor of *develop*. Caroline is still raving about the great time she had at the Worldwide Developers

Conference in May. We suspect the highlight was when a developer asked her to sign his copy of *develop* — or was it when she took the Karaoke dare at the WWDC party? Recent delights outside of work include singing like a wannabe Bonnie Raitt at jam sessions with her friends and listening to tapes of John Prine and Richard Thompson from an incredible music festival she attended in a mountain meadow. •

LETTERS

NEW TECH NOTE NUMBERS

In *develop* Issue 14 Peter Fink complains about the loss of Tech Note numbering. In your answer you write: “We’re always open to suggestion, but so far you’re the only one to mention this. If others reading this reply have similar feedback, I hope they’ll let us know.”

Actually, I and others have complained about the loss of Tech Note numbers for quite some time, but sometimes talking about these things is like complaining about the weather. I suspect most people just grumble along and don’t complain in writing.

—Johan G. E. Strandberg

Good news: Numbers are back; see the editorial on page 2 and the Tech Notes on this issue’s CD. It’s too bad when people grumble without writing; the latter is much more effective, and we really appreciate the effort. Thanks!

—Caroline Rose

DEBUGGING LESSON FLUB

There seems to be a problem with the point made in the “Debugging Lesson” letter in the Letters section of *develop* Issue 14. According to the source listed, there would be no problem even if memory moved when FillWithData was called. Since the handle is being passed as a handle (that is, not dereferenced), and handles are 4-byte pointers (which get pushed onto the stack as is), FillWithData would always get a valid handle, which it can dereference internally to its heart’s content.

Unfortunately, there seems to be a lot of confusion (and a great deal of paranoia)

over the usage of handles. As long as you pass handles as handles (and watch your dereferencing), you’ll be OK.

—Charlie Reading

Um, oops. (Imagine, if you will, me standing gaping and red-faced, astonished at my own carelessness and idiocy.)

You’re right, of course; the danger is only in passing dereferenced handles to routines in other segments, and the code in the letter passes the handle itself. I guess I just read the text and didn’t really look closely at the code before replying. Obviously, it was only intended as an example, and the writer’s point is still valid: passing dereferenced handles to routines in other segments is dangerous. You should either pass the handle itself (as the example code did!) or lock the handle first.

Sorry about the confusion.

—Dave Johnson

BABBLING ON

I enjoyed reading Dave Johnson’s column in *develop* Issue 13 (“Tower of Babble”). While reading his discussion of natural versus programming languages, I think I was able to put my finger on what has bugged me about HyperTalk® for years. Natural languages imply a fluidity of meaning, giving its users great flexibility for nuances, emphasis, and brevity. HyperTalk *looks* like a natural language, but it possesses none of that fluidity of meaning. In HyperCard®, SET has but one meaning. But in my Oxford dictionary, “set” has 194 definitions! In other words, HyperCard looks so much like English, one winds up assuming that all kinds of structures could be used

WE HATE IT WHEN YOU DON’T WRITE

We welcome timely letters to the editors, especially from readers reacting to articles that we publish in *develop*. Letters should be addressed to Caroline Rose (or, if technical *develop*-related questions, to Dave Johnson) at Apple Computer, Inc., 20525 Mariani Avenue, M/S 303-4DP, Cupertino, CA 95014 (AppleLink CROSE or JOHNSON.DK). All letters should

include your name and company name as well as your address and phone number. Letters may be excerpted or edited for clarity (or to make them say what we wish they did). •

as long as they make sense in English (for example, “set me to true”).

Allow me to submit a suggestion, which you may wish to pass along to the software gods. Since Apple has now all but declared that C and C++ are the only computer languages the human race will ever need, maybe it would be great to have a HyperTalk-to-C translator. Why? Well, C is a write-only language (no one ever can figure out what’s happening in someone else’s C listing) and HyperTalk is a read-only language (it’s a cinch to understand, but impossible to code in). With a translator, we could write code that still could be understood, just like when we used to code in Pascal. (Yes, this is a joke, from a die-hard Pascal enthusiast.)

Thanks again for a nifty column!

—Kevin Killion

I’m glad you liked the column. Your comment about HyperTalk is very well taken! I think that’s what always bothered me about it, too, though it was just a sort of vague unease. Unfortunately, I think people who didn’t learn “regular” programming languages first may have a distinct disadvantage: we “real” programmers know up front how limited and terse and strict programming languages are, and we don’t expect anything more. But power users who learn HyperTalk (or, more likely now, AppleScript) as their very first programming language may be in for a struggle if they don’t keep reminding themselves that it’s not a natural language.

As a die-hard C enthusiast, I’ll graciously ignore your slams against my favorite language, and assume that since you’re obviously an intelligent person, sooner or

later you’ll realize the error of your ways and come around to the truth, bemoaning the years you’ve spent in bondage to an uptight compiler.

—Dave Johnson

ASYNCH SUPPORT ON A/UX

After reading through Jim Luther’s Asynchronous Routines article in *develop* Issue 13, I spiffed up my application to make all my file handling asynchronous, anticipating the glorious day when all the drivers on my Macintosh support asynchronous calls. But since the SCSI Manager isn’t asynchronous yet, there’s no apparent difference to the user of my code (or to me, the hard-working programmer who wants payoff for my labors).

Is there any platform or configuration I can test my application on and see the results of my effort?

—Tony Amaretto

Try your code on A/UX 3.0.1 (the operating system for the Apple Workgroup Server 95); it features an enhanced File Manager that supports asynchronous calls to UNIX® file systems simply by using the techniques in Jim Luther’s article. AppleShare Pro takes advantage of this capability on AWS 95 to get performance up to four times better than AppleShare 3.0’s under System 7 (your mileage, as always, will vary).

For more information related to the Asynchronous Routines article, see the box on the next page.

Have fun!

—Dave Johnson

4

SUBSCRIPTION INFORMATION

Subscriptions to *develop* are available through APDA (see inside back cover for APDA information), or you can use the subscription card in the back of this issue. Please address all subscription-related inquiries to *develop*, Apple Computer, Inc., P.O. Box 531, Mt. Morris, IL 61054 (or AppleLink DEV.SUBS). •

BACK ISSUES

For information about back issues of *develop* and how to obtain them, see the last page of this issue. Back issues are also on the *develop* Bookmark CD and the *Developer CD Series* disc (Reference Library edition). •

MORE ON ASYNCHRONOUS ROUTINES IN ISSUE 13

For developers interested in the “Asynchronous Routines on the Macintosh” article in *develop* Issue 13, here’s some new information that has surfaced since then.

StackSpace. Contrary to the advice on page 28 of the article, you should not call StackSpace at interrupt time, because the Memory Manager might not be in a consistent state. Furthermore, StackSpace clears MemErr, which may have an adverse effect on the current process’s handling of Memory Manager errors.

PPC polling. Unlike the Device Manager and File Manager, the PPC Toolbox stuffs the result of an asynchronous routine into ioResult *before* it’s really done with the parameter block. If your interrupt code — such as a VBL task — polls ioResult periodically to check for completion and reuses the parameter block to make another call, the system can crash because one or more system queues will be corrupted.

Context switching. The System 7 Process Manager (and MultiFinder under System 6) will wait until all currently active asynchronous requests to the File Manager have completed before performing a context switch. This check was added for compatibility reasons to prevent system crashes caused by a few applications that accessed program globals within File Manager completion routines without restoring their A5 world. What this means to you is that if your application makes an asynchronous File Manager call and then calls WaitNextEvent or GetNextEvent, the system may wait for your call to complete. If the asynchronous File Manager call takes a long time to complete, it will appear to the user that the system isn’t responding.

Synchronous drivers. Although you can execute low-level file access routines asynchronously, a volume’s underlying device driver may not support asynchronous operations. Once the File Manager passes a request to a synchronous driver, that driver doesn’t give up control until it has completed the task. Synchronous drivers (such

as those using the current SCSI Manager) affect programs using asynchronous File Manager calls in two ways: unexpected pauses and unsightly stack frame buildup.

- When calling the File Manager asynchronously, your program passes control to the File Manager and the request is placed in a queue or, if the queue is empty, the request is handled immediately. Either way, once a synchronous driver gets the request, it retains control until it has responded to the request. If the request takes a long time to complete, the user may think the system isn’t responding. If your program is a background task, these pauses will affect the performance of the current foreground application. To keep this to a minimum, avoid time-consuming asynchronous File Manager requests.
- Chaining asynchronous File Manager calls will work on volumes controlled by synchronous drivers, but watch out: When the driver is synchronous, the stack frames from the system and your completion routines will keep building up on the current stack until the last asynchronous call in the chain completes, or until the stack overwrites the current heap. You’ll need to break the asynchronous File Manager call chain every few completion routines. A simple way to do this is to start a Time Manager or VBL task from your File Manager call’s completion routine, and let the task start the next asynchronous File Manager call in the chain.

File Sharing and AppleShare. Chained asynchronous File Manager calls can fail when either the Macintosh File Sharing or AppleShare 3.0 (not 3.0.1) file server is running. The file server software intercepts almost all calls made to the File Manager. Due to how the file server keeps track of what requests it has or hasn’t seen, there are a few situations where the file server can do the wrong thing with chained asynchronous File Manager calls. To make sure the file server sees and handles all chained asynchronous File Manager calls correctly, use two parameter blocks for the chained calls and switch parameter blocks at every completion routine.

Thanks to François Grien and Lawrence D’Oliveiro for providing some of this information. •

GETTING STARTED WITH QUICKDRAW

GX

A beta version of QuickDraw GX comes to you on this issue's CD. As you contemplate the vast scope of it all, you may wonder how you're ever going to get your arms around this new imaging technology. Not to worry — this article will get you started. It walks you through the steps to getting QuickDraw GX up and drawing and presents a simple "GX-hip" application shell that incorporates the basics for you to experiment with.



PETE ("LUKE") ALEXANDER

QuickDraw GX offers developers a totally new and markedly improved way of imaging on the Macintosh. Yes, you'll have to learn the new system, but look at what you get: The API is simpler and the human interface is better. The amount of control your application can exercise over text and graphics has been greatly increased. Your application will be able to produce consistent output no matter what the output device. And extensive support for color is built in throughout the system.

With the beta version of QuickDraw GX in hand, you're no doubt eager to create a QuickDraw GX application and start drawing. This article covers just about everything you need to know to get started: initializing QuickDraw GX, using windows, creating and manipulating QuickDraw GX shapes, printing, and debugging. By way of illustration, we discuss the QuickDraw GX shell that you'll find on this issue's CD. But before we do that, let's take a quick look under the hood at the major features of QuickDraw GX and how it fits into the Macintosh architecture.

QUICKDRAW GX: A QUICK LOOK UNDER THE HOOD

QuickDraw GX coexists happily with QuickDraw, thank goodness. It doesn't replace QuickDraw, but instead "moves in next door," so you still live with a Macintosh Toolbox based on QuickDraw. You can run QuickDraw-based applications on a QuickDraw GX system. These applications won't even notice that QuickDraw GX is installed, but they'll be able to take advantage of some of the QuickDraw GX

6

PETE ("LUKE") ALEXANDER started out as a meteorology major in college and to this day is great at forecasting the weather. He'd be happy to see nothing but blue skies, and in all kinds of weather he strives to incorporate something blue in his attire. He loves raw carrots, despises cooked ones. He hates waiting in lines and wearing a wristwatch. Sometimes when he's driving down the street and spots someone sitting

at a bus stop, he'll honk his horn to wake them from their dazed stupor. And when he sees bicyclists sitting in their bike seats while riding up hills, he's been known to yell out the window, "Stand up, you weenie!" •

printing features, including improved background printing to all devices, desktop printers, print job queuing, and better type management.

QuickDraw GX has three major pieces: graphics, typography, and printing. You can visualize the relationship of these three different pieces to each other and to QuickDraw as shown in Figure 1.

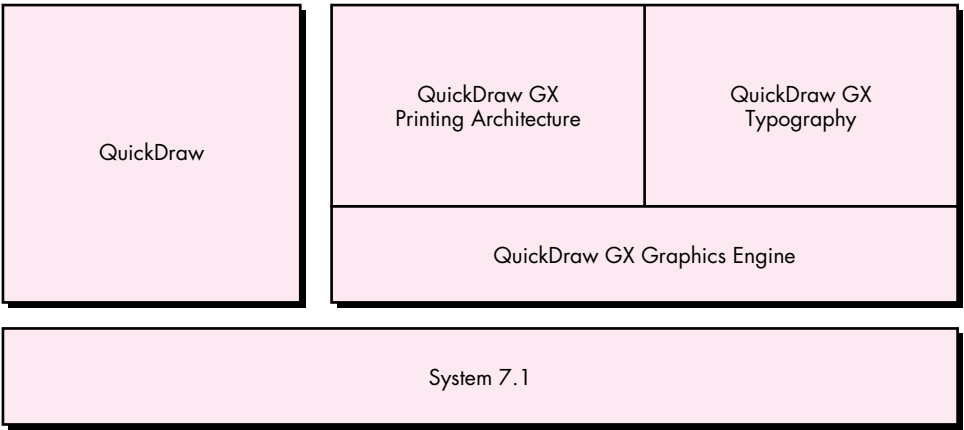


Figure 1
QuickDraw and the Pieces of QuickDraw GX

GRAPHICS

The basic building block of QuickDraw GX graphics is the *shape*. A shape is an object that contains, among other things, a geometry of some type and a fill property that specifies how the geometry should be interpreted when drawn (such as framed or filled).

There are four basic types of shapes, classified by the nature of the geometry they contain: geometric shapes, typographic shapes, bitmap shapes, and pictures.

- A *geometric shape* contains a primitive geometry: a point, a line, a curve, a rectangle, a polygon (a series of points connected by straight lines), or a path (a series of points connected by straight or curved lines). In addition, there are two other special geometric shape types: empty (no geometry at all) and full (covers the entire coordinate system).
- A *typographic shape* contains text, glyphs (renditions of individual characters or character combinations over which your application has direct control), or layouts (pieces of text for which QuickDraw GX automatically chooses and positions glyphs, given certain information by the application).

If you're a PostScript language whiz
making the transition to QuickDraw GX, you'll find the article "QuickDraw GX for PostScript Programmers" in this issue helpful. •

- A *bitmap shape* contains a reference to a block of memory containing a bit image, as well as information on how to interpret the bits: the pixel size, color space, color set, and color profile.
- A *picture* contains a list of other shapes. The shapes in the list can be other pictures, so that a picture is actually a hierarchical database of shapes.

Besides containing a geometry, a shape contains references to three other objects that describe how it should be rendered. These objects are the style, the transform, and the ink.

- The *style* defines the pen thickness, the place where the pen draws (inside, outside, or on the geometry), the kind of start and end cap (such as round, pointy, or square), and ways to dash, join, and pattern shapes. For a text shape, the style also defines the font, size, variation, and text face.
- The *transform* controls the skew, scale, perspective, and clipping of the geometry. It also specifies where to draw it and how to hit-test it.
- The *ink* tells the system which color to draw the geometry in. Ink also includes information about the color matching and transfer mode.

Some of these objects in turn contain references to other objects. For example, a transform points to a list of *view port* objects that describe where to draw the geometry. A view port is like a QuickDraw *grafPort* in that it defines an area of local space as a drawing environment. Unlike a *grafPort*, though, a view port doesn't contain state information about the drawing environment (pen, color, transfer mode, and so on). A view port contains the mapping used to convert from the view port's local space to a global space described by a *view group*. A view port object points in turn to a list of *view device* objects, which describe the clip shape, mapping, and bitmap associated with a physical device such as a monitor.

A shape can also have one or more *attributes*, which modify the shape's behavior. These attributes enable your application to specify how a shape is edited or how QuickDraw GX stores the shape. For example, if you set the shape attribute *gxMapTransformShape*, this tells QuickDraw GX that you want it to manipulate the transform referenced by the shape, instead of directly manipulating the data contained within the geometry of the shape.

Figure 2 depicts the shape object and what it references. The owner count is the number of other objects within the application that reference that object. The tag list is a list of tag objects, which are simply containers for any data the application associates with the owning object.

8

Shapes are completely resolution independent, which enables accurate representation at any size to the screen or printer. •

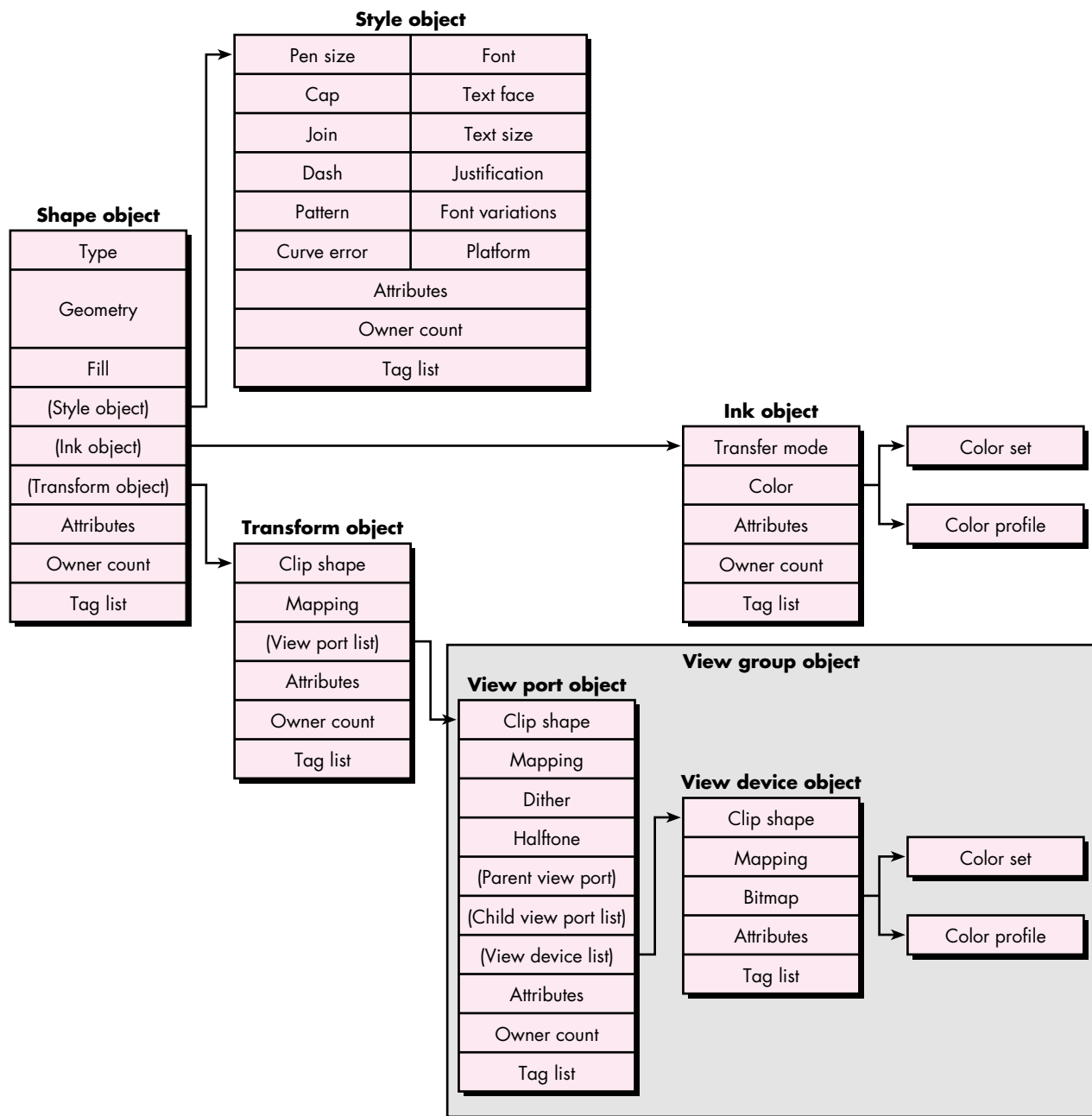


Figure 2
The Shape Object and What It References

TYPOGRAPHY

QuickDraw GX has a sophisticated typographic model that's fully integrated with graphics. The ability to do kerning, tracking, and justification, as well as ligatures and ornamental forms of various characters, is provided by the line layout routines, supported by the QuickDraw GX smart font format. The line layout routines work with the typographic information contained in the TrueType GX and Type 1 GX fonts to give you a ton of control over how text is placed on a page.

Because QuickDraw GX typography is fully integrated with graphics, you can rotate, skew, and change the perspective of typographic shapes the same way you can geometric shapes. You can use the text shape to draw a line of text with one style. The glyph shape enables you to draw text in several styles and graphically manipulate each glyph. The layout shape uses the information contained in a TrueType GX or Type 1 GX font to automatically kern, justify, and track, and to support ligatures, final forms (special forms found at the ends of words), and ornamental forms of the various glyphs contained within the layout shape.

Note that although QuickDraw GX supports all existing Macintosh font formats (Type 1, bitmap, and TrueType), to take full advantage of its extensive line layout capabilities you must use TrueType GX or Type 1 GX fonts.

PRINTING

QuickDraw GX improves printing for both users and developers. Users get an improved human interface, and developers get much more control and functionality. From the application's point of view, QuickDraw GX offers true device independence: you can send the same data to all supported devices and the output will be rendered appropriately on each device.

QuickDraw GX introduces three new printing concepts: desktop printers, portable digital documents, and printing extensions.

Users can create *desktop printers* with the Chooser. These are represented as icons on the desktop and are full Finder citizens; users can drag and drop print files and documents to them. Users can also manage the print queue and redirect print files and documents by dragging them to and from desktop printers, and can share desktop printers with other users via PrinterShare GX.

A *portable digital document* (PDD) file contains all the objects required to render a document on a screen or printer, so you can open, review, and print the file on any system running QuickDraw GX without the application or fonts used to create the document. When a PDD file is created, only the glyphs used in the document are saved along with it; since the document can't very well be edited, the PDD is secure for transporting fonts. When you print, you can save the print job as a PDD with or without the fonts required. If you know that the person you're sending the PDD file to has the fonts you used, you can choose not to save the fonts with the PDD.

Printing extensions are small standalone pieces of code that modify the behavior of printing and give users vastly increased control, at a system rather than a program level, over how a printed page looks. For example, through a printing extension the user can direct a printer to print “Confidential” diagonally across each page, no matter what program is doing the printing. The user selects a printing extension from a list displayed in the expanded Print, By Page Setup, and Document Setup dialogs (which appear when More Choices is clicked in the regular dialogs).

The API for QuickDraw GX printing gives you easy access to information about the page size and orientation of a print job and enables you to keep the user from changing these settings.

QuickDraw GX supports raster, vector, and PostScript™ devices. The bad news is that if your system is running QuickDraw GX, you won’t be able to use any non-QuickDraw GX printer drivers. The good news is that because QuickDraw GX provides system-level support for developing printer drivers, it’s a whole lot easier to develop printer drivers for QuickDraw GX than it is for the old QuickDraw-based printing architecture — you can plan on months of development time as opposed to years. And in many cases you may find that a printing extension, which is easier yet to develop, will suffice to implement the desired functionality; for more information, see the article “Developing QuickDraw GX Printing Extensions” later in this issue.

PROGRAMMING AMENITIES

QuickDraw GX offers you some truly useful programming goodies: libraries of handy high-level routines, extensive error-handling capabilities, and a powerful new debugging tool called GraphicsBug.

THE QUICKDRAW GX LIBRARIES

As you cruise around the QuickDraw GX folder on this issue’s CD, you’ll notice a folder named Libraries. Open it and you’ll find libraries of code for many common graphics, line layout, and printing tasks. These provide sample code that most applications will need in order to create a QuickDraw GX application. But unlike Macintosh Toolbox code, this library code can be modified or extended by you to meet your own particular needs. All the library code is based on core QuickDraw GX calls.

ERROR HANDLING IN QUICKDRAW GX

The goal of QuickDraw GX’s error-handling capabilities is to never allow QuickDraw GX to crash your Macintosh, and to inform you anytime QuickDraw GX can’t complete an operation. QuickDraw GX uses two different models for handling errors: one for graphics and layout errors and another for printing errors. We’ll discuss graphics and layout errors here. Printing errors are described later in this article, under “Basic Printing in QuickDraw GX.”

The By Page Setup and Document Setup

dialogs are new in QuickDraw GX. They’re described later in this article under “Basic Printing in QuickDraw GX.” •

There's both a debugging and a nondebugging version of the combined graphics and layout portions of QuickDraw GX. The debugging version provides extensive error-handling capabilities to help you debug your applications under development. The nondebugging version is lean and mean; it has fewer error-handling capabilities and is faster than the debugging version. You can differentiate between the two versions by their sizes and version strings: the nondebugging version is smaller, and the version string for the debugging version has the word "debug" in it. When you're developing your QuickDraw GX application, you should be using the debugging version.

In the debugging version, information about internal data and drawing problems comes in three flavors: notices, warnings, and errors. Only a few selected errors and warnings are issued in the nondebugging version.

Notices. A notice informs you that the operation you're performing isn't really needed. Notices aren't necessarily bad things; they're just information to help you improve the efficiency of your application. For example, if you've already colored a shape and you try to color it again, you'll receive the following notice in the installed debugger:

```
GRAPHICS NOTICE: color already set
```

Warnings. A warning informs you that QuickDraw GX doesn't allow the operation you're trying to perform. While this might not cause any problems, you also might not get the result you expected. For example, if you try to use a font that isn't available, QuickDraw GX will substitute the default font and give you the following warning:

```
GRAPHICS WARNING: font substitution took place
```

Errors. An error means that QuickDraw GX couldn't draw your shape or complete a routine. For example, if you try to draw an empty shape or one that hasn't been defined, you'll receive the following error:

```
GRAPHICS ERROR: shape is nil
```

Checking for drawing errors. Once you've finished developing your application, you'll still want to be able to check for drawing errors. The QuickDraw GX routine `GXGetShapeDrawError` lets you do this and, in case of an error, fail in a graceful manner. For example, this code fragment checks that drawing was successful and fails if it wasn't:

```
GXDrawShape(gthePage);

if (drawingError = GXGetShapeDrawError(gthePage) != noDrawError)
    // Your error-handling code here!
```

12

For a complete list of all the errors, warnings, and notices provided by the graphics and layout portions of QuickDraw GX, take a look at the `graphics errors.h` header file. •

Routine naming in QuickDraw GX is very predictable and logical. All calls preceded by "GX" are from the core API, while ones without "GX" are from the library or application code. In addition, all calls pertaining to the same object are very similar, and once you grasp how to operate on one object, you pretty much know how to operate on all objects. •

Ignoring notices and warnings. Sometimes you might want to ignore a particular notice or warning because you know what you're doing. Use these routines to ignore a notice or warning:

```
void GXIgnoreGraphicsNotice(gxGraphicsNotice notice);
void GXIgnoreGraphicsWarning(gxGraphicsWarning warning);
```

For example, if you wanted to ignore a notice about recoloring a shape, you would make this call:

```
void GXIgnoreGraphicsNotice(color_already_set);
```

When you call `GXIgnoreGraphicsNotice` or `GXIgnoreGraphicsWarning`, the notice or warning is added to the top of the notice stack or warning stack, respectively. (It's added to the stack even when not ignored, but the stack handling is taken care of behind the scenes for you in that case.) So you must balance this with a call to one of the following routines to ensure that you don't overflow the notice or warning stack:

```
void GXPopGraphicsNotice(void);
void GXPopGraphicsWarning(void);
```

In the nondebugging version, where notices and most warnings aren't available, calling the `GXIgnoreGraphicsXXX` and `GXPopGraphicsXXX` routines still results in a trap call and dispatch even though they just return immediately. There may be a small performance penalty for this, so you should remember to remove the unnecessary calls for a shipping application.

Grabbing errors, notices, and warnings. In the nondebugging version, you receive only a few selected errors and warnings. If you've tested your application thoroughly, these should be the only errors you see:

```
out_of_memory
not_enough_memory_for_graphics_client_heap
graphics_client_memory_too_small
could_not_create_backing_store
```

These should be the only warnings you see:

```
<<font or character>>_substitution_occurred
<<map, move, scale, rotate, or skew>>_shape_out_of_range
<<map, move, scale, rotate, or skew>>_transform_out_of_range
```

You probably don't want your user to end up in a debugger or with a system bomb, so you should catch errors by calling the `GXGetGraphicsError` routine; you can then handle the error appropriately within your application or present it to the user.


```
gxGraphicsError GXGetGraphicsError(gxGraphicsError *stickyError);
```

You can also grab notices (in the debugging version only) and warnings with these calls:

```
gxGraphicsNotice GXGetGraphicsNotice(gxGraphicsNotice *stickyNotice);  
gxGraphicsWarning GXGetGraphicsWarning(gxGraphicsWarning *stickyWarning);
```

GRAPHICSBUG: A POWERFUL NEW DEBUGGING TOOL

The only way to create and modify shapes in QuickDraw GX is through the public API; you can't operate on any data directly. This is a very good thing because it lets Apple expand the system in the future with minimal compatibility risk. But if you can't see the data you're working with, won't debugging be a nightmare? Here's where GraphicsBug comes to the rescue. GraphicsBug is an application that enables you to inspect the contents of any QuickDraw GX graphics or layout object to make sure it contains the correct information. The command set is very similar to that of MacsBug; just type "?" to get a list of the commands available. GraphicsBug works only in the debugging version of QuickDraw GX 1.0b1 but in both versions of later QuickDraw GX releases.

INITIALIZING QUICKDRAW GX

Now that we've checked out the horsepower under the hood and the amenities built in for programmers, we're ready to get QuickDraw GX up and drawing. The first step is to initialize QuickDraw GX, but before you do, you need to make sure the user has installed it. Use the Gestalt selector 'grfx' to determine whether the graphics and typography portions of QuickDraw GX have been installed, and the Gestalt selector 'pmgr' to determine whether QuickDraw GX printing has been installed.

In the case of our QuickDraw GX shell, the following routine finds out which parts of QuickDraw GX have been installed:

```
Boolean QuickDrawGXAvailable()  
{  
    long theFeatureInQuestion;  
  
    if (Gestalt('grfx', &theFeatureInQuestion) == noErr)  
    {  
        if (Gestalt('pmgr', &theFeatureInQuestion) == noErr)  
            gQDGXPrintingInstalled = true;  
        return (true);  
    }  
    return (false);  
}
```

The QuickDraw GX shell uses the global variable `gQDGXPrintingInstalled` to determine if QuickDraw GX printing has been installed. If it has, the printing menu items in the File menu are enabled. Otherwise, an alert tells the user that QuickDraw GX printing hasn't been installed, and the application works without printing.

Once you know that the user has QuickDraw GX, you're ready to initialize it. After the generic Macintosh Toolbox initialization, you create a new graphics client to allocate memory. Then you can set up error handling and validation as an aid to tracking down problems, although if you're eager to get on with drawing, you don't have to do this right now. Finally, you can initialize the common color library to get quick and easy color. In the QuickDraw GX shell, the routine `QuickDrawGXInit` does all of this initialization.

CREATING A NEW GRAPHICS CLIENT

A graphics client is a reference to the block of MultiFinder memory used by QuickDraw GX graphics and layout called the *QuickDraw GX heap*. When your application creates a new graphics client, QuickDraw GX usually allocates this block of memory. The QuickDraw GX heap contains all the graphics and layout objects your application creates while running QuickDraw GX, as well as a few objects QuickDraw GX uses to manage the heap. (See “Managing Memory in the QuickDraw GX Heap” for more details.)

The simplest way to create a graphics client is to call `EnterGraphics`, which defines a client for you based on some fundamental assumptions. If you want to have more control over the graphics client you create, call `GXNewGraphicsClient`:

```
gGraphicsClient = GXNewGraphicsClient(nil, gGraphicsHeapSize * 1024, 0L);
```

The variable `gGraphicsClient` holds the new graphics client. You can use this variable anytime you need to access this graphics client. In our shell, we only need this variable when the application shuts down. The first parameter tells the Memory Manager where we want to create the QuickDraw GX heap. A `nil` value tells the Memory Manager to create the heap within MultiFinder memory. That's usually where you want it, but you can also specify a pointer to a block of memory in your application heap or even the system heap. The second parameter defines the size of the heap in bytes. Our shell uses a 115K heap (`gGraphicsHeapSize = 115`); if you pass 0, you get the default heap size of 600K. The last parameter, named `separateStack`, tells QuickDraw GX to allocate a stack for the graphics client, which is necessary if that client is going to run at interrupt time. To get a separate stack, just pass any nonzero value that defines the stack size.

SETTING UP ERROR HANDLING AND VALIDATION

After setting up the graphics client, you should enable the error, warning, and notice capabilities if you want to make debugging easier for yourself down the line. In our

MANAGING MEMORY IN THE QUICKDRAW GX HEAP

The graphics and layout portions of QuickDraw GX use their own heap format and their own relocating Memory Manager to improve efficiency. Why is this necessary? The GX system is object based, so it needs the ability to quickly move graphics and layout objects from the heap onto disk when they aren't needed and additional memory is required, and to move these objects back into memory when they're needed. This isn't possible with the current Memory Manager.

Your application has quite a bit of control over the objects it creates in the QuickDraw GX heap. The QuickDraw GX API provides calls to unload objects from the heap, and shape attributes that indicate when a shape should be paged out of the heap to the backing store on disk. If you don't unload the objects or set any of the shape attributes yourself, QuickDraw GX unloads the oldest objects first.

You can use a GXUnload call to tell QuickDraw GX to page a graphics or layout object — specifically, a shape, style, transform, ink, color profile, color set, or tag — from the heap to the backing store on disk when the next memory management call occurs. For example, to unload a particular shape from the heap, you would use the following call:

```
GXUnloadShape(gxShape target);
```

QuickDraw GX maintains a reference to the objects, so it can read them from disk back into the heap if your application needs them. It does this automatically, but if

you want to improve the performance of your application, you can explicitly load graphics and layout objects with a GXLoad call. For example, to load a particular shape into the heap, you would use the following call:

```
GXLoadShape(gxShape target);
```

There are two shape attributes that you can set to indicate when to page a shape out to disk when memory is needed:

- When `gxDiskShape` is set, this shape will be the first shape to be paged out of memory to disk when memory is needed.
- When `gxMemoryShape` is set, this shape will be the last to be paged out to disk.

It's generally a good idea to dispose of an object as soon as you've finished with it. If you leave a bunch of unused objects lying around in the heap, QuickDraw GX will maintain them and page them out to disk when memory becomes tight. This wasted operation will cause your application to take a performance hit.

If you have a collection of shapes, say a picture, that you won't need for a while, you should consider flattening them to disk and then disposing of them. When you flatten a shape to disk, each object is compressed and sent to the file as a stream of data. When you dispose of the objects you just flattened, you free up some heap space.

QuickDraw GX shell we enable all three with the routines `SetGraphicsLibraryErrors` (which enables errors and warnings) and `SetGraphicsLibraryNotices` (which enables notices).

The debugging version of QuickDraw GX graphics and layout provides extensive validation facilities that let you determine whether you're passing valid data to the QuickDraw GX API and whether anything's gone awry internally. (See "Tracking Down a Memory Trashing Problem" for details on how to use validation to find out

how damage is being caused in the QuickDraw GX heap.) In general, you should always run with `gxPublicValidation + gxTypeValidation` while you're developing a QuickDraw GX application. To set this level of validation, make this call:

```
GXSetValidation(gxPublicValidation + gxTypeValidation);
```

This setting ensures that for all calls, QuickDraw GX checks the routines on entry and makes sure that the types are correct. For example, if you call `GXDrawShape` with this validation setting, QuickDraw GX makes sure that the shape being passed in is a valid shape and that it's an object of the correct type. This will result in a slight decrease in performance but will help you catch bad data earlier.

Your application can validate various QuickDraw GX graphics and layout objects — shape, style, transform, ink, view port, view device, view group, color profile, color set, tag, or graphics client, or any combination — with a `GXValidate` call before they're passed to a routine. For example, if you wanted to validate the inks in the heap, you would make the following call:

```
GXValidateInk(gxInk);
```

If you're interested in using only these validation calls, set the validation level to `gxPublicValidation`.

TRACKING DOWN A MEMORY TRASHING PROBLEM

Validation is a good way to track down a memory trashing problem. If the QuickDraw GX heap gets trashed, you need to know if it's your application that's doing the damage or if it's QuickDraw GX. These validation levels can help you find this out:

<code>gxNoMemoryManagerValidation</code>	<code>= 0x0000</code>
<code>gxApBlockValidation</code>	<code>= 0x0100</code>
<code>gxFontBlockValidation</code>	<code>= 0x0200</code>
<code>gxApHeapValidation</code>	<code>= 0x0400</code>
<code>gxFontHeapValidation</code>	<code>= 0x0800</code>
<code>gxCheckApHeapValidation</code>	<code>= 0x1000</code>
<code>gxCheckFontHeapValidation</code>	<code>= 0x2000</code>

To see if the problem is caused by your application, add `gxApHeapValidation` or `gxFontHeapValidation` to the

types of validation you set. The one you choose depends on whether you believe the problem lies in a general API call or within the font heap (the heap separate from the QuickDraw GX heap where all the font information is cached). This type of validation checks the public API calls when they make a memory request.

If your application doesn't trash memory after adding this validation, it means that the problem is probably caused by an internal call within QuickDraw GX. In that case, add the validation level `gxCheckApHeapValidation` or `gxCheckFontHeapValidation`. This additional validation checks the memory allocation for the internal core calls. If you die at this point, you know that the internal call is causing the problem.

SETTING UP THE COMMON COLOR LIBRARY

The common color library provided by QuickDraw GX gives you a quick way to color the shapes you create. This library contains 103 common colors, including maroon, teal, fluorescent blue, apple green, Mars orange, and Venetian red. To initialize this library, make this call:

```
InitCommonColors();
```

With the color library set up, you can call `SetShapeCommonColor` to color a shape. We'll do this later with the shapes we draw in the QuickDraw GX shell window.

USING WINDOWS WITH QUICKDRAW GX

Now that you've initialized QuickDraw GX, you need to set it up to work within a window. The first step is to create a view port and attach it to a window. Then you need to provide routines to zoom and resize the view port and to scroll its contents.

Recall that QuickDraw GX draws all shapes in a view port, which contains the mapping used to convert from the view port's local space to the global space (described by a view group). The view group determines whether the contents of the view port are drawn on the screen in a window or to an off-screen area. A view port also contains its clip shape, describing the area in which drawing can take place.

ATTACHING A VIEW PORT TO A WINDOW

By attaching a view port to a window, you guarantee that all the shapes you draw to the view port will be drawn in the correct location even after the user moves the window. You attach a view port to your window with this call:

```
gxViewPort  windowParentViewPort;  
  
windowParentViewPort = GXNewWindowViewPort(theWindow);
```

If all goes well, `windowParentViewPort` contains a reference to the view port attached to the window (the parent view port). You can't change the clip shape or mapping of this particular view port, because QuickDraw GX automatically maintains all the characteristics of the parent view port. This is a problem if you don't want your scroll bars to be overwritten. However, you can attach a view port to the parent view port (see Figure 3) and then adjust this newest view port (the child view port) to reflect changes due to scrolling or resizing the window.

Defining the new view port's bounds. Before we create a new view port we need to determine its bounds. We can find the bounds of the window by converting the window's QuickDraw `portRect` into a fixed-point QuickDraw GX rectangle. The following utility routine does this.

If you're looking for a shortcut, you can skip the `InitCommonColors` call, because your first call to `SetShapeCommonColor` will initialize the library. •

The number of view ports that can be attached to another view port is limited only by the amount of memory you make available in the QuickDraw GX heap. •

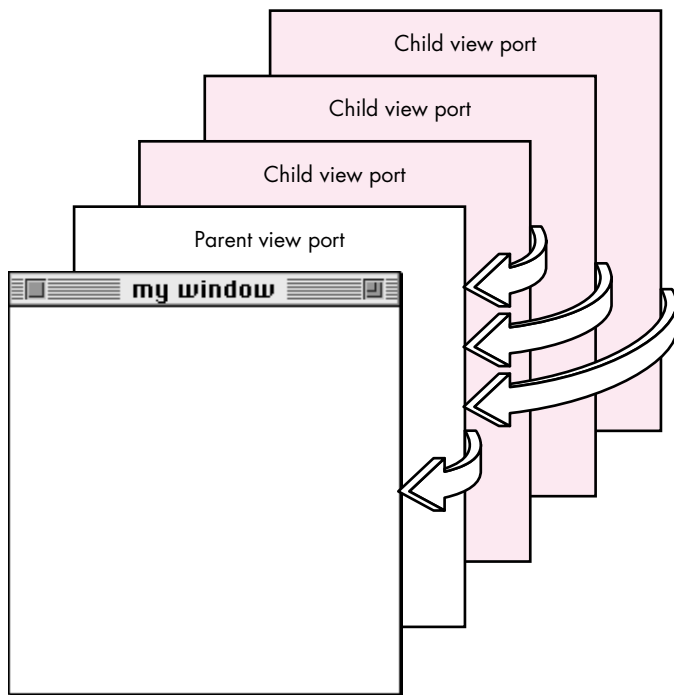


Figure 3
The View Port Hierarchy

```
void GetFixedWindowBounds(WindowPtr myWindow, gxRectangle *boundingBoxPtr)
{
    GrafPtr      oldPort;
    Rect          qdBounds;
    gxRectangle    gxBounds;

    GetPort(&oldPort);
    SetPort(myWindow);
    qdBounds = myWindow->portRect;

    // Convert the QuickDraw rectangle into a GX fixed-point rectangle.
    GXQDGlobalToFixedLocal((Point *) &qdBounds.top,
                           (gxPoint *) &gxBounds.left);
    GXQDGlobalToFixedLocal((Point *) &qdBounds.bottom,
                           (gxPoint *) &gxBounds.right);

    *boundingBoxPtr = gxBounds;
    SetPort(oldPort);
}
```

Now we can create a QuickDraw GX rectangle that represents the portRect of the window. The rectangle will reside in the variable viewRect.

```
GetFixedWindowBounds(theWindow, &viewRect);
```

We need to adjust viewRect for the scroll bars attached to the window.

```
viewRect.right -= ff(kScrollBarWidth - 1);  
viewRect.bottom -= ff(kScrollBarWidth - 1);
```

Creating and activating the child view port. Now we're ready to create a child view port that's attached to the parent view port. We want to create a new view port within the same view group (that is, sharing the same global space) as the window's view port:

```
gxViewPort gcontentViewPort;  
  
gcontentViewPort =  
    GXNewViewPort(GXGetViewPortViewGroup(windowParentViewPort));
```

The gcontentViewPort variable now contains a valid view port that we can work with. Now we're ready to set the clip shape of this view port. The clip shape can be any geometry-based shape like a rectangle, a polygon, or a path. The clip shape for gcontentViewPort will simply be defined by the rectangle contained in viewRect, which is the portRect of the window, minus the scroll bar areas. After we set the clip shape, we dispose of the shape because it's no longer needed, thereby freeing up space within the QuickDraw GX heap:

```
gxRectangle contentViewPortShape;  
  
contentViewPortShape = GXNewRectangle(&viewRect);  
GXSetViewPortClip(gcontentViewPort, contentViewPortShape);  
GXDisposeShape(contentViewPortShape);
```

Next, we need to set the mapping of gcontentViewPort to be the default mapping and attach gcontentViewPort to the parent view port:

```
GXSetViewPortMapping(gcontentViewPort, nil);  
GXSetViewPortParent(gcontentViewPort, windowParentViewPort);
```

Now we need to tell QuickDraw GX which view port we want the shapes to be drawn in. We could have all shapes drawn to both view ports, but that wouldn't be especially efficient. So we make the following call, which tells QuickDraw GX to draw all the shapes we make from now on in gcontentViewPort:

QuickDraw GX uses fixed-point

numbers, offering the advantage of speed, instead of the floating-point numbers used by QuickDraw. The **ff** macro uses IntToFixed to convert an integer into a fixed-point number; another handy macro, **fl**, converts a floating-point number into a fixed-point number. •

```
SetDefaultViewPort(gcontentViewPort);
```

ZOOMING AND RESIZING THE CHILD VIEW PORT

Anytime the user zooms or resizes the window, we must update the clip shape of the child view port we attached to the window's parent view port. To do this, we get the portRect of the window in a fixed-point rectangle, adjust it for the scroll bars, create a new clip shape from this rectangle, and reset the clip shape of the view port to this new clip shape. The following routine does the work:

```
void ResetContentViewPortClip(WindowPtr theWindow)
{
    gxRectangle    viewRect;
    gxShape        contentViewPortClipShape;

    // Get the window's portRect into the fixed-point viewRect.
    GetFixedWindowBounds(theWindow, &viewRect);

    // Adjust the viewRect to accommodate the scroll bars.
    viewRect.right -= ff(kScrollBarWidth - 1);
    viewRect.bottom -= ff(kScrollBarWidth - 1);

    // Create and set the new clip shape.
    contentViewPortClipShape = GXNewRectangle(&viewRect);
    GXSetViewPortClip(gcontentViewPort, contentViewPortClipShape);
    GXDisposeShape(contentViewPortClipShape);
}
```

SCROLLING THE CHILD VIEW PORT'S CONTENTS

When the user scrolls in the window, we need to reset the mapping of the child view port before we call ScrollRect to scroll the bits in the window. That will ensure that when we redraw the contents of the window on the next update event, all the shapes will be located in the correct window-relative position. Otherwise they would be redrawn in their old position, because the geometry of the shapes — which includes their position in the view port — doesn't change.

This remapping approach gives us an advantage at print time. If we didn't adjust the mapping of the child view port, we would need to adjust the mapping of each shape. While it's possible to do that, we would have to do it again at print time to ensure that the shapes printed on the right page. If the page contained a lot of shapes, that could be a very time-consuming operation.

To update the mapping of gcontentViewPort (the child view port) to reflect the scrolling of the window, we get its current mapping, adjust it to translate the view port by the scroll amount, and set the mapping to the changed one.


```

gxMapping          viewportMapping;

GXGetViewportMapping(gcontentViewport, &viewportMapping);
GXMoveMapping(&viewportMapping, ff(hScroll), ff(vScroll));
GXSetViewportMapping(gcontentViewport, &viewportMapping);

```

CREATING, MANIPULATING, AND DRAWING QUICKDRAW GX SHAPES

At this point, you’ve learned how to initialize QuickDraw GX and deal with view ports. It’s finally time to talk about creating, manipulating, and drawing shapes.

A shape contains all the information required to draw it. To create a shape with QuickDraw GX, you simply define its geometry. Then you can draw it by calling `GXDrawShape(myShape)`. If you haven’t specified otherwise, your shape will use the default style, transform, and ink supplied by QuickDraw GX for the particular type of shape. When you change a shape’s style, transform, or ink, QuickDraw GX copies a reference to the new style, transform, or ink into your shape.

To illustrate the process of creating, manipulating, and drawing shapes, in our QuickDraw GX shell we’ll create a typographic shape containing text. We’ll outline the text in some color and fill the inside of each letter with a pattern composed of stars. Then we’ll create a typographic shape containing a line layout of some text, which we’ll render in a combination of different fonts and scripts.

In the QuickDraw GX shell we’ll use a picture, which we’ll store in the global variable `gthePage`, to collect all the shapes we draw to the window. Using a picture enables us to make just one call to `GXDrawShape` to draw the contents of the window. We also need to set the `gxUniqueItemsShape` shape attribute so that each time we add a shape to the picture, QuickDraw GX will make a copy of the shape and add the copy, rather than just adding a reference to the shape. These calls create our picture and set the shape attribute:

```

gxShape  gthePage

gthePage = GXNewShape(gxPictureType);
GXSetShapeAttributes(gthePage, gxUniqueItemsShape);

```

The variable `gthePage` now holds an empty picture, ready to have shapes added to it.

EXAMPLE 1: A SHAPE CONTAINING TEXT

First we’ll create a shape containing text, which we’ll store in the variable `tempTextShape`. We want the text to read “GX.” We set the text shape’s position, create the new shape, set the text size, and set the font to New York:

```

gxPoint    textPosition = {ff(10), ff(205)};
gxShape    tempTextShape;

tempTextShape = GXNewText (2, (unsigned char *) "GX", &textPosition);
GXSetShapeTextSize(tempTextShape, ff(250));

// This next call comes from the Font Library.
SetShapeCommonFont(tempTextShape, newyorkFont);

```

The variable tempTextShape now holds the text shape.

Outlining the text. We said that we want to outline the text with some color and fill the text shape with stars. The approach we'll take to outlining our text shape is to first convert it into a path shape, which requires only the GXSetShapeType call.

After the conversion to a path shape, each character in the text shape becomes a path. Thus, the converted shape will contain two different paths, one for each character. To draw the outline of each path, we set the fill type to gxClosedFrameFill. Then we set the pen to draw on the outside of each contour, set the pen thickness, set the color of our path to a color from the common color library, and scale it 125% on the x-axis and 65% on the y-axis so that it will come out looking short and fat.

```

GXSetShapeType(tempTextShape, gxPathType);
GXSetShapeFill(tempTextShape, gxClosedFrameFill);
GXSetShapeStyleAttributes(tempTextShape, gxOutsideFrameStyle);
GXSetShapePen(tempTextShape, ff(3));
SetShapeCommonColor(tempTextShape, blue);
GXScaleShape(tempTextShape, fl(1.25), fl(0.65), 0, 0);

```

Now we add our path shape to the picture we've stored in gthePage:

```

GXSetPictureParts(gthePage, 0, 0, 1, &tempTextShape, nil, nil, nil);

```

From now on, whenever we draw gthePage, our path shape stored in tempTextShape will be drawn as well. (See Figure 4.)

Filling with stars. To fill our shape with stars, we start by changing the fill type and color of our path. Then we define the star shape and the pattern record that will replace the style of our shape, add our new patterned path to our picture, and dispose of all the unneeded shapes.

We need to change the fill type of tempTextShape to solid fill because at this point we want to fill the contents rather than draw the outline of each path. We also want our stars to be gray, so we need to reset the color.

Another approach to outlining text,
described in the article "QuickDraw GX for
PostScript Programmers" in this issue, involves
using text faces. •

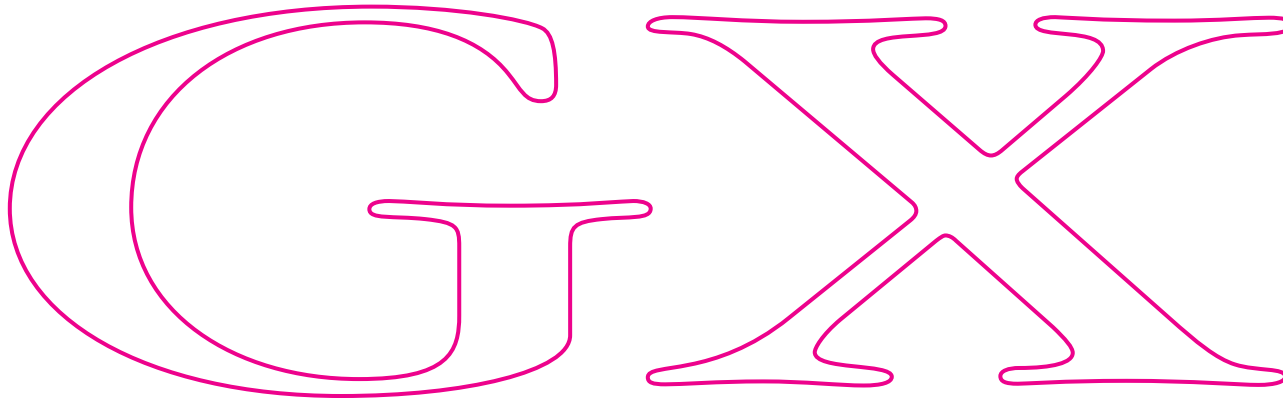


Figure 4
The Path Shape Outlining Our Text

```
GXSetShapeFill(tempTextShape, gxSolidFill);
SetShapeCommonColor(tempTextShape, cold_grey);
```

We define the star as a polygon shape containing one contour and five points, with the default fill of evenOdd (which results in a star with a hollow pentagon inside), and we scale it by 15% to make it tiny:

```
gxShape starShape;
long starGeometry[] = {1, // One contour.
                       5, // Five points defining the polygon.
                       ff(60), 0, ff(90), ff(90), 0, ff(30),
                       ff(120), ff(30), 0, ff(90)};

starShape = GXNewPolygons((gxPolygons *) starGeometry);
GXScaleShape(starShape, fl(0.15), fl(0.15), 0, 0);
```

The pattern record contains the shape to be used as the pattern, two vectors (u and v) that describe how to tile the pattern, and the pattern attribute. (See Figure 5.) We'll attach this record to the style of our path shape, replacing the default style.

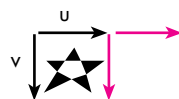


Figure 5
Making the Star Pattern

The bounding box of our star shape will be a fixed-point rectangle contained in the variable `starShapeBounds`. This information is used to define the `u` and `v` vectors of our pattern record.

```
gxRectangle      starShapeBounds;

GXGetShapeBounds(starShape, 0L, &starShapeBounds);
```

We define `u` and `v` to place the stars side by side without overlapping:

```
gxPatternRecord  starPattern;

starPattern.u.x = 0;
starPattern.u.y = starShapeBounds.bottom;
starPattern.v.x = starShapeBounds.right + fix1;
starPattern.v.y = 0;
```

We set the attributes of the pattern record to `gxNoAttributes` and then add our star polygon shape to the pattern record:

```
starPattern.attributes = gxNoAttributes;
starPattern.pattern = starShape;
```

Finally, we add the `starPattern` to `tempTextShape`. QuickDraw GX copies a new style to the converted `tempTextShape` with the pattern record, replacing the default style currently referenced by `tempTextShape`. Anytime this shape is drawn, it will be drawn patterned with stars. We add our updated `tempTextShape` to the picture stored in the variable `gthePage`:

```
GXSetShapePattern(tempTextShape, &starPattern);
GXSetPictureParts(gthePage, 0, 0, 1, &tempTextShape, nil, nil, nil);
```

Now we can dispose of our star polygon shape because it's contained in our pattern record, which has been encapsulated into the style of our shape. We also dispose of `tempTextShape` because there's now a unique reference to it in our picture.

```
GXDisposeShape(starShape);
GXDisposeShape(tempTextShape);
```

The results. Our picture, `gthePage`, now contains two shapes. The first shape is a colored path that outlines the text “GX.” The second shape is the same path filled with gray stars. When we call `GXDrawShape(gthePage)`, all the shapes contained in our picture are drawn. The results are shown in Figure 6.

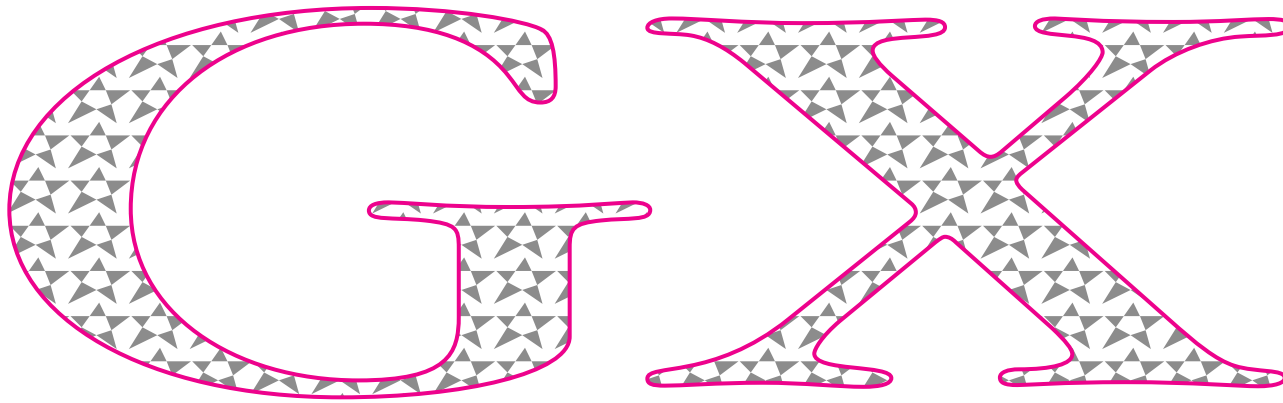


Figure 6
Our Text Outlined With Color and Filled With Stars

EXAMPLE 2: A SHAPE CONTAINING A LINE LAYOUT

In a text shape, all the text can be rendered in only one font and size. In a layout shape, on the other hand, the text can be rendered in a combination of different fonts and sizes by applying multiple style runs. A layout shape can contain as many style runs as you like. A style controls the font, the size, the script (such as Roman or Arabic), run features (which pertain to swashes, ligatures, and final forms available with the specified font), justification overrides, and kerning adjustments.

To illustrate QuickDraw GX's line layout features, we'll create a layout shape containing the text "Catch the Nasty WAVE, Dude." We'll start by creating the styles to be applied in three different style runs, and then we'll define the layout shape. Finally, we'll make the whole line of text slant backward, to demonstrate that you can perform any graphical operation on typographic shapes in QuickDraw GX.

Creating the styles. We're going to apply styles to our layout shape in three different runs. The first run of text, "Catch the Nasty," will use the Hoefler Italic font; the second run, "WAVE," will use the Times Roman font; and the third run, "Dude," will use Hoefler Italic again. In the first and third runs, we'll enable a total of three of the run features available with Hoefler Italic. But before we create the styles, we need to initialize the run controls used to regulate the run features. We specify the number of run features and styles we'll use.

```
gxRunControls    runControls;  
gxRunFeature     runFeatures[3];  
gxStyle         styles[3];  
  
InitializeRunControls(&runControls);
```

Here's how we specify the style used for our first text run:

```
styles[0] = NewLayoutStyle((char *)"\pHoefler Italic", ff(36), 0,  
                           &runControls, nil, 0, nil);
```

We want to enable two run features in our first text run: an “as” ligature in “Nasty” and swashes on the “C” and “N.”

```
runFeatures[0].featureType = gxLigatureType;  
runFeatures[0].featureSelector = gxLigatureRareOnSelector;  
runFeatures[1].featureType = gxAlternateDesignsType;  
runFeatures[1].featureSelector = gxAlternateDesignsChanceryOnSelector;
```

Now we add the two run features to the style used by this run of text:

```
GXSetStyleRunFeatures(styles[0], 2, runFeatures);
```

We create the styles for the second and third text runs:

```
styles[1] = NewLayoutStyle((char *)"\pTimes Roman", ff(38), 0,  
                           &runControls, nil, 0, nil);  
styles[2] = NewLayoutStyle((char *)"\pHoefler Italic", ff(40), 0,  
                           &runControls, nil, 0, nil);
```

The run feature used by the last run of text will enable the final form of the “e” in “Dude” and will update the style:

```
runFeatures[2].featureType = gxSmartSwashType;  
runFeatures[2].featureSelector = gxSmartSwashLineFinalsOnSelector;  
GXSetStyleRunFeatures(styles[2], 3, runFeatures);
```

We'll also get the swash on “D” in “Dude” because we enabled the swash capabilities in the second run feature. When we call `GXSetStyleRunFeatures`, we tell QuickDraw GX line layout to use all three entries in the array.

Defining the layout shape. We need to define the length of our layout (in bytes) and initialize the lengths array used to tell QuickDraw GX line layout the length of each run of text:

```
short    totalLengthOfLayout, lengthsArray[3];  
  
lengthsArray[0] = 15; lengthsArray[1] = 6; lengthsArray[2] = 5;  
totalLengthOfLayout = 26;
```

We now have all the information required to define our layout shape and add it to our picture stored in the variable `gthePage`. Our layout shape will contain three runs of text using three different styles. Each style will use a different text size. Two different fonts will be used, and three run features will be enabled. The definition looks like this:

```
char      *sampleText = "Catch the Nasty WAVE, Dude";
gxPoint    layoutPosition = {ff(10), ff(65)};

tempLayoutShape =
    GXNewLayout(1, &totalLengthOfLayout, (void *) &sampleText,
                3, lengthsArray, styles,
                0, nil, nil, nil, &layoutPosition);

GXSetPictureParts(gthePage, 0, 0, 1, &tempLayoutShape, nil, nil, nil);
```

We draw `tempLayoutShape` by calling `GXDrawShape(tempLayoutShape)`. The result is shown in Figure 7. Note that “WAVE” has been kerned automatically.



Figure 7
Our Line Layout Shape

Skewing the shape and cleaning up. Because QuickDraw GX treats all typographic shapes as graphics, we can perform any graphical operation on our layout shape, now stored in `tempLayoutShape`. We’ll skew it by 125% in the x direction about its origin, move the shape 15 pixels in the x direction to put it back into the window after the skew, and move it 75 pixels in the y direction to move it away from the beginning position. Finally, we’ll add the shape to our picture stored in the variable `gthePage`. When drawn, it will look like Figure 8.

```
GXSkewShape(tempLayoutShape, fl(1.25), 0, shapePosition.x,
            shapePosition.y);
GXMoveShape(tempLayoutShape, ff(15), ff(75));
GXSetPictureParts(gthePage, 0, 0, 1, &tempLayoutShape, nil, nil, nil);
```

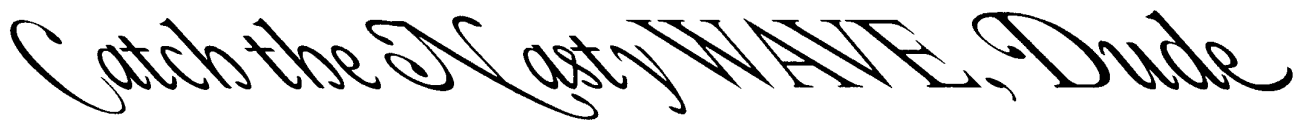


Figure 8

Our Line Layout Shape, Skewed

We no longer need tempLayoutShape because it's now part of our picture, so we can dispose of tempLayoutShape and our array of styles:

```
short    loop;

GXDisposeShape(tempLayoutShape);
for (loop = 0; loop <= 2; loop++)
    GXDisposeStyle(styles[loop]);
```

Now gthePage contains three shapes. The first shape is a colored path that outlines the text "GX," the second shape is the same path filled with stars, and the third shape is a typographically cool and skewed rendition of "Catch the Nasty WAVE, Dude." When we call GXDrawShape(gthePage), all these shapes are drawn. In the QuickDraw GX shell, gthePage is set up in the CreateThePageOfGXShapes function.

BASIC PRINTING IN QUICKDRAW GX

QuickDraw GX adds quite a few new printing features, as mentioned earlier in this article. We'll briefly explore printing here by discussing methods for printing shapes, how to initialize QuickDraw GX printing, how to handle printing errors, how to implement three new print items added to the File menu, and how to finish up printing.

METHODS FOR PRINTING SHAPES

Three approaches to printing shapes are available to your application: you can send shapes to QuickDraw GX printing one by one, you can send a picture that contains all the shapes used to represent a page, or you can send a picture that contains other pictures.

Depending on your application, sending an entire picture full of information may be simplest, but if the picture contains quite a few shapes, say 20,000 to 25,000, you might suffer a performance penalty. On the other hand, sending shapes one by one is more complicated for some applications, but it may be the most efficient in terms of performance when you have lots of shapes — more than 20,000. If you have lots of shapes in one big picture but you don't want to send them one by one, you should

consider breaking the big picture into smaller pictures. (Recall that a QuickDraw GX picture is a shape that can contain other pictures.) This approach would enable you to group similar items in the same way as in MacDraw®.

You need to decide which method works best for your application. In the case of our shell, we'll send a picture for each page of our document (which conveniently consists of only one page).

INITIALIZING QUICKDRAW GX PRINTING

To initialize QuickDraw GX printing, we call `InitPrinting` after we create the graphics client. We then create a job, which is an extensible data structure containing all the information required to print a document. For example, a job specifies the formatting printer driver, the printer for which the job has been targeted, when the document should be printed, and the number of pages in the document.

```
OSErr      myQDGXPrintError;
gxJob      gDocumentJob;

myQDGXPrintError = GXInitPrinting();

if (myQDGXPrintError == noErr)
    myQDGXPrintError = GXNewJob(&gDocumentJob);
else
    // Your error-handling code here!
```

HANDLING PRINTING ERRORS

QuickDraw GX printing handles printing errors differently from the Printing Manager. The Printing Manager requires you to check for an error after each call. If you receive an error, you must match all your open calls with the appropriate close calls, close up the Printing Manager, and report the error to the user.

In QuickDraw GX printing, printing errors are local to the job. This gives you the ability to poll for errors after a group of routines, making your code smaller and cleaner. If you do receive an error within a group of routines, the routine that caused the error will prevent the remaining calls from executing until the error is cleared by calling `GXGetJobError`. You can then alert the user to the problem. All the QuickDraw GX printing errors are listed in `PrintingErrors.h`.

IMPLEMENTING THE NEW MENU ITEMS

Your application should implement three new items in the File menu: Print One Copy, Document Setup, and By Page Setup.

- Print One Copy should print one copy of the document without presenting the user with any dialog other than a status dialog.

- Document Setup should present a dialog that lets the user format the entire document, similar to the old Page Setup dialog.
- By Page Setup should present a dialog that enables page-by-page formatting of a document (for instance, the user should be able to choose a different page size for each page in the document).

We won't take the space to reprint the code to implement Document Setup or By Page Setup here; check it out in our QuickDraw GX shell.

When the user chooses Print One Copy from the File menu, we want to print the application's document. In our QuickDraw GX shell, that means printing the contents of `gthePage`:

```
OSErr DoPrintOneCopy(WindowPtr window)
{
    Str255      windowTitle;
    OSErr       printError;

    if (window)
    {
        GetWTitle(window, windowTitle);

        // Start sending the job. The job has the same name as our window,
        // and it contains one page. The name appears in the status dialog.
        GXStartJob(gDocumentJob, windowTitle, 1);

        // Send the entire page down to the printer. (All the shapes that
        // are being printed have been collected into gthePage.)
        GXPrintPage(gDocumentJob, 1, GXGetJobFormat(gDocumentJob, 1),
                    gthePage);

        // This call tells QuickDraw GX printing we've finished sending the
        // job, so terminate the transmission (that is, the connection to
        // the printer).
        GXFinishJob(gDocumentJob);
        if (GXGetJobError(gDocumentJob) != noErr)
            // Your error-handling code here!
    }
}
```

FINISHING UP

Once we've finished printing, we leave the printing system open. This isn't a problem, because the QuickDraw GX printing system has a very small memory requirement when not in use — approximately 35K. The main reason we leave it

open is that we want to keep the job around, ready to be used the next time the user prints. If we were to close the printing system, we would need to recreate the job.

CLOSING UP THE QUICKDRAW GX WORLD

Closing up the QuickDraw GX world is as straightforward as initializing it. You should dispose of all the QuickDraw GX objects you created while your application was running. For example, if you called the `GXNewShape` routine to create a shape, you should call the `GXDisposeShape` routine to dispose of it.

In our QuickDraw GX shell's shutdown, we need to dispose of the QuickDraw GX picture we created to contain all the shapes drawn to our window, the common color library, our window, and the graphics client. To dispose of the picture we created, we first test to make sure it contains something. If it does, we call `GXDisposeShape` on it:

```
if (gthePage != nil)
    GXDisposeShape(gthePage);
```

The QuickDraw GX shell has also been using the common color library, which we dispose of by calling

```
DisposeCommonColors();
```

Now we can dispose of the window; this also disposes of the view ports. In our shell, we created our own pointer to the window record, so we need to dispose of it with the appropriate call.

To close up the QuickDraw GX printing system, we dispose of the job and then call `GXExitPrinting` before we dispose of the graphics client. If we were going to save our picture to disk, we would want to flatten the job to disk as well.

```
if (GXDisposeJob(gDocumentJob))
    // Your error-handling code here!

if (GXExitPrinting())
    // Your error-handling code here!
```

Finally, we need to call `GXDisposeGraphicsClient` to dispose of the graphics client we created earlier:

```
GXDisposeGraphicsClient(gGraphicsClient);
```

If for some reason you haven't yet disposed of all the QuickDraw GX objects you've created while your application has been running, `GXDisposeGraphicsClient` disposes

of them. If you enabled graphics notices earlier in your application, you'll receive a graphics notice for the first QuickDraw GX object you didn't dispose of. For example, if you didn't dispose of a shape, you'll receive this notice:

GRAPHICS NOTICE: shape not disposed

This isn't a problem when you're shutting down your application, because the QuickDraw GX heap is cleaned up when `GXDisposeGraphicsClient` is called. But getting a notice like this when your application ends means that you've been leaving unused items in the QuickDraw GX heap. That could mean reduced performance, since the QuickDraw GX Memory Manager has to page these unused objects into and out of the heap in a low-memory situation.

If you do receive a graphics notice about something you didn't dispose of, you should track down the object to improve your runtime memory management. Prevent your application from calling `GXDisposeGraphicsClient` so that the QuickDraw GX heap doesn't disappear before you can analyze it. In `GraphicsBug`, use the Heap menu to choose your application's heap; then type "hd" to get a dump of your QuickDraw GX heap, or "hd shape" to get a list of all the shapes you forgot to throw away.

In our QuickDraw GX shell, the routine `ShutDownProgram` closes up the QuickDraw GX world.

WHERE TO GO FROM HERE

We've taken a look at what you need to do to begin making use of the new QuickDraw GX technology. Now you're ready to tackle that huge pile of QuickDraw GX material.

I suggest starting with the sample code that comes with QuickDraw GX. There are sample applications ranging from very simple to relatively complex, and some tools to play with. Try out the samples that interest you. For an illustration of the minimum number of lines of code needed to draw GX shapes, take a look at the "One Rectangle" sample. It initializes the QuickDraw GX world, attaches a view port to a window, creates a rectangle, and draws the shape to the window.

Then take a look at the QuickDraw GX documentation. You should start with the "QuickDraw GX Objects" chapter, which describes all the objects illustrated in Figure 2. From there, proceed to the parts of the documentation that make the most sense for your application.

I hope you've enjoyed this overview of QuickDraw GX. We haven't covered all the exciting parts of the technology, by any means. This is only the beginning!

THANKS TO OUR TECHNICAL REVIEWERS

Hugo Ayala, Cary Clark, Tom Dowdy, Dave
Hersey, Daniel Lipton, Dave Opstad •

DEVELOPING QUICKDRAW GX PRINTING EXTENSIONS

With QuickDraw GX comes a new extensible printing architecture that simply invites you to jump in and tinker. Writing printer drivers has never been easier. But with the advent of printing extensions, you may never have to write a printer driver again! This article tells you what you need to know to create QuickDraw GX printing extensions.



SAM WEISS

Macintosh system software has long provided hooks that enable developers to add system-level features in the form of INITs (now called system extensions) and cdevs (control panels). QuickDraw GX extends the extension concept to printing via *printing extensions*, plug-in software modules whose sole purpose in life is to modify the behavior of printing. Want to stamp the word “Confidential” on every page printed to a specific printer? Write a printing extension. Want to drive a sheet feeder attachment to a LaserWriter? Write a printing extension. Chances are that if there’s something you want to do that the QuickDraw GX printing architecture doesn’t already do, you can do it with a printing extension.

In this article, I’ll describe the steps you must take to create a QuickDraw GX printing extension. By the end of the article we’ll have a working, environmentally friendly extension called 4-Up. As shown in Figure 1, the 4-Up printing extension maps four document pages onto one physical page at print time, saving countless trees in the process!

Notice that I referred to a printing extension as a “software module.” I would love to use the currently hip term “component,” but I don’t want to confuse the issue by implying that printing extensions are implemented as Component Manager components. In fact, printing extensions are a whole new sort of beast. QuickDraw GX includes a new low-level software manager called the Message Manager, which it uses to implement its extensible printing architecture. A printing extension is a client of the Message Manager called a *message handler*. The Message Manager manages the construction of a hierarchy of related message handlers into a *message class*, allows *message objects* to be instantiated from message classes, and directs messages among message handlers. I won’t be going into too much detail about the Message Manager

SAM WEISS (AppleLink S.WEISS), famous for the PrintSpy INIT (which surfaced in the former Soviet Union when the KGB was dismantled), has been working on QuickDraw GX printing in one form or another for the last five years. In this capacity, he can often be found in his office managing collections, sending messages, rewriting the core, or cutting his own hair. But in his spare time, Sam enjoys sharing his favorite

hobby (musical electronic fishkeeping) with his one-year-old daughter Talia, whose first word was DoubleDespoolDatFileDaDaPlecostomous. When asked about life after QuickDraw GX, Sam appeared confused and muttered something about having a few philosophers over for dinner. •

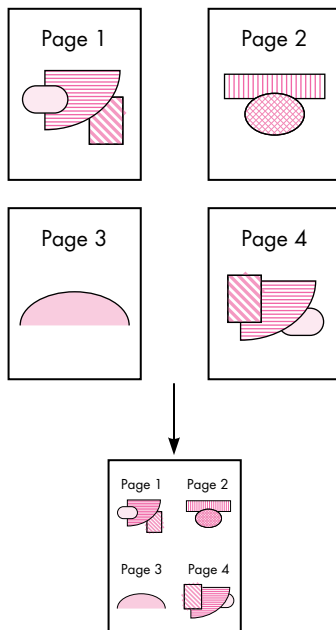


Figure 1
4-Up at Work

in this article. But there are times when you'll have to call the Message Manager directly from your printing extension, so you need to be aware of it.

Another new manager included in QuickDraw GX is the Collection Manager, which manages the creation of lightweight extensible data structures called *collections*. QuickDraw GX printing makes heavy use of collections. Although we won't need to call the Collection Manager from the 4-Up extension, nearly all serious printing extensions will need to do so.

THE QUICKDRAW GX PRINTING ARCHITECTURE

Before we jump into the technical details of writing a printing extension, it will be helpful to have a general overview of the QuickDraw GX printing process. First, I'll describe how the printing process occurs in several distinct phases, each responsible for a specific subtask of the entire process. Then, I'll explain the flow of control under QuickDraw GX printing, as contrasted with that of the Printing Manager. And finally, I'll discuss how printing extensions fit into the picture.

THE FOUR SEASONS OF PRINTING

Under QuickDraw GX, printing occurs in four distinct phases:

- *Spooling*, which takes place in the foreground under the control of the printing application, results in a device-independent print file being stored to disk. This print file is also known as a *portable digital document* or PDD, and it can be viewed using the version of TeachText included with QuickDraw GX. Because the contents of the print file are stored in a device-independent format, it can be redirected to a device other than the original target, potentially even a different class of device, for actual printing. For example, a print file originally targeted for a PostScript LaserWriter can be redirected to an ImageWriter. The print file contains enough information to allow the document to be rendered to the best of the printer's ability, regardless of the originally intended target printer.
- *Despooling*, which always occurs in the background under the control of the PrinterShare GX background printing task, is the process of reading pages from the print file for imaging. Despooling need not occur on the same machine on which spooling occurred. PrinterShare GX sends print files over the network when the target device is a remote server. Also, users may copy print files to floppy disk, or otherwise manually move them to other machines for printing.
- *Imaging* also occurs in the background under the control of PrinterShare GX. Despooling and imaging always happen together, but it's useful to consider them as distinct phases since they accomplish different tasks. While the despooling process is responsible for reading the correct pages from the print file, the imaging process is responsible for translating the device-independent graphics data contained in the file into a format recognized by the target device. This would be PostScript code in the case of a PostScript LaserWriter, and it would be device-resolution bitmaps and appropriate escape sequences in the case of an ImageWriter.
- *Device communications* encompasses the actual dialog that occurs between the printer driver and the target device. This is a distinct phase, as it may actually take place on a machine other than that on which imaging occurred. For example, if the target device is a remote server, and PrinterShare GX determines that certain necessary resources (such as fonts) aren't available on the server, PrinterShare GX may opt to image the job locally into an *image file*, which it sends to the server. By definition, image files are device-dependent, nonviewable, and nonredirectable. The image file is "played back" on the server during the device communications phase.

GO WITH THE FLOW

At the highest level, the flow of control under QuickDraw GX printing remains similar to that in the existing Printing Manager. There are three major players: the application, the printing API, and the printer driver. The printing application calls a layer of printing API routines, each of which either performs some task or calls the appropriate printer driver to perform some task. However, there are two major differences in QuickDraw GX printing. First, the printing API has been greatly expanded. And second, the way in which control is transferred from the API to the driver is completely different. I won't be going into detail about the expanded printing API — but you'll need to understand the new flow of control in order to write printing extensions.

The existing Printing Manager calls the driver simply by loading the appropriate PDEF (code resource) from the driver, computing the offset to the specified routine, and jumping to it. As shown in Figure 2, QuickDraw GX printing uses an intermediary, the Message Manager, to transfer control to the driver. When the application makes a QuickDraw GX printing call, and the printing API needs to call the driver, it does so by calling the Message Manager to request that a message be sent to the driver. The advantage of this approach is flexibility. The Message Manager allows message handlers to be inserted between the printing API and the driver (which is itself a message handler). Aha! The light goes on! This gives us the ability to extend the behavior of printing, or even fix bugs, without modifying the driver's code. It also serves as the foundation upon which the printing extension mechanism is built.

DON'T SHOOT THE MESSENGER

QuickDraw GX printing defines over a hundred messages. When an application calls the printing API, QuickDraw GX either performs the requested task itself or sends one of these messages (via the Message Manager) to the driver to perform the task. For many tasks, QuickDraw GX provides a default implementation for the associated message, but sends a message to the driver anyway. This gives the driver a chance to do things its own way, or to massage the message parameters before forwarding the message on to the default implementation.

This is where printing extensions come in; they're inserted between the printing API and the driver, thereby having the opportunity to *override* any message before it gets to the driver. There are two flavors of message overriding, partial and complete.

In a partial message override, the extension will do some work to customize the behavior of the message, but it will still forward the message to the driver. The extension may do its work before forwarding the message (preprocessing), after forwarding the message (post-processing), or both before and after forwarding the message. For example, message preprocessing might involve changing one or more of the message parameters before the driver sees them. Post-processing might involve modifying result parameters returned by the driver.

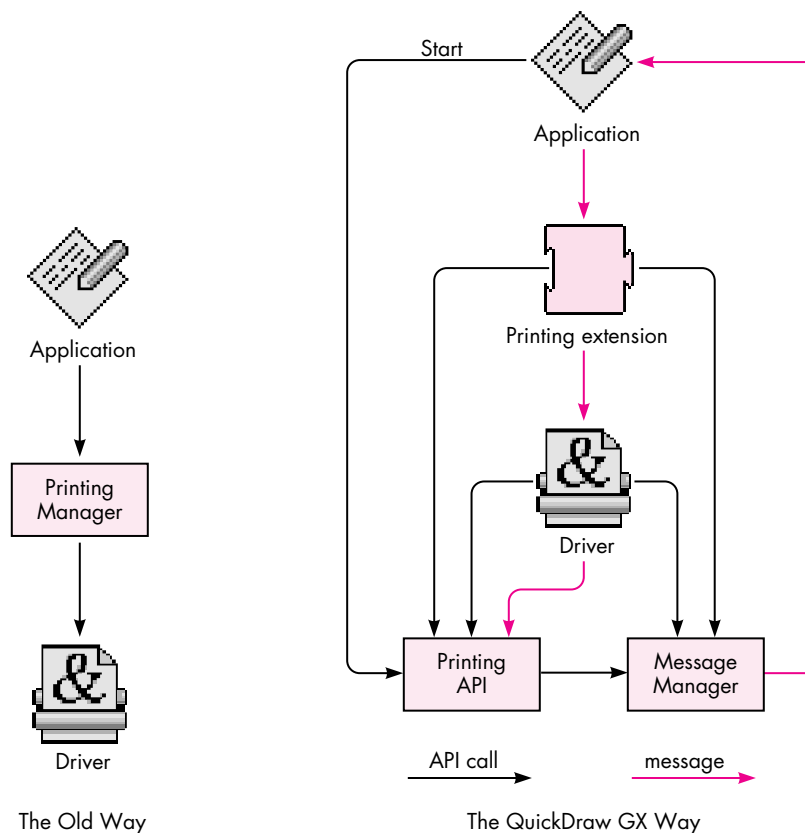


Figure 2
The Old Way Versus the QuickDraw GX Way

In a complete message override, the extension is responsible for implementing the message entirely and does not forward the message to the driver. Since any number of extensions may be inserted between the printing API and the driver, complete overrides will also inhibit any other extensions inserted between the driver and the extension performing the override from receiving the message. So before completely overriding a message, think hard and be sure to consult the documentation (*Inside Macintosh: Printing Extensions and Printer Drivers*, available from Addison-Wesley in September), which gives details regarding which messages are appropriate candidates for complete overrides.

Any message not explicitly overridden by the extension is implicitly forwarded by the Message Manager to the driver — or if there is another extension loaded before the driver, it will get the next crack at the message.

LESS IS MORE

Now that you have some background on the behind-the-scenes operations of QuickDraw GX printing, we can investigate what it takes to cram four pages into one. The first step is deciding which messages to override. With over a hundred messages to choose from, this may well be the most difficult aspect of printing extension development. If you've ever worked with a large class library like MacApp, you know what I mean. Half the battle is understanding the context within which various messages are sent. QuickDraw GX printing is no different; in fact, you can envision it as a printing class library. As in MacApp, it takes some time to learn your way around.

In our case, there are at least four different messages we could override in order to stuff four pages into one: GXSpoolPage, GXDespoolPage, GXImagePage, and GXRenderPage. The one we choose will depend on the desired effect. I chose the GXDespoolPage message, which is sent by QuickDraw GX to read each page from the print file during background printing. QuickDraw GX printing always spools data to a device-independent print file and then releases control back to the foreground application. PrinterShare GX, a background printing process, despools the print file, images it, and sends the result to the target printer. By overriding the GXDespoolPage message, we have no effect on the print file itself and we don't affect spooling performance. Our modifications will be done on the fly during the background despooling/imaging phase of printing. One implication of this strategy is that our extension won't affect what the user sees when viewing the print file on the screen (soft copy); it will affect only what's actually printed (hard copy).

During the background despooling/imaging phase of printing, QuickDraw GX sends the GXDespoolPage message for each page in the print file. QuickDraw GX supplies a default implementation for this message which reads the requested page from the print file and returns it to the caller as a picture shape. The 4-Up extension simply needs to override the GXDespoolPage message and return a picture shape containing the requested page plus the three succeeding pages. Of course, we'll need to scale down the original pages and move them around on the physical page a little. But that's the basic idea.

OK, it sounds great in theory. But we have a problem. Suppose the print file contains 16 pages. We're effectively creating a 4-page document. But QuickDraw GX is going to send the GXDespoolPage message 16 times and expect a valid picture shape to be returned each time. What happens when we run out of pages, after we're called for the fourth time? As you've probably guessed, there's another message we must override called GXCountPages. This message is sent at the beginning of the imaging process to determine how many times the GXDespoolPage message should be sent. In the example given above, we would need to override GXCountPages to return 4 instead of 16.

WRITING THE CODE

Now that we know which messages to override, we can write some code. Writing the code for a printing extension consists of implementing message overrides. In the 4-Up extension, we'll override two messages, GXCountPages and GXDespoolPage. Coding a message override is fairly straightforward, similar to coding a callback routine for other Toolbox managers. The most important part is ensuring that we declare the function with the expected calling interface.

OVERRIDING GXCOUNTPAGES

The GXCountPages message has the following interface:

```
OSErr GXCountPages (gxSpoolFile thePrintFile, long* numPages);
```

In fact, all messages share certain common interface elements. For example, *all messages must return a result of type OSErr*. This is important because all printing code is segmented. If the segment dispatcher fails to load your extension's code segment, it must be able to report the error condition to the caller. If your particular override doesn't do anything that can fail, simply return noErr.

Now that we know the interface to the GXCountPages message, we can implement our override. Since we're squeezing four pages into one, we can determine the number of actual pages with the following simple formula:

$$\text{physicalPages} = (\text{originalPages} + 3) / 4$$

Trick #1. Determining the value of originalPages is one of two tricky things we need to do in this extension. Your first thought might be to retrieve the document's page count from the *current job*, which is an implicit parameter to every message. The current job is an abstract object of type gxJob, which you can think of as a replacement for the print record in the existing Printing Manager. It contains all sorts of information relating to the document being printed. We can get a reference to the current job by calling GXGetJob, a QuickDraw GX printing routine, and then access information such as the document's page count from the job.

Although this technique would work for simple cases, it won't work if another printing extension is present and also modifying the result returned by the GXCountPages message. Consider the case where the user is running our 4-Up extension and another 2-Up extension. Ideally, the result should be an 8-to-1 mapping; both extensions would do their work, each oblivious to the other's existence, yet the final output would be the result of a cooperative effort!

The correct technique is to forward the GXCountPages message to the next message handler, and use the result we get back as the value for originalPages. Note that the value we get may actually be the result of modifications made by other extensions, but

we don't care! That's the beauty and flexibility inherent in the messaging architecture. Forwarding the GXCountPages message is as easy as calling the predefined routine Forward_GXCountPages. Here's the full implementation of our GXCountPages override:

```
OSErr FourUpCountPages (gxSpoolFile thePrintFile, long* numPages) {
    OSErr anErr;
    long originalPages;

    anErr = Forward_GXCountPages(thePrintFile, &originalPages);
    nrequire (anErr, FailedForward_GXCountPages);

    *numPages = (originalPages + 3) / 4;
    return noErr;

FailedForward_GXCountPages:
    return anErr;
}
```

Note the use of the nrequire exception-handling macro. It displays an error string and jumps to the FailedForward_GXCountPages label if anErr is nonzero. See the article “Living in an Exceptional World” in *develop* Issue 11 for more information.

OVERRIDING GXDESPOOLPAGE

The meat of the 4-Up extension is contained in the GXDespoolPage override. The GXDespoolPage message has the following interface:

```
OSErr GXDespoolPage (gxSpoolFile thePrintFile, long pageNumber,
                    gxFormat pageFormat, gxShape *pagePicture, Boolean *formatChanged);
```

Normally, the default implementation of this message will be executed, resulting in the page identified by pageNumber being read from the print file specified by thePrintFile. Additionally, the page's associated format will be read into pageFormat, and the Boolean formatChanged will be set to true if the format being returned is different from the format returned by the last invocation of the GXDespoolPage message. Formats are abstract objects that contain formatting information for one or more pages in a document. We'll need to query the page format for its dimensions so that we can properly place our four scaled-down pages onto the physical page.

Our GXDespoolPage override must do the following work:

1. Using pageNumber, compute the actual page numbers for the pages we must read. Remember, when QuickDraw GX asks for page 2, we'll be returning pages 5, 6, 7, and 8. Page numbers start at 1.

2. Read up to four pages from the print file, being careful not to read past the last remaining page.
3. Query the page's format for its dimensions. We'll be fairly unsophisticated and ignore all but the first format for each group of four pages. As a result, we won't handle all cases correctly — for example, mixed portrait and landscape pages. This is left as an exercise for the reader.
4. Scale down and translate each page to the correct size and position.
5. Add the modified pages to a new picture shape, and return it.

This may seem like a lot of work. Fortunately, we can rely on the default implementation of `GXDspoolPage` for much of it. Here's the code:

```
OSErr FourUpDspoolPage (gxSpoolFile thePrintFile, long pageNumber,
                        gxFormat pageFormat, gxShape* pagePicture,
                        Boolean* formatChanged) {
    OSErr    anErr;
    long     firstPage, lastPage, numPages, whichPage;
    gxShape  fourUpPage, thePages[4];
    gxShape* atPage;

    /* Determine actual page numbers of the pages to dspool. */
    lastPage = pageNumber * 4;
    firstPage = lastPage - 3;

    /* Determine page number for last page in spool file so that we can
       constrain our dspooling loop to a valid range if fewer than four
       pages remain in the file. */
    anErr = ForwardMessage(gxCountPages, thePrintFile, &numPages);
    nrequire (anErr, FailedForward_GXCountPages);

    if (lastPage > numPages)
        lastPage = numPages;

    /* Create picture shape to hold subpages. */
    fourUpPage = GXNewShape(gxPictureType);
    anErr = GXGetGraphicsError(nil);
    nrequire (anErr, FailedGXNewShape);

    /* Dspool backwards so that pageFormat ends up containing the format
       for the first page in the group. */
    atPage = &thePages[lastPage-firstPage]; /* Last page in group */
    numPages = 0; /* Track number of successfully dspooled pages */
```

```

for (whichPage = lastPage; whichPage >= firstPage; --whichPage) {
    anErr = Forward_GXDespoolPage(thePrintFile, whichPage, pageFormat,
                                  atPage--, formatChanged);
    nrequire (anErr, FailedForward_GXDespoolPage);
    ++numPages;
}

/* Map the despoiled pages onto a single physical page. */
{
    gxRectangle pageRect;
    fixed      tx, ty;
    gxMapping  aMapping;

    /* Get the dimensions of the physical page. */
    GXGetFormatDimensions(pageFormat, &pageRect, nil);

    /* Compute x and y translation factors. */
    tx = (pageRect.right - pageRect.left) >> 1;
    ty = (pageRect.bottom - pageRect.top) >> 1;

    /* Initialize the mapping to scale by 50%. */
    GXResetMapping(&aMapping);
    aMapping.map[0][0] = fixed1/2;
    aMapping.map[1][1] = fixed1/2;

    /* Map the pages onto the physical page. */
    GXMapShape(thePages[0], &aMapping);

    if (numPages > 1) {
        GXMoveMapping(&aMapping, tx, 0);
        GXMapShape(thePages[1], &aMapping);
        if (numPages > 2) {
            GXMoveMapping(&aMapping, -tx, ty);
            GXMapShape(thePages[2], &aMapping);
            if (numPages > 3) {
                GXMoveMapping(&aMapping, tx, 0);
                GXMapShape(thePages[3], &aMapping);
            }
        }
    }

    /* Place the mapped pages into a single picture. */
    GXSetPictureParts(fourUpPage, 1, 0, numPages, thePages, nil, nil,
                     nil);
    anErr = GXGetGraphicsError(nil);
}

```

```

nrequire (anErr, FailedGXSetPictureParts);

/* GXSetPictureParts cloned the pages, so we must dispose of our
   references to them. */
for (atPage = &thePages[numPages-1]; atPage >= thePages; --atPage)
    GXDisposeShape(*atPage);
}

/* Return the 4-up page. */
*pagePicture = fourUpPage;

/* Since we don't know whether the format for "actual page number 5"
   is the same as that for "actual page number 1," we always set
   formatChanged to true. A more sophisticated extension could do the
   right thing here. */
*formatChanged = true;

ncheck (anErr);
return noErr;

/*-----
Exception-handling code
-----*/

FailedGXSetPictureParts:
FailedForward_GXDespoolPage:
    for (atPage = &thePages[numPages-1]; atPage >= thePages; --atPage)
        GXDisposeShape(*atPage);
        GXDisposeShape(fourUpPage);
FailedGXNewShape:
FailedForward_GXCountPages:
    return anErr;
}

```

Trick #2. This code is pretty easy to follow, but one line demands further explanation. Remember earlier I said that there were two “tricky things” our extension would have to do. The first was correctly determining the number of original pages in the document. The second trick occurs early in the above code, in the line

```
anErr = ForwardMessage(gxCountPages, thePrintFile, &numPages);
```

We need to know the actual page number for the last page in the document so that we can make sure not to read a nonexistent page. Since the last page’s page number is equal to the number of pages in the document, your first thought might be to send

the GXCountPages message. QuickDraw GX supplies the Send_GXCountPages routine to do this, and we would call it like so:

```
anErr = Send_GXCountPages(thePrintFile, &numPages);
```

However, this would not produce the desired result. Since this would invoke our own GXCountPages override, as well as those of all other message handlers that override GXCountPages, the result would be the total number of physical pages actually printed, not the total number of logical pages we must despool.

What we really want to do is forward the GXCountPages message. But here's the potential gotcha! We can't use QuickDraw GX's supplied Forward_GXCountPages routine to do it.

Here's why: The Message Manager provides two routines for forwarding a message. The ForwardMessage routine takes a selector, which indicates the message to be forwarded, and zero or more message-specific parameters. ForwardThisMessage takes only the message-specific parameters and assumes you want to forward the *current* message. The current message is the one corresponding to the override you're currently executing — that is, the override from which you're calling ForwardThisMessage. The problem with calling Forward_GXCountPages from within the GXDespoolPage override is that all QuickDraw GX's Forward_XXX routines are simply inline aliases to ForwardThisMessage, with the message-specific parameters added for type-checking purposes. Since it's far more common to forward the current message than it is to forward an arbitrary message, QuickDraw GX assumes the common case and provides only the corresponding aliases. Therefore, if we were to call Forward_GXCountPages from within our GXDespoolPage override, we would actually forward the GXDespoolPage message with a bogus parameter list!

This doesn't mean we can't forward arbitrary messages, but to pull it off we do have to call the Message Manager directly. In the above code, we call ForwardMessage and pass the constant gxCountPages, defined by QuickDraw GX, as the message selector. *Warning:* You don't get type checking when you call the Message Manager's sending and forwarding functions directly, so be careful out there!

THE JUMP TABLE

The only code left to write is the jump table for our code segment. As you'll see in the next section, QuickDraw GX determines which messages an extension overrides and the location of the associated override code from special resources within the extension. Since these resources specify the location of the extension's overrides in terms of byte offsets from the beginning of a specified code segment, it's easiest to begin each code segment with a simple assembly language jump table. That way, the extension's entry points are always well defined, independent of any changes we make to the code in the future. Also, QuickDraw GX requires that each code segment begin with a long word set to 0, and this is easily accomplished from assembly

language. Since we're overriding only two messages, and our code is small, we'll have just a single segment, with the following jump table:

```
FourUpEntry PROC EXPORT
    DC.L    0    ; long word required by QuickDraw GX

    IMPORT  FourUpCountPages
    JMP     FourUpCountPages

    IMPORT  FourUpDespoolPage
    JMP     FourUpDespoolPage

ENDPROC

END
```

PUTTING IT ALL TOGETHER

Writing code is great, but it's useless if it never runs. There are several things we must do if our printing extension is to be recognized, loaded, and executed by QuickDraw GX.

BE RESOURCEFUL

There are four required resources (besides the code) that must be present in every printing extension. All resources, including your own, should have their `sysHeap` and `purgeable` bits set, unless otherwise noted. Except for code resources, which have IDs starting at 0, all resources should have IDs in the range reserved for printing extensions. This range extends from 0x9600 to 0x97FF (-27136 to -26625 decimal). The predefined constant `gxPrintingExtensionBaseID` (equal to -27136) is provided for your use. All the required resources have predefined constants for their types and IDs. For the actual values, see the appropriate header files.

The four required resources give QuickDraw GX the following important information:

- 'over' resource: which messages you override and where to find the associated override code
- 'eopt' resource: during which phases of printing your extension needs to run, and if and when it modifies the contents of the page
- 'load' resource: where in the message handler chain your extension should load, relative to other printing extensions
- 'scop' resource: which driver and printer types your extension is compatible with

The remainder of this section gives all the gory details for each of the required resources, including examples.

The 'over' resource. The override resource lists the messages you're overriding and where the associated override code is located. Printing extension code resources always have type 'pext' and should have IDs starting at 0 for maximum performance. Given the above jump table, and the fact that we override only universal printing messages, we have a single override resource that looks like this:

```
#define fourUpCodeSegmentID 0
#define gxCountPagesOffset 4 /* first entry follows zero long word */
#define gxDespoolPageOffset 8 /* jump table entries are 4 bytes long */

resource 'over' (-27136, purgeable, sysHeap) {
    {
        gxCountPages, fourUpCodeSegmentID, gxCountPagesOffset;
        gxDespoolPage, fourUpCodeSegmentID, gxDespoolPageOffset
    };
};
```

If your extension overrides imaging messages, you'll need separate override resources for each distinct class of imaging messages you override. For example, PostScript message overrides would go in a separate table, and vector overrides in yet another table. You can choose any ID within the printing extension range for these tables. You let QuickDraw GX know the override resource ID with a mapping resource whose type is the same as the driver type for the imaging messages you're overriding, and whose ID is equal to `gxPrintingExtensionBaseID`. There are predefined constants for these values. For example, if your extension overrides the PostScript message `gxPostscriptEjectPage`, you would have the following two resources:

```
resource 'post' (-27136, purgeable, sysHeap) {
    -27135 /* ID of our PostScript 'over' resource */
};

resource 'over' (-27135, purgeable, sysHeap) {
    {
        gxPostscriptEjectPage, postscriptSegmentID, gxPostscriptEjectPageOffset;
    };
};
```

The 'eopt' resource. The extension optimization resource provides QuickDraw GX with additional information that helps it perform optimally under certain conditions. This resource consists of a bit field containing predefined flags that tell the system when the extension executes, whether it needs to communicate with the device directly, and if and when it makes changes to the page. The 4-Up extension

runs during the despooling/imaging phase and changes the page during the GXDespoolPage message. It doesn't need to communicate with the device.

Using the predefined resource template, our 'eopt' resource looks like this:

```
resource 'eopt' (-27136, purgeable, sysHeap) {
    gxExecuteDuringImaging, gxDontNeedDeviceStatus,
    gxChangePageAtGXDespoolPage, gxDontChangePageAtGXImagePage,
    gxDontChangePageAtGXRenderPage
};
```

The 'load' resource. The extension load resource tells QuickDraw GX your default loading order preference. The first extension's message handler is loaded directly above the driver. Subsequent extensions are loaded one above the other. The last extension to be loaded is the first to override a given message, and therefore has the most control. Most extensions should use the predefined constant `gxExtensionLoadAnywhere`, which indicates that the extension has no loading preference. If you prefer to load first, use the constant `gxExtensionLoadFirst`; if you prefer to load last, use `gxExtensionLoadLast`. You should regard this resource as a hint, not as a guarantee. For one thing, several extensions may indicate that they want to load last. Obviously, only one will win. More important, the user can reorder the extensions in any way desired, and that ordering always takes priority over the default ordering.

Our 'load' resource looks like this:

```
resource 'load' (-27136, purgeable, sysHeap) {
    gxExtensionLoadAnywhere
};
```

The 'scop' resource. The extension scope resource tells QuickDraw GX the scope of your extension's compatibility with the various driver types that are supported by QuickDraw GX. Built-in support exists for raster devices, such as the ImageWriter and LaserWriter SC, vector devices, such as plotters, and PostScript devices. If your extension is PostScript-only, you would specify that in a 'scop' resource. An example of a PostScript-only extension might be one that drives a sheet feeder attachment to a LaserWriter, which understands PostScript commands for selecting bins.

You may have up to three separate 'scop' resources. The main 'scop' resource lists the types of drivers that the extension is compatible with. It has an ID of `gxPrintingExtensionBaseID`. The currently supported types are 'rast', 'post', 'vect', and 'univ', for raster, PostScript, vector, and universal, respectively. For example, an extension compatible with PostScript and vector drivers, but not with raster drivers, would have the following 'scop' resource:

```
resource 'scop' (-27136, purgeable, sysHeap) {
    {
        'post'; /* compatible with all PostScript devices */
        'vect'; /* compatible with all vector devices */
    };
};
```

The second 'scop' resource has an ID of `gxPrintingExtensionBaseID+1` and lists the specific printer types that the extension is compatible with. A printer's type is defined by the creator type of its resource file. For example, the LaserWriter SC has the creator type 'lwsc'. If your extension isn't generally compatible with a class of devices but does support a particular device, you should list it here. For example:

```
resource 'scop' (-27135, purgeable, sysHeap) {
    {
        'lwsc'; /* compatible with LaserWriter SC */
    };
};
```

The third 'scop' resource has an ID of `gxPrintingExtensionBaseID+2` and lists the specific printer types that the extension is *not* compatible with. If your extension is generally compatible with a class of devices but doesn't support a particular device, you should list it here. For example:

```
resource 'scop' (-27134, purgeable, sysHeap) {
    {
        'dpro'; /* incompatible with DraftPro plotter */
    };
};
```

Taken together, the above three 'scop' resources would indicate that the extension is compatible with all PostScript devices, all vector devices except for the DraftPro plotter, and additionally the LaserWriter SC printer.

If your extension is not driver-specific, you can indicate that it has universal scope. 4-Up is one such extension, so we have a single 'scop' resource that looks like this:

```
resource 'scop' (-27136, purgeable, sysHeap) {
    {
        'univ'; /* universal scope => compatible with all devices */
    };
};
```

Note that if this 'scop' resource had instead included 'post', 'vect', and 'rast', the extension would indeed be loaded for all three device types. However, should a fourth

device type be defined in the future, the extension would not support it. Thus, if your extension truly has universal scope, you should use the 'univ' type rather than enumerating all known device types.

BUILDING THE BEAST

We've written the code, we've added the necessary resources. All that's left is to build it properly. The two most important things to remember are:

- Link the code so that it's given the proper resource type ('pext') and ID (zero-based).
- Set the file's type to 'pext' and the creator type to something unique. This is very important! Your extension's creator type should not be all lowercase (lowercase types are reserved by Apple) and you should register it with the Developer Support Center (AppleLink DEVSUPPORT) to ensure uniqueness. When QuickDraw GX encounters two printing extensions with the same creator type, it will reject all but the first it finds. Your Link and Rez step will look something like this:

```
Link      -ra =resSysHeap,resPurgeable  0
          -t 'pext'                      0
          -c '4-Up'                      0
          -rt pext=0                      0
          -sg 4-Up                       0
          -m FourUpEntry                  0
          {CObjs}                         0
          -o "4-Up";
Rez -rd -o "4-Up" 4-Up.r -append
```

BUT WAIT! THERE'S MORE!

There you have it — a completely functional printing extension, with minimal effort. Hopefully, by now you're brimming with ideas for extensions you can write. The mechanism is extremely powerful, and I've barely scratched the surface in this article. Serious extensions will need to override the dialog messages to install panels and gather user settings, call the Collection Manager to satisfy persistent storage needs (that is, save user settings across the spooling/imaging threshold), and call the Message Manager to manage dynamic contexts (global state that persists from one message to the next).

Extensions can also customize the Finder's desktop printer menu, save printer-wide settings in desktop printers, and manage user interface interactions via the built-in alert and status mechanisms. Perhaps a future article will explore these and other topics. For now, you should have enough information to get more than your feet wet!

QUICKDRAW GX

FOR POSTSCRIPT

PROGRAMMERS

With QuickDraw GX, the Macintosh gets a brand new, powerful, and totally different model for text and graphics. Programmers of graphics and page layout applications accustomed to using custom PostScript code during the printing process will have to learn new techniques for imaging on the Macintosh, but the reward is a robust feature set, an easier API, and consistent output whether to a screen (of any resolution or color depth) or a printer (PostScript, raster, or vector). This article should help those programmers make the transition.



DANIEL LIPTON

QuickDraw, while a powerful imaging model for its time and well suited for interactive graphics on the screen, lacks many features demanded by today's users. To provide features such as transformation (rotation, skewing, and so on) and Bézier curves (ubiquitous in most modern graphics applications), applications in a QuickDraw world must do much of the work of drawing to the screen themselves. However, when printing to PostScript devices such as the Apple LaserWriter, these applications can offload much of this work to the printer by simply using the PostScript language to draw most, if not all, of their graphics and text. For this reason, many Macintosh application programmers have also become PostScript programmers and know how to get things done with the PostScript language.

WHAT IS POSTSCRIPT?

Before getting into the details of how to make the transition from the PostScript language to QuickDraw GX, you need to understand the two models. The article "Getting Started With QuickDraw GX" in this issue of *develop* provides an introduction to QuickDraw GX. For an overview of the features of the PostScript language, read on.

The PostScript language is probably best known as a robust graphics model with many capabilities. These capabilities include the ability to fill or frame paths made up of line and cubic Bézier segments, render continuous tone images in both color and

DANIEL LIPTON, in addition to being an accomplished PostScript programmer, is an avid animal lover. He lives with a variety of pets, most notably his dog SpotFunction. As a result of many hours of training, SpotFunction can perform some impressive tricks, including both "roll" and "loop." Dan's affinity for animals extends beyond the canine domain to include his pet iguana, who can neither roll nor loop. Although warm-blooded

himself, Dan can often be found sunning himself on a rock outside his office at Apple. "I find myself mysteriously drawn to the reptilian lifestyle," he confesses, his eyes intently tracking a fly buzzing about his office. Dan is known to break into fits of uncontrollable laughter whenever he's shown a picture of a gorilla, a fact that his coworkers often use to their advantage during meetings. •

grayscale, transform graphics with a matrix, and clip to a path made up of line and cubic Bézier segments. PostScript code can also draw text in a variety of different typefaces and manipulate this text as a graphic.

To a limited extent the PostScript language is also a printing model. Certain operators in the PostScript language are related to printing. These include operators for page control (**showpage** and **copypage**), for controlling paper selection, and for controlling device-specific features (**setpagedevice** in PostScript Level 2).

In addition, the PostScript language serves as a document interchange format. Since it's so widely available on so many different platforms and printers, a PostScript file can be treated as a device-independent document interchange. (However, it's not easily edited except by an expert.) Similarly, it's also used to export clip art. Encapsulated PostScript (EPS) files are widely used for exporting and importing artwork into documents.

But the most important attribute of the PostScript language is that, more than a graphics model, it's a programming language with most of the constructs of modern high-level languages. The PostScript language is really a wrapper for the PostScript graphics model. The graphics are invoked by operators in the language. This full programmability makes it easy for programmers to extend the PostScript model to meet their needs. If a desired feature isn't in the PostScript graphics model, it can frequently be programmed in the PostScript language. For example, PostScript Level 1 doesn't contain patterns, but a PostScript procedure can be written to fill a PostScript path with a pattern.

Due to this programmability, it's possible to emulate directly on PostScript printers many of the QuickDraw GX features that aren't present in the PostScript graphics model. When QuickDraw GX generates a PostScript stream, it includes a complex set of PostScript procedures to do so.

COMPARING QUICKDRAW GX AND POSTSCRIPT

This section compares QuickDraw GX and the PostScript language in terms of their graphics, text-drawing, printing, and programming models.

THE GRAPHICS MODEL

Some differences between the QuickDraw GX and PostScript graphics models include math types, Bézier curves, matrix transformation, and orientation of the y-axis.

Math types. Before entering the world of QuickDraw GX programming, a PostScript programmer must understand the basic differences in how numbers are represented by QuickDraw GX and the PostScript language.

The PostScript language uses floating-point numbers and QuickDraw GX uses fixed-point numbers. The advantage to floating-point numbers is numeric range; the advantage to fixed-point numbers is speed. With fixed-point numbers, addition and subtraction are no slower than with regular integers. QuickDraw GX uses 16.16 fixed-point numbers (32-bit numbers with the high 16 bits representing the integer portion and the low 16 bits representing the fractional portion).

In the PostScript language, color component values are represented by floating-point numbers between 0.0 and 1.0. In QuickDraw GX, color component values are represented by a type called `colorValue`, which is really a short such that 0x0000 is 0.0 and 0xFFFF is 1.0. QuickDraw GX also uses a type called `fract`. The `fract` type is like the fixed type except that only the high two bits are the integer portion and the low 30 bits are the fractional portion. This is generally used for numbers between -1 and 1 where fractional precision is important.

Curves. Both QuickDraw GX and the PostScript language support Bézier curves. However, each supports a different kind (see Figure 1). While the PostScript language uses cubics, GX uses quadratics. A cubic Bézier curve segment is defined by four control points: a starting point on the curve, two points off the curve, and an ending point on the curve. A quadratic Bézier curve is defined by three control points: a starting point on the curve, a control point off the curve, and an ending point on the curve.

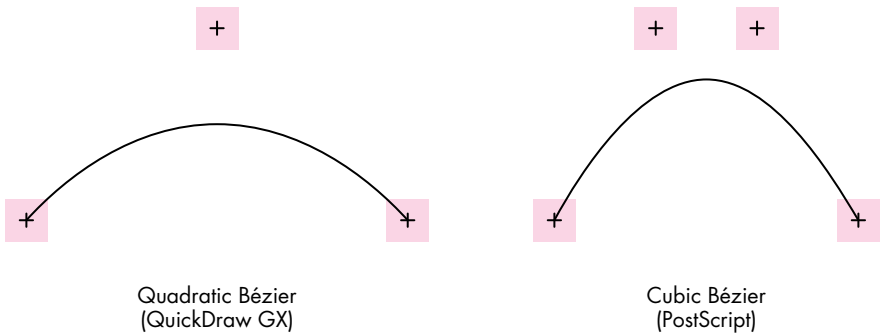


Figure 1
Comparing Control Points on Bézier Curves

Figure 2 illustrates two similarly shaped paths. The one on the left is defined by two quadratic segments, requiring five control points. The one on the right is defined by a single cubic segment, requiring four control points. This seems to imply that in drawing similar shapes, more points are required using quadratics than using cubics and that, therefore, quadratics are at a disadvantage. However, to reduce data size, the data structure for a QuickDraw GX path allows implied points. Each point in the

QuickDraw GX path has a control bit, indicating whether the point is on or off the curve. If two consecutive points in the path are off the curve, there's an implied point halfway between the two explicitly specified points. So, as shown in Figure 2, it's only necessary to supply four points for the quadratic path, as the point between point 2 and point 3 is implicit.

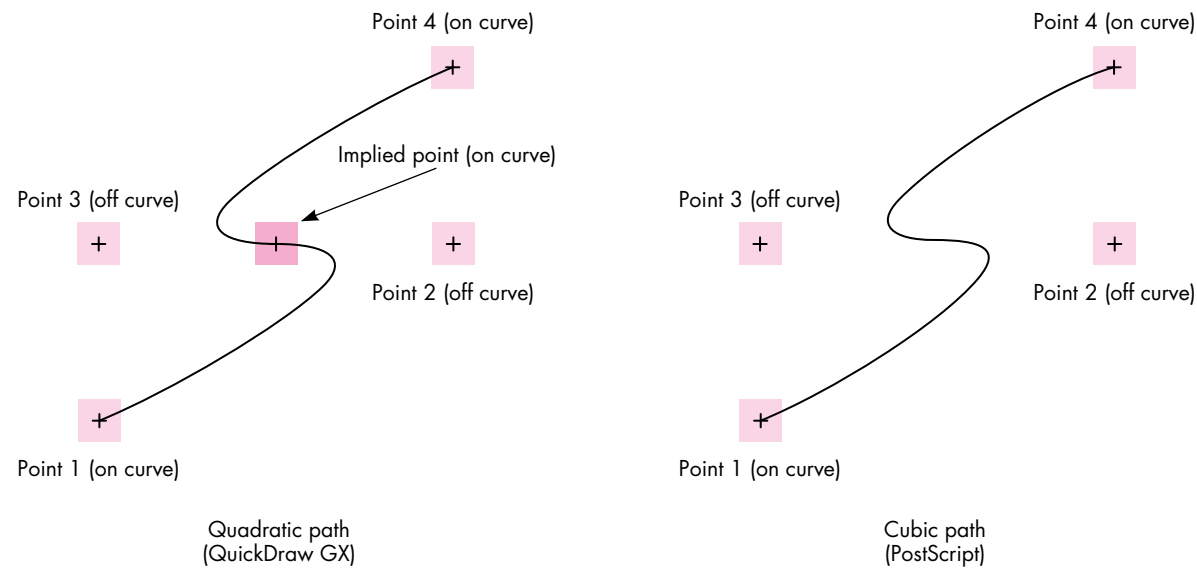


Figure 2
Control Points for Paths

Matrix transformations. Both QuickDraw GX and the PostScript language allow anything to be transformed through a matrix before being drawn. Both use a 3 x 3 transformation matrix. However, in the PostScript language the matrix has implicit constant values in the last column, so there are only six degrees of freedom rather than nine. QuickDraw GX allows you to specify all nine elements of the matrix.

To modify the current transformation matrix (CTM) by a given transformation, an application may use the following PostScript code:

```
[ 4.17 0.0 0.0 -4.17 -1280.0 1650.5 ] concat
```

This code concatenates the following matrix with the CTM in the graphics state:

$$\begin{bmatrix} 4.17 & 0.0 & 0.0 \\ 0.0 & -4.17 & 0.0 \\ -1280.0 & 1650.5 & 1.0 \end{bmatrix}$$

QuickDraw GX has a data structure called `gxMapping`, which is a structure containing one field. The field is a 3 x 3 array. The first two columns contain fixed-point numbers and specify the skewing, scaling, rotation, and translation of the transformation. The third column is made up of fractional numbers (numbers of type `fract`) and specifies the perspective portion of the transformation. The following code generates a mapping that's equivalent to the matrix in the PostScript code:

```
/* Declare a mapping structure (fract1 is a constant for 1.0 in
   mathtypes.h). */
gxMapping      aMapping = { { fl(4.17), fl(0.0), fl(0.0) }
                             { fl(0.0), -fl(4.17), fl(0.0) }
                             { -fl(1280.0), fl(1650.5), fract1 } };
```

The y-axis. The y-axis orientation differs in the PostScript graphics model and QuickDraw GX. In the PostScript model, the y-axis increases from the bottom of the page or window to the top and in QuickDraw GX, as in QuickDraw, it increases from top to bottom (see Figure 3).

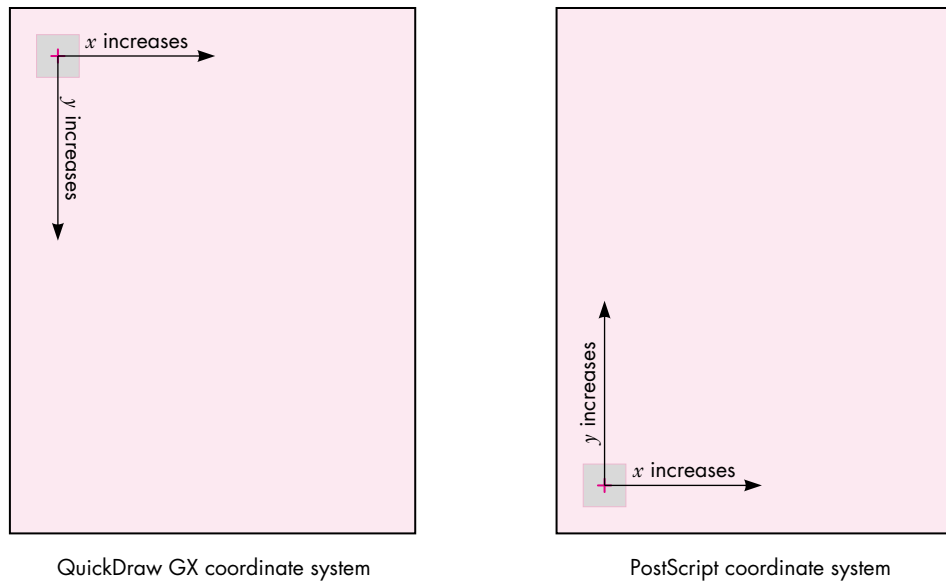


Figure 3
Coordinate Systems

OBJECT-BASED MODEL VERSUS STREAM-BASED PROTOCOL

A fundamental difference between graphics code for QuickDraw GX and PostScript code is that QuickDraw GX is object-based and PostScript code is essentially a

stream-based protocol. Although the PostScript language is a programming language, documents usually consist of a set of PostScript procedures followed by a stream that invokes those procedures. Each model has advantages: With a stream-based protocol the graphic content of any given page is virtually infinite. As long as PostScript code is continuously streamed to the printer, it renders into the frame buffer until **showpage** is issued — which essentially says, “This page is done; start the next one.” With an object model it’s relatively easy to share data between objects. A quick summary of objects in QuickDraw GX illustrates this advantage.

QuickDraw GX objects. The shape object is the basic element of the QuickDraw GX graphics model. A shape contains a geometry of any primitive type and points to three other objects that describe how to render that geometry: the ink object, which describes how to apply color to the geometry (as well as transfer mode); the style object, which describes how to affect the geometry before rendering (pattern, dash, and so on); and the transform object, which describes how to map and clip the geometry before rendering. These objects can, in turn, point to other objects. For example, a transform object points to a list of view port objects that describe where to draw the geometry (such as in which window). An ink object points to a color profile object that describes the colors in the ink in a device-independent manner. All the previously described objects could also have lists of tag objects. A tag object is simply a container for any data the application associates with the owning object.

Data sharing is extremely easy in this object model. If I have a picture made up of 100 different shapes and 30 of them have the same color, these 30 shapes can all point to the same ink object. The color for these 30 shapes is stored only once. In a stream-based protocol, it’s only convenient to share data between consecutive items in the stream. (You can write PostScript code that shares data between nonconsecutive objects, but it’s not easy.)

PostScript procedures and dictionaries versus QuickDraw GX objects.

Emulating the object model in PostScript code is possible because it’s a programming language. You could use PostScript dictionaries as containers for shapes and then have a PostScript procedure that draws one of these dictionaries. The following is a simple example of how this could work. (Warning: Serious PostScript code ahead.)

```
/aShape 7 dict def          % Make a dictionary for the shape.
aShape begin                % Put it on the dictionary stack.
  /geometryProcedure {      % Define a procedure for the geometry
    newpath                 % to draw a rectangle.
    100 100 moveto
    0 100 rlineto
    100 0 rlineto
    0 -100 rlineto
    closepath
  } bind def
```

```

% Dictionary entries for transform.
/Transform [ 10 0 0 10 0 0 ] def

% Dictionary entries for the color.
/redComponent 1.0 def
/greenComponent 0.0 def
/blueComponent 1.0 def

/penWidth 5.0 def
/fillType (framed) def
end                                     % Dictionary definition.

% The following procedure takes a shape dictionary and draws it.
/DrawShapeDict {
  begin                               % Put the shape dictionary on the stack.
  gsave                               % Shape shouldn't affect graphics state.
  Transform concat                     % Apply transform.
  redComponent greenComponent blueComponent setrgbcolor % Set the color.
  geometryProcedure                   % Execute the geometry procedure.
  fillType (framed) eq {              % If the shape is framed,
    penWidth setlinewidth             % set the pen width and
    stroke                             % stroke the path.
  } {                                   % Else, fill the shape.
    eofill
  } ifelse
  grestore
  end
} bind def

% The following would be in the stream to draw the shape stored in the
% dictionary.

aShape DrawShapeDict

```

You could use PostScript code in this manner, but most printers have limited memory, and memory management in PostScript printers is difficult (a subject for another article), so it's usually not done.

Graphics state versus shape attributes. In a stream-based graphics model, a graphics state determines how a particular item is drawn. In the PostScript language, the graphics state attributes include the color, pen thickness, transformation matrix, clip, miter limit, end caps, and joins that will be used to fill or stroke the current path, bitmap, or text to be drawn. Applications using PostScript code must efficiently manage the graphics state — you never want to send more information to the printer than necessary, but sending too little is fatal to the fidelity of the graphics. So, an

application emitting PostScript code must send the color for the item to be drawn only if it's different from that of the last item drawn, and do similarly for pen thickness, transformation matrix, and so on.

In the QuickDraw GX object-based model, every shape points to all the information necessary to draw itself. An application merely needs to call `GXDrawShape` to draw the shape properly with the designated color and pen thickness and other designated characteristics. The application no longer has to keep track of the graphics state. However, there's a different burden (though less cumbersome): making sure shape objects share other objects when possible to reduce the memory used by the picture.

The following code samples effectively illustrate the difference between the two models. Each sample draws two rectangles, one red and one blue, and offsets the second one by (100, 100). First, here's the PostScript code:

```
/DrawRect {100 100 moveto 25 0 rlineto 0 25 rlineto -25 0 rlineto
  closepath fill} bind def
gsave
1.0 0.0 0.0 setrgbcolor      % Set the color red.
DrawRect
100 100 translate            % Move the coordinate system by 100,100.
0.0 0.0 1.0 setrgbcolor      % Set the color blue.
DrawRect
grestore
```

Now compare the QuickDraw GX code:

```
void GXDraw2Rectangles()
{
    gxRectangle    aRectangle = { ff(100), ff(100), ff(125), ff(125) };
    gxShape        rectShape;
    gxInk          redInk, blueInk;
    gxColor        aColor;

    aColor.space = rgbSpace;          /* Color will be RGB. */
    aColor.profile = nil;             /* No color profile. */
    aColor.element.rgb.red = 0xFFFF;
    aColor.element.rgb.green = 0x0000;
    aColor.element.rgb.blue = 0x0000;
    redInk = NewInk();                /* Make red ink. */
    GXSetInkColor(redInk, &aColor);

    aColor.element.rgb.red = 0x0000;
    aColor.element.rgb.green = 0x0000;
    aColor.element.rgb.blue = 0xFFFF;
```

```

blueInk = NewInk();                /* Make blue ink. */
GXSetInkColor(blueInk, &aColor);

rectShape = GXNewRectangle(&aRectangle); /* Create a shape. */

GXSetShapeInk(rectShape, redInk);    /* Use red ink. */
GXDrawShape(rectShape);              /* Draw it. */

/* Move it over and draw it in blue. */
GXMoveShape(rectShape, ff(100), ff(100));
GXSetShapeInk(rectShape, blueInk);   /* Use blue ink. */
GXDrawShape(rectShape);              /* Draw it. */

/* Clean up. */
GXDisposeShape(rectShape);
GXDisposeInk(blueInk);
GXDisposeInk(redInk);
}

```

The PostScript code uses a procedure to draw the rectangle. The procedure is analogous to the shape object. However, each time the rectangle is drawn, the graphics state must be modified to change the color and the transformation. At the end, the graphics state for subsequently drawn items is blue and the origin is shifted by (100, 100) from the original rectangle. The **grestore** operator is needed to set the graphics state back to what it was to begin with.

The QuickDraw GX code created a shape, made it red, drew it, moved it, made it blue, and drew it again. No global state was affected, only the shape itself. Moving the rectangle with PostScript code necessitated modifying the graphics state's CTM. With QuickDraw GX code, moving the rectangle involved only translating the shape's own geometry with the `GXMoveShape` routine.

QuickDraw GX database versus PostScript container. In QuickDraw GX a picture is a type of shape object whose geometry is a list of other shapes. Those shapes in the list can also be picture shapes. Therefore, a QuickDraw GX picture shape is a hierarchical database of shapes. This database can be queried and modified with QuickDraw GX routines such as `GXSetPictureParts`, which inserts or replaces shapes in a picture shape, and `GXGetPictureParts`, which retrieves shapes from a picture. Since a picture can have objects that refer to each other, QuickDraw GX must have the whole picture shape available at one time (although not actually in memory, but rather in the disk-based backing store in low-memory conditions).

A PostScript file describing a picture is essentially a container for graphics. The file contains all the data needed to draw the picture, but it can't be readily edited or queried without having an interpreter for the PostScript language built into your

application. The PostScript stream-based protocol lets the device draw the stream on the fly.

Again, each model has its advantages. The object model is best suited for interactive applications and the stream-based protocol is best suited for printers with limited memory and no disk drives.

TEXT-DRAWING MODELS

QuickDraw GX contains three different types of shape objects for drawing text: the text type, the glyph type, and the layout type. The text type is the simplest of the three, although it's not the most primitive form. Drawing a text shape is similar to using the PostScript **show** operator, as illustrated by the following code samples. First, here's a PostScript "Hello World" program:

```
/Times-Roman findfont      % Get the font dictionary for Times.
24 scalefont               % Scale it to 24 points.
setfont                    % Make it the current font.
100 100 moveto              % Move to location 100, 100.
(Hello World) show         % Draw the text.
```

Now here's the QuickDraw GX "Hello World" program:

```
void GXHelloWorld(void)
{
    gxShape      helloShape;
    gxFont       aFont;
    gxPoint      location = {ff(100), ff(100)};

    /* Find the font object for Times. */
    aFont = FindPNameFont(fullFontName, "\pTimes Roman");

    /* Make a text shape. */
    helloShape = NewText(11, "Hello world", &location);

    GXSetShapeFont(helloShape, aFont);
    GXSetShapeTextSize(helloShape, ff(24));
    GXDrawShape(helloShape);
    GXDisposeShape(helloShape);
}
```

You can use these examples to help you get started with drawing text in QuickDraw GX. They also show some similarities between the PostScript model and the QuickDraw GX model. Both have font entities: in PostScript code it's a dictionary and in QuickDraw GX code it's an object. In PostScript code, the font matrix entry in the dictionary itself is scaled by the **scalefont** operator to set the point size; in

QuickDraw GX code, the point size is contained in the shape's style and can be set by a call to `GXSetShapeTextSize`. For the most part, that's it for similarities.

Characters and glyphs. To understand the full capabilities of QuickDraw GX typography, you must first understand the difference between characters and glyphs. Characters are symbols that have linguistic meaning, usually a letter from an alphabet. Glyphs are renditions of those characters or combinations of them. For example, for any given character from an alphabet, there may be various forms of this character that are appropriate to draw at different times (see Figure 4).



Figure 4
Different Glyphs From a Roman Font

The most complex text drawing in QuickDraw GX comes from the layout shape. With a layout shape, the application specifies which characters in the language make up the piece of text to be displayed. Given the language and script system specified in the layout shape's style object, QuickDraw GX can then figure out which glyphs to use for those characters.

The top line of Figure 4 shows three glyphs from a particular Roman font. They're the glyphs for lowercase *f*, lowercase *i*, and a lowercase *fi* ligature. The ligature is an example of a glyph that represents two characters. With a layout shape, QuickDraw GX can detect when the *i* follows the *f* and automatically choose the *fi* ligature glyph when drawing. This allows the user to type *f* followed by *i* rather than having to figure out what key combination to type and what font to select to get the *fi* ligature.

The bottom line of Figure 4 illustrates another example of different glyphs for the same character. The glyph on the left is the normal capital *A* for that font. The glyph

on the right is a glyph to use at the beginning of the line. With a layout shape, QuickDraw GX detects when the character is at the beginning of the line and automatically chooses the correct glyph.

Platforms and encodings. In most versions of the PostScript language, any given encoding of a font has access to only 256 glyphs at a time. If a font contains more than 256 glyphs, you must use different font dictionaries, each with a different encoding. In effect, the application generating the PostScript code must select from different font dictionaries to create all the glyphs in a given typeface. QuickDraw GX can take advantage of fonts that contain up to 65535 glyphs, all of which are available to any shape object.

The bytes contained in the geometry of a text, layout, or glyph shape can have different meanings depending on the `gxFontPlatform` and `gxFontScript` attributes set in the style object. When `gxFontPlatform` is set to `gxMacintoshPlatform` and `gxFontScript` is set to `gxRomanScript`, the stream of bytes means the same thing it does on a Roman Macintosh system today. If `gxFontPlatform` is set to `gxGlyphPlatform`, the bytes are taken two at a time as a short and are treated as glyph indices in the font; a shape object then has direct access to all the glyphs in a font. Each font indicates which platforms, languages, and scripts it supports.

Positioning glyphs in a line of text. The PostScript language provides several methods for allowing your application to explicitly position glyphs on a page. The simplest is the **show** operator. This operator simply draws each glyph specified by the string and moves the current point by the advance width of that glyph. The **ashow** and **awidthshow** operators allow the application to modify the advance width for all glyphs or a particular glyph. With the **kshow** operator you can call a general procedure between the drawing of each pair of glyphs specified by the string and the procedure can modify the graphics state before drawing the next glyph. This process is generally used for kerning. In kerning the procedure is passed the two character codes, uses those two codes to look in a kerning table, and modifies the current point appropriately. This method, while totally flexible, is difficult to use because the application must parse font metric files to derive kerning tables to use with the **kshow** procedure.

PostScript Level 2 provides a way to position characters without executing a procedure for each glyph drawn — the **xshow**, **yshow**, and **xyshow** operators. With these operators the application can specify an array of advance widths to use in place of the built-in advance widths of the font. This is faster than using the **kshow** operator. Again, the application must generate the advance widths to use, usually by parsing font metric tables and deriving kerning information.

It's possible for applications to generate PostScript code that uses the **kshow** operator or the **xshow**, **yshow**, and **xyshow** operators to justify, kern, and track text. In QuickDraw GX, a layout shape does this automatically, as specified by metrics in the

QuickDraw GX font. Each font contains tables that specify kerning pairs with kerning amounts, optimal tracking values, and optimal choices for how justification should occur. Your application can override these values if you choose, but the values in the font are written by the font designer and therefore cause QuickDraw GX to position the glyphs as the font designer intended. Your application need not parse the font metric tables and position glyphs directly.

When you use layout or glyph shapes, text can have multiple style runs. This allows a single shape object to switch between fonts, sizes, text faces, and languages as many times as desired (see Figure 5).

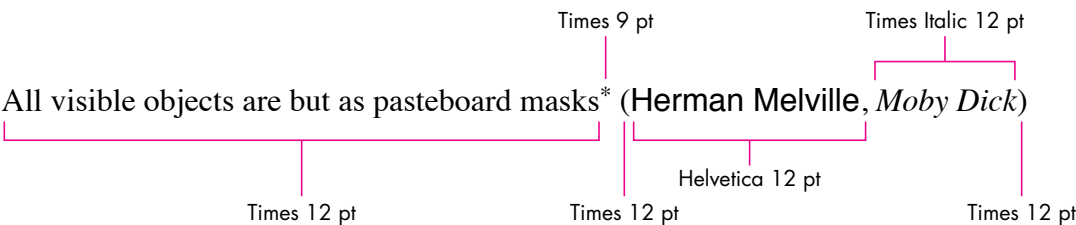


Figure 5
A Shape Object With Multiple Style Runs

If you want the application to have direct control over positioning glyphs, use a glyph shape object rather than a layout shape object. Glyph shapes bypass the automatic positioning information. This approach is similar to using PostScript operators. When using a glyph shape, you specify exactly which glyphs are to be drawn in what styles and at what positions and angles. Then, when GXDrawShape is called, it uses this information for rendering.

Using the positions and advance bits in a glyph shape, your application can draw the glyphs anywhere on the page. Figure 6 illustrates some of the data in a glyph shape with various values in the positions and advance bits. Where the advance bit is 1, the

Glyphs	A l l v i s i b l e o b j e c t s																	
Advance bits	1	1	1	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1
Positions/deltas	(x ₀ ,y ₀)	(x ₁ ,y ₁)	(x ₂ ,y ₂)	(x ₃ ,y ₃)	(0,0)	(0,0)	(0,0)	(0,0)	(1,0)	(1,0)	(1,0)	(1,0)	(1,0)	(x ₁₄ ,y ₁₄)	(x ₁₅ ,y ₁₅)	(x ₁₆ ,y ₁₆)	(x ₁₇ ,y ₁₇)	(x ₁₈ ,y ₁₈)

Figure 6
Some of the Data in a Glyph Shape

value in the positions array is that glyph's absolute position on the page (before being mapped through the shape's transform). Where the advance bit is 0, the value in the positions array is an amount to add to the normal advance vector of the glyph. Not shown in Figure 6 is the tangents array. Each glyph in a glyph shape object also has a tangent vector that specifies an orientation for the glyph in addition to the position.

Given the tangent (T_x , T_y), the glyph is transformed through the following 2 x 2 matrix:

$$\begin{bmatrix} T_x & T_y \\ -T_y & T_x \end{bmatrix}$$

It's important to note that glyph shapes don't do any character-to-glyph mapping, as do layout shapes. They map character codes to glyph codes as specified by the `gxFontPlatform` attribute in the style object, but they don't automatically pick alternate forms of characters (ligatures, for example). If you use glyph shapes in your application, you have to do nearly everything; however, glyph shapes provide the most flexibility.

QuickDraw GX is language aware. When you use the layout type of shape objects, QuickDraw GX is aware of the language and script as specified by the style object. This allows QuickDraw GX to automatically run text, for example, from left to right for English, right to left for Hebrew, and vertically for Chinese. Each font/language combination has a set preference for which way to run text. Because the layout shape can automatically determine where to position the glyphs based on the language, your application can maintain the text for the shape in its linguistic order rather than the display order.

QuickDraw GX also uses script-dependent information when justifying text. For example, in English, justification involves adding or removing white space between glyphs. In Arabic, glyphs are joined by horizontal lines called Kashidas. When justifying Arabic text, QuickDraw GX automatically varies the length of the Kashida to compensate for added or removed space in the text.

PRINTING MODELS

The printing models for QuickDraw GX and the PostScript language differ in much the same ways as the graphics models do. PostScript code uses a stream-based protocol while QuickDraw GX uses an object model.

PostScript stream-based printing. PostScript language elements invoke various printing commands such as commands for choosing a particular type of paper or a particular page orientation. With PostScript Level 1, some implementations added operators for bin selection and other device-dependent features. PostScript Level 2 has the `setpagedevice` operator, which is a generalization of this idea.

A PostScript stream that represents an actual document rather than a particular encapsulated graphic has those various operators embedded between the pages to instruct the printer page by page. In addition to operators, there's a defined protocol for including comments in a PostScript stream to identify the document elements. Some of these occur at the beginning of the stream and some of them occur between the pages. They're called document structuring conventions (DSCs) and are described in detail in the *PostScript Language Reference Manual*, Second Edition.

QuickDraw GX object-based printing. For each element of a printed document, there's a corresponding QuickDraw GX object. Your application simply associates the appropriate objects when spooling the document's pages and QuickDraw GX does the rest. Your application need not worry about the details of paper trays and transformation matrices to reorient the page.

Global document properties, such as the device information and the number of pages or copies, are stored in the job object. Properties associated with a particular page are stored in the format object. Each format object owns a paper type object.

The job object can be thought of as a context for the document that your application is spooling. The format object contains information such as the page orientation (portrait or landscape) and paper type (US Letter, Envelope, and so on). Each page your application generates can have a different format associated with it. The job object contains a default format that's used if a specific format isn't specified for a page. All your application needs to do to set up the contents of these objects is call the QuickDraw GX printing dialog boxes. The following code example shows how five pages in one job can be printed with four different formats. (The contents of each page are stored in a picture shape object.)

```
OSErr Print5Pages(shape page1, shape page2, shape page3, shape page4,
                 shape page5)
{
    OSErr          status;
    EditMenuRecord myEditMenu;
    gxFormat       format1, format2, format3;
    gxJob          myJob;
    DialogResult   result;

    status = GXNewJob(&myJob);
    if (status != noErr) return (status);

    /* Add code here to set up the Edit menu record. */
    . . .

    /* Use dialog box to set up default format for the job. */
    result = GXJobDefaultFormatDialog(myJob, &myEditMenu);
}
```

```

if (result == okSelected) {

    /* Create three separate formats for the first three pages. */
    format1 = GXNewFormat(myJob);
    format2 = GXNewFormat(myJob);
    format3 = GXNewFormat(myJob);

    /* Bring up dialog box to set up page 1's format. */
    result = GXFormatDialog(format1, "\pPage Setup for Page 1",
        &myEditMenu);
    if (result != okSelected) goto canceled;

    /* Bring up dialog box to set up page 2's format. */
    result = GXFormatDialog(format2, "\pPage Setup for Page 2",
        &myEditMenu);
    if (result != okSelected) goto canceled;

    /* Bring up dialog box to set up page 3's format. */
    result = GXFormatDialog(format3, "\pPage Setup for Page 3",
        &myEditMenu);
    if (result != okSelected) goto canceled;

    /* Bring up the Job dialog box. */
    result = GXJobPrintDialog(myJob, &myEditMenu);
    if (result != okSelected) goto canceled;

    /* Now spool the document. */
    GXStartJob(myJob, "\pdevelop Article", 5);
    GXPrintPage(myJob, format1, page1, 1);
    GXPrintPage(myJob, format2, page2, 2);
    GXPrintPage(myJob, format3, page3, 3);
    GXPrintPage(myJob, nil, page4, 4);      /* Page 4 uses job's
                                           default format. */
    GXPrintPage(myJob, nil, page5, 5);      /* So does page 5. */
    GXFinishJob(myJob);

canceled:
    GXDisposeFormat(format1);
    GXDisposeFormat(format2);
    GXDisposeFormat(format3);
}
status = GXGetJobError(myJob);
GXDisposeJob(myJob);
return (status);
}

```

This example calls the QuickDraw GX routines that present dialog boxes, allowing the user to configure all the job and format properties. However, the QuickDraw GX printing API allows the programmer to control these properties directly, if desired. Using this API, your application can exert total control of all aspects of printing without ever bringing up a dialog box!

IF YOU CAN DO IT IN POSTSCRIPT . . .

This section shows you how to use QuickDraw GX to do some of those tricky things you've figured out how to do with PostScript code.

FRAMING SOMETHING WITH A NONSQUARE PEN

QuickDraw has the concept of a nonsquare pen. You can set the width and height of the pen independently. Both the PostScript language and QuickDraw GX have only one pen dimension; however, you can simulate the framing of a path with a nonsquare pen. Here's the PostScript code:

```
% Assuming there exists a path ready for drawing in the graphics state:
gsave                % Save the current graphics state to muck with later.
1 setlinewidth       % Set the current line width to 1.
xPen yPen scale       % Scale the CTM by the pen width and pen height.
stroke               % Stroke the path. The scaled matrix will scale the
                    % 1-unit line width by xPen in the x-axis and yPen in
                    % the y-axis when stroking. This produces the desired
                    % effect.
grestore             % Put back the CTM and line width.
newpath              % Clear the path since we did grestore after stroke.
```

Now, here's how to do it with QuickDraw GX:

```
void FrameNonSquare(gxShape theShape, fixed xPen, fixed yPen)
{
    gxShape    tempShape;
    gxMapping   aMapping;
    gxTransform aTransform;

    /* Make a copy of the shape to operate on. */
    tempShape = GXCopyToShape(nil, theShape);
    /* Make a new transform for the shape so it's scaled by the pen. */
    aTransform = GXCopyToTransform(nil, GXGetShapeTransform(tempShape));
    GXScaleTransform(aTransform, xPen, yPen, 0, 0);
    GXSetShapeTransform(tempShape, aTransform);
    /* Make an inverse mapping to premap the shape so that when it's
       scaled by the pen it will return to its original self. */
    GXResetMapping(&aMapping);    /* Set to identity. */
}
```



```

GXScaleMapping(&aMapping, FixedDivide(ff(1), xPen),
    FixedDivide(ff(1), yPen), 0, 0);
GXMapShape(tempShape, &aMapping);

GXSetShapePen(tempShape, ff(1));    /* Set pen width to 1. */
GXDrawShape(tempShape);             /* Draw it. */
GXDisposeShape(tempShape);
GXDisposeTransform(aTransform);
}

```

MODIFYING GLYPHS IN A FONT

The PostScript language allows you to modify the behavior of glyphs by changing entries in the font dictionary, either directly or with the **makefont** operator.

Oblique text. This PostScript code creates oblique text:

```

/Helvetica findfont 24 scalefont      % Put 24-point Helvetica dictionary
                                        % on the stack.
[1.0 0.0 -0.25 1.0 0.0 0.0] makefont % Skew the font dictionary.
setfont                               % Make it the current font.

```

This is how to do it with QuickDraw GX:

```

/* Modify the style object to do oblique text. */
void ObliqueText(gxStyle aStyle)
{
    gxTextFace    theFace;
    gxTransform    aTransform;

    aTransform = GXNewTransform();          /* Make a transform. */
    GXSkewTransform(aTransform, fix1/4, 0, 0, 0); /* Skew it. */
    theFace.faceLayers = 1;                /* Set text face to 1 layer. */
    GXResetMapping(&theFace.advanceMapping); /* Make advance mapping */
                                           /* the identity mapping. */

    theFace.faceLayer[0].outlineTransform = aTransform;
    theFace.faceLayer[0].outlineStyle = nil;
    theFace.faceLayer[0].boldOutset.x = 0;
    theFace.faceLayer[0].boldOutset.y = 0;
    theFace.faceLayer[0].outlineFill = gxSolidFill;
    theFace.faceLayer[0].flags = 0;

    GXSetStyleFace(aStyle, &theFace);
    GXDisposeTransform(aTransform);
}

```

The text face data structure is used to modify the way glyphs are drawn. A text face can have multiple layers and each layer can have a style (for patterns and so on), a transform, a boldness, and a fill type. By using all of the fields and layers in the text face, you can affect the drawing of text in all sorts of nasty ways. The next example uses the fill type to simulate outline text.

Outline text. This code creates outline text in QuickDraw GX:

```
/* Modify the style object to do outline text. */
void OutlineText(gxStyle aStyle)
{
    gxTextFace    theFace;
    gxStyle        layerStyle;

    theFace.faceLayers = 1;          /* Set text face to 1 layer. */
    GXResetMapping(&theFace.advanceMapping); /* Make advance mapping */
                                          /* the identity mapping. */

    layerStyle = GXNewStyle();
    GXSetStylePen(layerStyle, fix1/16); /* Make pen 1/16 point. */
    theFace.faceLayer[0].outlineTransform = nil;
    theFace.faceLayer[0].outlineStyle = layerStyle;
    theFace.faceLayer[0].boldOutset.x = 0;
    theFace.faceLayer[0].boldOutset.y = 0;
    theFace.faceLayer[0].outlineFill = gxClosedFrameFill;
    theFace.faceLayer[0].flags = 0;
    GXSetStyleFace(aStyle, &theFace);
    GXDisposeStyle(layerStyle);
}
```

To do the same thing with PostScript code, modify the font dictionary:

```
/Helvetica findfont          % Put Helvetica's dictionary on the stack.

dup length 1 add dict begin  % Make a copy of the font dictionary and put
{                             % it on the stack.
    1 index /FID eq {pop pop} {def} if else
} forall

/PaintType 2 def % Make the font PaintType 2. This means stroked.
/StrokeWidth 1.0 16.0 div def % Make the stroke width 1/16.
currentdict
end

/HelveticaFramed exch definefont pop % Define the outlined font.
/HelveticaFramed findfont 24 scalefont setfont % Make it current.
```

CONVERTING FRAMED OBJECTS INTO FILLED OBJECTS

Sometimes you want a shape that, if filled, is the same as the result of stroking the original shape. In PostScript code, calling **strokepath** on the current path applies the current pen width to the path, and the resulting path is one that can be filled to produce the result that calling **stroke** would have produced.

In QuickDraw GX, the `GXPrimitiveShape` routine applies the fill and style to any shape to produce a primitive shape. A primitive shape is one that's completely described by its geometry and fill and doesn't need a style object to be drawn properly. For example, a path that's framed with a pen width of 10 becomes a `solidFilled` shape.

CONVERTING TEXT INTO A PATH

In the PostScript language, the **charpath** operator takes a string and converts it into a path using the current font in the graphics state. The following code converts the word *Hello* into a path using the font and font size of the current graphics state:

```
(Hello) false charpath
```

Any QuickDraw GX text, glyph, or layout shape object can be turned into a path shape object by calling `GXSetShapeType` as follows:

```
GXSetShapeType(myTextShape, pathType);
```

This converts the shape object `myTextShape` into a path shape object by applying the font, font size, and text face in the shape object's style object.

NOW YOU'RE READY FOR QUICKDRAW GX

Whether or not you're familiar with the PostScript language, the preceding samples and comparisons should help you get going on your QuickDraw GX application. In the days of QuickDraw, you frequently had to resort to generating PostScript code from your application because the graphics constructs simply didn't exist in QuickDraw. However, QuickDraw GX is a robust graphics, text, and printing architecture that does all the things that current drawing applications do and then some. There should be no need to generate your own PostScript code from your application in the world of QuickDraw GX. Using QuickDraw GX as the medium for all drawing also gives your application the added benefit of being able to produce application-independent portable digital documents. You can view portable digital documents with TeachText and print them on any printer, PostScript or not. Enjoy!

THANKS TO OUR TECHNICAL REVIEWERS

Pete ("Luke") Alexander, Tracey Davis, Herb Derby, Dave Williams •



DAVE JOHNSON

THE VETERAN NEOPHYTE

THROUGH THE LOOKING GLASS

Symmetry is more interesting than you might think. At first glance there doesn't seem to be much to it, but if you look a little closer you'll find that symmetry runs swift and cold and deep through many human pursuits. Symmetry concepts are found at the heart of topics ranging from the passionately artistic to the coolly scientific, and from the trivial to the fundamental.

I learned a lot about symmetry while trying to learn how to create tile shapes. I've always been intrigued and tantalized by M. C. Escher's periodic drawings, the ones that use lizards or birds or fish or little people as jigsaw puzzle pieces, interlocking and repeating forever in a systematic way to completely tile a surface (mathematicians call this *tessellation* of a plane). My own halting attempts to draw tessellations have met with only tepid success. Especially hard is creating tiles that are recognizably something other than meaningless abstract shapes.

To accomplish this feat of tiling a plane, you have to apply a set of constraints to everything you draw. Every line serves multiple purposes. In one of Escher's prints, for example, the same line that forms the left arm of one lizard also forms the tail of an adjacent lizard. That line is also repeated ad infinitum across the plane; *every* lizard's left arm and tail is defined by that same line shape. Now think about drawing a line like that. Not only are you drawing two shapes with one line (which is difficult enough), but you're also drawing innumerable

identical lines simultaneously. They sort of spin out from the point of your pencil in a dazzling dancing tracery of lines. Trying to hold all that complexity and interrelatedness in your head is very, very difficult.

Being a basically lazy person with too much time on my hands, I decided to write a program that would handle it all for me. I envisioned a direct manipulation kind of thing: as I changed a line, all the other corresponding lines in the pattern would change simultaneously. I figured it would be easy to draw little people and leaves and fishes that perfectly interlocked, if only I didn't have to keep all those interdependencies and constraints in mind and could just draw. Also, I thought maybe that by interactively "doodling" and being able to watch the whole pattern change on the fly, I could get some sort of gut feeling for the constraints.

All this was way back in 1990. To learn more, I bought a book called *Handbook of Regular Patterns: An Introduction to Symmetry in Two Dimensions* by Peter S. Stevens. The book is a sort of systematic catalog of hundreds of regular patterns, including many of Escher's, and also has a great introduction to the mathematics of symmetry (which turns out to figure heavily in this tiling business). Unfortunately, after an intense but superficial examination and an evening or two playing with pencil and paper and little dime store pocket mirrors (bought in a frenzy of excitement the day after I bought the book), I decided that the program would be *way* too hard to write to make it worth it, and shelved the whole thing.

Well, last month I finally picked up the idea again. QuickDraw GX was getting close to being released, and it had features that made it relatively easy to implement what I wanted: very flexible transformation and patterning capabilities, and excellent hit testing, which makes implementing direct manipulation of lines a snap. So I dusted off Stevens's book and my little mirrors and got to work, trying to figure out the constraints on the tiles and implement the program.

Here's a basic fact about tiling a plane that I still find thoroughly remarkable three years after I first learned

DAVE JOHNSON once thought that maybe computers contained the secret of life, but has since decided that no, it can't be found there, either. He's now beginning to look elsewhere. Compost piles (preferably hot, steaming, and active) are currently being eagerly investigated. •

about it: there are only 17 possible arrangements of tiles. “But wait!” I hear you cry in your many-throated voice, “How can that be? Surely there are a very large number — nay, an infinite number — of possible tile shapes?”

Well, yes, that’s true. But the way they fit together, the underlying structure, will always be one of only 17 possibilities. This applies to *any* two-dimensional pattern made up of regularly repeating motifs, not just seamless tilings. The motif that’s repeated, of course, can be anything: a leaf, a loop, or a lizard; a frog, a flower, or a fig — it makes no difference. There are still only 17 ways to build a regularly repeating 2-D pattern. This was proved conclusively in 1935 by a mathematician named von Franz Steiger. (Yes, that’s his name; I checked twice.)

To see why, you need to learn a little about the fundamental symmetry operations and how they combine with one another to breed other symmetry operations. I’ll gloss over most of the details (see Stevens’s book, or any introductory text on crystallography, for more info), but the gist of it is that when you sit down and begin to repeat some motif by repeatedly applying fundamental symmetry operations — like reflection and rotation — you find an interesting thing: combining symmetry operations with one another often causes other types of symmetry to sort of spring into existence. And the operations always seem to gather themselves into the same few groups.

Figure 1 shows a very simple example. We start with a simple motif (a comma shape) and repeat it by applying a transformation to it, in this case by reflecting it across a vertical line. Then we reflect the whole thing again, this time across a line perpendicular to the first one. The resulting pattern of four commas possesses mirror symmetry in two directions, meaning that a reflection of the *entire pattern* across either one of the lines leaves the pattern unchanged. But if you study it, you’ll find another symmetry embedded in the pattern that we didn’t explicitly specify. In particular, it shows *rotational* symmetry: rotating the pattern 180° about its center leaves it unchanged, too.

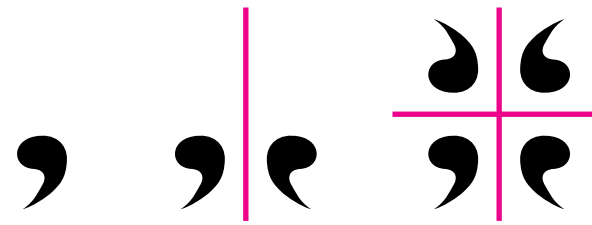


Figure 1
Building a Simple Symmetry Group

Figure 2 shows an alternative way to create the same pattern. This time we begin with the rotation (the point of rotation, or *rotocenter*, is shown by an oval). If we then run a mirror line through the rotocenter, we produce exactly the same structure, the same *symmetry group*, as we did by combining two perpendicular reflections above. These three symmetry operations (two perpendicular reflections and a 180° rotation) come as a set. Combining any two automatically produces a pattern that also contains the third. This is where the constraints on the structure of regular 2-D patterns appear. No matter how you combine and recombine the fundamental operations to cover a plane, you find yourself generating the same 17 arrangements, the same 17 groups of operations.



Figure 2
Another Way to Build the Group

By the way, this example group isn’t one of the 17 plane groups. It’s one of the 10 *point* groups, groups whose constituent transformations operate around a single point. In case you’re curious, there are also 7 *line* groups (ways to repeat motifs endlessly along a line)

and 230 *space* groups (ways to repeat a solid shape to fill three-dimensional space). I don't know if anyone has figured out the groups of higher-dimensional spaces. Knowing mathematicians, I don't doubt it.

So what about that computer program I was going to write? As this column goes to press, it's undergoing its second major overhaul, having suffered mightily from my "write it first, *then* design it" philosophy. So far I have 5 of the 17 groups implemented, and it's pretty cool. There's no telling how far I'll actually get before my deadline arrives, but I'll put the results, however clunky and raw they may be, on this issue's CD so that you can check it out.

I've learned a couple of things already: Even with the constraints automatically handled by the computer, it's still really hard to create representational shapes that will tile a plane, though creating abstract tile shapes is suddenly a piece of cake. Also, I still haven't gotten the kind of gut-level understanding of the *structure* of the patterns that I was hoping for (though just watching them change as I doodle is very entertaining).

I've also learned along the way that symmetry concepts go far deeper than the simple plane groups I'm messing with. The rules of symmetry and of form are, in a sense, manifestations of the structure of space itself. It's an odd thought that space *has* a structure, isn't it? Normally we think of space as a sort of continuous nothingness, as an *absence* of structure or as a formless container for structure. But space itself *does* have a structure, and every single material thing must conform to that structure in order to exist.

Physicists, of course, have been trying very hard for a long time to describe precisely the nature of space. Einstein thought that there was really nothing in the world *except* curved, empty space. Bend it this way, and you get gravity, tie it in a tight enough knot and you get a particle of matter, rattle it the right way and you get electromagnetic waves.

And there are other symmetries, symmetries even more fundamental. Einstein's theory of special relativity broke some of the central symmetries in physics, and thus called attention to the *role* of symmetry in science. Shortly afterward a mathematician named Emmy Noether established a remarkable fact: each symmetry principle in physics implies a physical conservation law. For instance, the familiar conservation of energy law is implied by symmetry in time — energy is conserved *because* time is symmetric. (Of course, I'm greatly oversimplifying here. The symmetry of time is one that Einstein tarred and feathered and ran out of town on a rail. He showed that under extreme conditions time is *not* symmetric, and energy *isn't* conserved. Reassuringly, he replaced these broken and bloodied false symmetries with fresh new ones, but they're well beyond the scope of this column and my poor addled brain.) The point is that symmetries seem to be part of the very fabric of the universe; they seem to be the warp and weft of existence itself.

Yes, it's heady stuff indeed, this symmetry business. I'm staying plenty busy just trying to understand the symmetries possible in a plane, thank you very much, so I'll leave worries about the symmetry of space-time or of K-meson decay to the pros. Once again, I find that by looking just beneath the surface of a seemingly innocuous topic, I find depth and complexity beyond measure. Ain't life grand?

RECOMMENDED READING

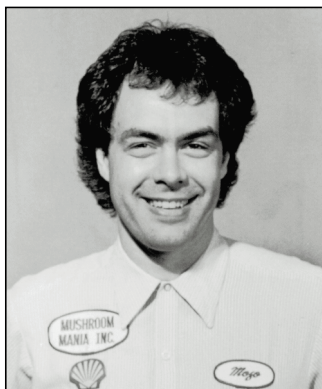
- *Handbook of Regular Patterns: An Introduction to Symmetry in Two Dimensions* by Peter S. Stevens (MIT Press, 1981).
- *Patterns in Nature* by Peter S. Stevens (Little, Brown & Company, 1974).
- *Where the Wild Things Are* by Maurice Sendak (Harper & Row, 1963).

Thanks to Jeff Barbose, Michael Greenspon, Bill Guschwan, Mark Harlan, Bo3b Johnson, Lisa Jongewaard, and Ned van Alstyne (aka Ned Kelly) for reviewing this column. •

Dave welcomes feedback on his musings. He can be reached at JOHNSON.DK on AppleLink, dkj@apple.com on the Internet, or 75300,715 on CompuServe. •

MANAGING COMPONENT REGISTRATION

One of the many design problems a component developer may face is how to register interdependent components in a predetermined fashion so that any given component is registered before the components that depend on it. This article and the sample code that accompanies it show you how to do just that.



GARY WOODCOCK

The Component Manager is an effective mechanism for providing extended functionality to the Macintosh platform. Although a single component can perform impressive tasks, often it's a hierarchy of components, cooperating with one another, that provides the most powerful capabilities. An example of such a hierarchy is found in QuickTime movie playback using the movie controller component (see Figure 1). This component uses the services of many other components, all of which cooperate together, to make interaction with QuickTime movies very simple yet very powerful.

There are distinct advantages to partitioning functionality in this manner. First, by creating components that perform simpler processing, you increase the likelihood that you can leverage the investment you've made in your code by using it in more than one place. Second, it's easier to debug smaller components than a gigantic everything-and-the-kitchen-sink component. Finally, a component that provides very elementary functionality is easier to override or update (via component replacement or capture) than a large, complex component.

This situation — a component depending on the presence of several lower-level components to perform its function — is very commonplace. In such cases, it's important to take steps to ensure that supporting components are available when your component needs them. There are two obvious choices for when to go looking for the components you depend on: when your component is being registered (in its register routine), or when your component is first opened (in its open routine). Most software-dependent components don't need to worry much about managing component registration. Generally such a component should just auto-register, and then check for any required components in the open routine; if the required components aren't available, your open routine can return an error. The caller of

74

GARY WOODCOCK, an optically challenged, melanin-impooverished male who lives with his feline-American companion Phaser, hopes someday soon to be able to spend a few motivationally deficient days enjoying a reduced state of awareness without becoming terminally inconvenienced. He feels P. J. O'Rourke's observation that "Giving money and power to

government is like giving whiskey and car keys to teenage boys" carries far too much truth. •

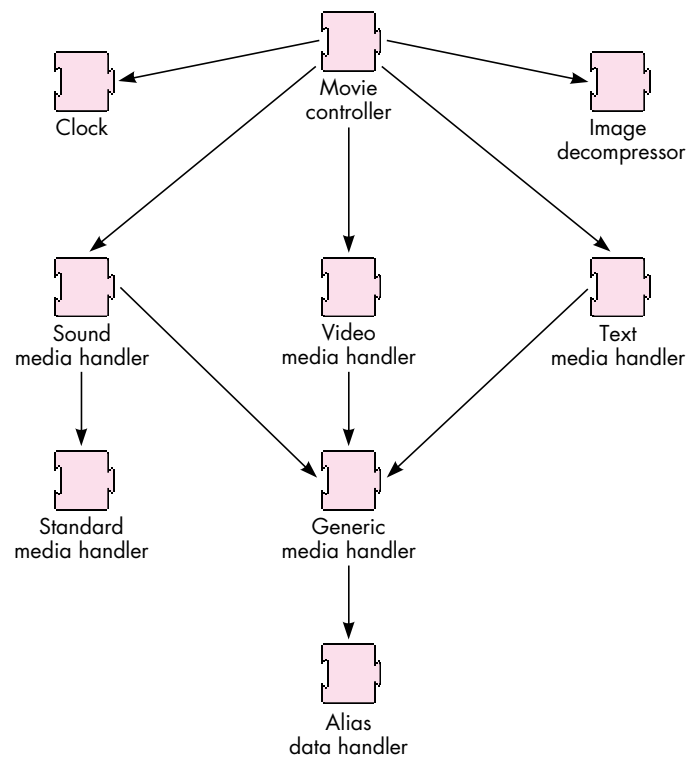


Figure 1
The Movie Controller Component Hierarchy

your component can then handle the error in whatever way is most appropriate. There is a case, however, where checking at registration time might be necessary; that's what this article is about.

DO I REALLY NEED TO WORRY ABOUT THIS?

One potential problem occurs in situations where the Component Manager's registration list is used to build some user interface element, such as a pop-up menu or a list. In this case, the general assumption is that because a component's name is displayed in a user interface element, a user can select it and it will do whatever it's supposed to do — after all, if the component couldn't perform its function, it wouldn't be displayed as an option for the user, right? Well, that depends.

Let's look at an example. The SuperOps company builds the WhizBang video digitizer card and supplies two software components with it — the WhizBang video digitizer component and the WhizBang sequence-grabber panel component (which is used to control features specific to the WhizBang hardware). The component files

For more information on overriding components, see Bill Guschwan's "Somewhere in QuickTime" column in this issue. •

are named WhizBang Video Digitizer and WhizBang Panel. In its register routine, the WhizBang video digitizer component checks for its hardware and registers with the Component Manager only if the hardware is present (this is normal behavior for components that encapsulate hardware functionality). The WhizBang sequence-grabber panel component checks for the availability of the WhizBang video digitizer component when it receives either an open message or a “panel can run” message — it doesn’t get a register message, and therefore it always registers successfully with the Component Manager.

Now let’s say I’ve got a Macintosh Quadra 950 with multiple sound and video digitizers installed (I can dream, can’t I?), one of which is the WhizBang card. I remove the WhizBang card from my computer, but I leave the two WhizBang components installed. I then start up my Macintosh Quadra and run my favorite movie capture application. I display the sequence-grabber video settings dialog box, and I see a dimmed item in the panel pop-up menu — “WhizBang panel.” The dimmed name indicates one of two things: another application has the WhizBang video digitizer open, so it’s not available, or the WhizBang video digitizer component isn’t registered at all, so the panel can’t run.

In this case, we already know that the WhizBang card isn’t installed, so there’s no way this panel can *ever* be enabled, given the current hardware configuration. Rather than confuse users by displaying the panel name in the pop-up menu (even if it is dimmed), it would be nicer if it weren’t displayed at all. To do that, we need to ensure the following order of events at startup: the video digitizer component must attempt to register first, and then the panel component must attempt to register (this implies that the panel component must implement a register routine), checking for the presence of the video digitizer component before it does so. Further, this sequence of events must not be influenced by the alphabetic order of the component filenames. Guess what? We can realize this goal by managing component registration.

This article and the sample code on this issue’s CD demonstrate various ways of managing component registration. We start with the easiest, most obvious approach and work our way up to a more sophisticated solution, pointing out the pros and cons of each along the way. If you just want the “answer” without any fanfare, skip ahead to the section “Mo’ Better: Use a Loader Component to Manage Registration.”

I assume that you’re familiar with the Component Manager and that you know something about how components are written. For more information on these topics, see *Inside Macintosh: More Macintosh Toolbox* and “Techniques for Writing and Debugging Components” in *develop* Issue 12.

THE SYSTEM VERSION AND THE COMPONENT MANAGER

The Component Manager behaves slightly differently depending on the version of system software it’s running under and how the Component Manager was installed.

For more information on QuickTime components, see *Inside Macintosh: QuickTime* and *Inside Macintosh: QuickTime Components* (included in the QuickTime Developer’s Kit v. 1.5) and “Inside QuickTime and Component-Based Managers” in *develop* Issue 13. •

It's important to know about these subtleties in order to understand how to work with the Component Manager to install your components properly.

In system software version 6.0.7, the Component Manager is installed as part of an INIT (usually the QuickTime INIT). During the INIT installation, the Component Manager examines the contents of the System Folder and its subfolders for files of type 'thng'; in each 'thng' file, it looks for resources of type 'thng', which it then uses to register the corresponding components. The important point here is that the Component Manager is not available until after the INIT has been installed.

Like system software version 6.0.7, versions 7.0 and 7.0.1 pick up the Component Manager via an INIT, and so again the Component Manager isn't around until after the INIT has been installed. The main difference in System 7 is that in addition to searching the System Folder and its subfolders for component files, the Component Manager will also examine the contents of any subfolders that are in the Extensions folder.

Examples of INITs in system software versions 7.0 and 7.0.1 that install the Component Manager are QuickTime, AppleScript, and Macintosh Easy Open. Note that your component can't assume that just because the Component Manager is installed, QuickTime is installed — always use the Gestalt selectors to determine what functionality is available.

The Component Manager is actually part of System 7.1 and, as a consequence, is available before the INIT process is started.

METHODS FOR MANAGING COMPONENT REGISTRATION

Now that we have a good idea of when the Component Manager is installed and where it's searching for components, let's see what we can do to make sure that our components get registered in the order we want them to be registered.

We'll use some simple components to illustrate the various methods we might use to manage component registration. In the sample code provided on the CD are three components — Moe, Larry, and Curly — that together establish a functional component hierarchy (see Figure 2). The hierarchy is such that Moe doesn't depend on any other components, Larry depends on Moe, and Curly depends on both Larry and Moe. To enforce these dependencies, we use register routines in Larry and Curly to make sure that the components they need are present before they actually allow themselves to be registered with the Component Manager. To let us know when each of these components is actually registered, Moe's register routine calls SysBeep once, Larry's calls SysBeep twice, and Curly's calls SysBeep three times. By the way, these components really don't do anything useful at all, but you probably figured that out already.

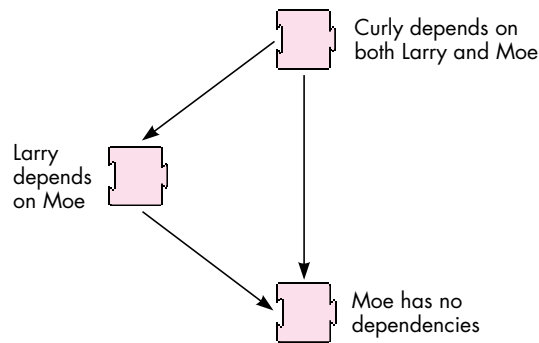


Figure 2
The Moe, Larry, and Curly Component Hierarchy

RISKY: LET THE COMPONENT MANAGER TAKE CARE OF IT

We can always simply let the Component Manager do whatever comes naturally — in this case, auto-registration. This method works only as long as you aren't picky about the order in which your components are registered. (Obviously, if your component doesn't depend in any way on the presence of other components, you're golden.) In our example scenario, though, we can't count on the Component Manager recognizing our constraints and doing the right thing. The Component Manager doesn't have enough information to know that our components have an ordering dependency (kinda reminds you of INITs, doesn't it?).

Nonetheless, let's look at what happens. The following is what occurs on my Macintosh Quadra 700 running System 7.1 and QuickTime 1.6, but you shouldn't infer that this is how the Component Manager will behave from now until eternity — there is no documentation whatsoever that provides this kind of detailed information on component registration behavior, so it *can* change.

We start with each component in a separate file. We might expect that the Component Manager would register component files in alphabetic order, and in fact this is exactly what happens. The first component that the Component Manager tries to register is Curly. However, Curly needs both Moe and Larry before it can be registered, and neither of them is present, so Curly bails. Larry comes next, and because Larry needs Moe, and Moe isn't around yet, Larry bails. Moe is last, and Moe doesn't depend on any components at all, so Moe is registered successfully. One out of three's not too good, though.

We might further expect that if we put all of our components in a single file, the Component Manager would walk the component resources from lowest resource ID to highest resource ID. If that were true, all we'd have to do is give our components ascending resource IDs in the order in which we want them to be registered (say, 200

for Moe, 300 for Larry, and 400 for Curly), and we'd be done! Well, I know we all long for the day that the omniscient System will always figure out the right thing to do regardless of how we've specified that it be done, but that day's not here yet — or, to quote KON, “It's just a computer.”

The Component Manager calls `Count1Resources` to find out how many 'thng' resources are in a file. It then iterates through these resources, using the `Get1IndResources` call. Unfortunately, there's no guarantee that the Resource Manager will index resources in the same numeric order as their corresponding resource IDs; that is, even if Moe's 'thng' resource ID is lowest (200), Moe's resource index (as maintained by the Resource Manager) may or may not be 1.

If we actually go ahead and try this (you can try this yourself with the Moe, Larry, and Curly component file on the CD, which I created by simply Rezzing the three components into a single 'thng' file), we find that we get exactly the same behavior we observed with the separate component files — first Curly fails, then Larry fails, and only Moe registers successfully. This approach just isn't reliable enough for our purposes, and we need a better mousetrap.

BETTER: USE AN INIT TO MANAGE REGISTRATION

Here's an idea — we can use an INIT to manage the registration order of our components! We'll create a resource that describes the order in which to register our components, and then the INIT can read this resource, registering the component resources in the specified order. The registration order is simply defined as the position in the component list; that is, the first component in the list is registered first, the second component in the list is registered second, and so on.

The component load order resource. We use a custom resource, called a component load order resource, to indicate to our INIT the order in which the components in the INIT file should be registered. The resource type is defined as 'thld' (for “thing load”) and the resource is a 1-based list of structures of type `ComponentLoadSpec`, as defined below:

```
#define kComponentLoadOrderResType  'thld'

typedef struct ComponentLoadSpec {
    ResType  componentResType;
    short    componentResID;
} ComponentLoadSpec, *ComponentLoadSpecPtr, **ComponentLoadSpecHdl;

typedef struct ComponentLoadList {
    short    count;
    ComponentLoadSpec spec[1];
} ComponentLoadList, *ComponentLoadListPtr, **ComponentLoadListHdl;
```

KON's pithy quote was immortalized in the
Puzzle Page in *develop* Issue 9. •

The `componentResType` field contains the component resource type, in this case 'thng', and the `componentResID` field contains the component resource ID.

The loader INIT. Our INIT — called, surprisingly enough, `LoaderINIT` — doesn't really do much work. When the INIT is executed, it checks to see whether the Shift key or mouse button is held down; if so, it quits. If not, it then checks for the presence of the Component Manager, and if the Component Manager is installed, it tries to load our components with a call to the `LoadComponents` routine.

```
main (void)
{
    KeyMap keys;

    // INIT setup for THINK C (these routines are defined in <SetupA4.h>)
    RememberA0();
    SetUpA4();

    // If mouse or Shift key down, don't bother.
    GetKeys (keys);
    if (!Button() && !(1 & keys[1])) {
        OSErr result = noErr;

        // Is the Component Manager available?
        if (HasComponentMgr()) {
            // Load the components!
            result = LoadComponents (kComponentLoadListResType,
                                     kLoaderBaseResID);
        }
    }

    // INIT cleanup for THINK C (this routine is defined in <SetupA4.h>)
    RestoreA4();
}
```

The `LoadComponents` routine. `LoadComponents` does the job of reading the component load order resource and loading each of the components it points to; this routine is shown below.

```
static OSErr
LoadComponents (ResType loadListResType, short loadListResID)
{
    OSErr    result = noErr;

    ComponentLoadListHdl componentLoadList = (ComponentLoadListHdl)
        Get1Resource (loadListResType, loadListResID);
```

```

// Did we get the component load list?
if (componentLoadList != nil) {
    ComponentLoadSpec      componentLoadSpec;
    ComponentResourceHandle componentResHdl;
    Component               componentID;
    short numComponentsToLoad = (**componentLoadList).count;
    short i;
    for (i = 0; i < numComponentsToLoad; i++) {
        // Get the component load spec.
        componentLoadSpec = (**componentLoadList).spec[i];

        // Get the component resource pointed to by this spec.
        componentResHdl = (ComponentResourceHandle) Get1Resource (
            componentLoadSpec.componentResType,
            componentLoadSpec.componentResID);

        // Did we get it?
        if (componentResHdl != nil) {
            // Register it.
            componentID = RegisterComponentResource (componentResHdl,
                kRegisterGlobally);
            if (componentID == 0L) {
                // RegisterComponentResource failed.
                result = -1L; // Return anonymous error
            }
        }
        else {
            // Get1Resource failed.
            result = ResError();
        }
    }
}
else {
    // Couldn't get component loader resource.
    result = ResError();
}
return (result);
}

```

Why it's too good to be true. LoaderINIT works fine if we're running System 7.1 or later (the Component Manager is installed *before* the INIT 31 process begins) or we name LoaderINIT something alphabetically greater than the name of the INIT that's installing the Component Manager (assuming we know this somehow). If neither of these conditions is met, LoaderINIT will execute before the Component Manager is installed, and none of our components will be registered. Bummer.

We could do something sneaky like patch a trap that we've observed being called right before the Finder comes up, and then execute our INIT code. In effect, this defers our normal INIT execution until after all other INITs load (provided they aren't pulling the same sneaky trick). However, we'd rather be more elegant and, dare I say, more compatible. We could also name our INIT ~LoaderINIT (or something similar) to guarantee that we run last in the INIT sequence (a somewhat naive hope), but we'd rather not become participants in the latest chapter of the ongoing saga of INIT Wars (Chapter XX: MacsBug Strikes Back). So what's a component developer to do?

MO' BETTER: USE A LOADER COMPONENT TO MANAGE REGISTRATION

Fortunately, we don't have to give up yet. We can avoid the shortcomings of the INIT approach by using a component to load our components — a component we'll call, oh, I don't know, something original; how about . . . a loader component.

The loader component. The loader component is a very simple component. It implements only the open, close, can do, version, and register selectors, and has no unique selectors of its own. It resides in a file of type 'thng', so the Component Manager will auto-register it. Also, the cmpWantsRegisterMessage flag is set in the componentFlags field of its component resource so the Component Manager will send it a register message at auto-register time. Our other three components (Moe, Larry, and Curly) are also included in the loader component file.

When the loader component is registered, it receives three messages from the Component Manager — open, register, and close. The register routine does all the work. It performs basically the same checks that are performed in LoaderINIT and calls the same LoadComponents routine described earlier to manually register Moe, Larry, and Curly. The loader component's register routine is shown below.

```
pascal ComponentResult
 LoaderRegister (Handle storage)
{
    KeyMap          keys;
    LoaderPrivateGlobalsHdl globals = (LoaderPrivateGlobalsHdl) storage;
    OSErr           result = noErr;

    #ifndef BUILD_LINKED
    short savedResRefNum = CurResFile();
    short compResRefNum = OpenComponentResFile ((*globals).self);

    // Use the component's resource file (not the THINK project resource
    // file) if we're running standalone.
    UseResFile (compResRefNum);
    #endif BUILD_LINKED
```

```

// If mouse or Shift key down, don't bother.
GetKeys (keys);
if (!Button() && !(1 & keys[1])) {
    // Load the components!
    result = LoadComponents (kComponentLoadListResType,
                             kLoaderBaseResID);
}

#ifdef BUILD_LINKED
// Restore the resource file (if running standalone).
CloseComponentResFile (compResRefNum);
UseResFile (savedResRefNum);
#endif BUILD_LINKED

return ((result == noErr) ? 0L : 1L);
}

```

The 'gnht' resource. Everything's pretty cool up to this point, except for one minor detail — we can't keep the component resources for Moe, Larry, and Curly as 'thng' resources in our loader component file. Why? Well, if they *are* kept as 'thng' resources, they'll be auto-registered along with the loader component, and our carefully constructed mechanism for managing registration goes right out the window! Worse, we end up trying to load our components twice — once via the Component Manager's auto-registration mechanism, and once by our own loader component!

So, we need to mildly fake out the Component Manager. We do this by keeping Moe, Larry, and Curly's component resources around as 'gnht' resources instead of 'thng' resources. The 'gnht' resource is identical to the 'thng' resource, but the Component Manager doesn't know to look for it, so Moe, Larry and Curly aren't auto-registered. The loader component (whose component resource is of type 'thng') *does* get auto-registered, and it knows where to find the component resources for Moe, Larry, and Curly because the component load order resource provides this information. Recall that in LoaderINIT, the component load specs in the component load order resource all point to resources of type 'thng'. We simply change these fields to point to resources of type 'gnht', and we're set!

PRACTICE SAFE REGISTRATION

In this article, we've looked at several approaches to installing components in a predetermined order. While you're encouraged to adapt these methods freely to fit your particular problem, keep in mind that your solution should strive to be as compatible as possible with other system extensions — your users will thank you for sparing them the frustration of renaming and removing extensions just to get your software running!

THANKS TO OUR TECHNICAL REVIEWERS

Bill Guschwan, Peter Hoddie, Casey King •



BILL GUSCHWAN

SOMEWHERE IN QUICKTIME

DYNAMIC CUSTOMIZATION OF COMPONENTS

QuickTime's component architecture lets you do a number of amazing and useful things by customizing components, which you can do by deriving one component based on another. Because QuickTime components are dynamically linked, preexisting applications can take advantage of a new, derived component without recompiling or rebooting. And because QuickTime is an extension of system software, the derived component will provide systemwide functionality.

In this column I'll describe how to use object-oriented techniques to customize components using a derived component. To illustrate, I'll show you how to customize the Apple text media handler to "speak" text tracks in movies using Apple's new Text-to-Speech Manager. You'll find the derived component on this issue's CD, along with a generic derived component that you can use as a basis for doing your own customizing. I've also supplied an application to help you debug your component. This application uses the debugging technique of registering the code inline. It's very basic and simply plays back a movie, but it gives you access to the full debugging environment of THINK C.

ABOUT THOSE OBJECT-ORIENTED TECHNIQUES

As any MacApp or class library programmer knows, there are three main steps to adding or altering functionality in an object-oriented program:

1. Identify the class responsible for the behavior you want to alter.
2. Identify the specific methods you need to add or override.
3. Create a new class derived from the original class and implement the new methods or enhance the inherited methods.

Because components can be overridden much like C++ classes, these object-oriented techniques can be applied to customizing components. (For a more in-depth comparison of components and C++ classes, see the "Be Our Guest" column in *develop* Issue 12.)

So, the three steps to customizing components are:

1. Identify the component to use as a starting point.
2. Identify the routines in the component to override.
3. Create a derived component.

Before we get into a discussion of these steps, let's drop back and look at the nuts and bolts of QuickTime component architecture with its dynamic linking capabilities. This should give you a clearer idea of how it's possible to alter QuickTime's behavior at run time.

DYNAMIC LINKING OF COMPONENTS

The QuickTime movie file format depends on the dynamic linking capabilities of the Component Manager. To play a QuickTime movie, you need more than just the movie data (video frames, digitized audio samples, text, and such): you also need a time source, code to read/write the data, and code to act on or interpret the data. It would be impractical to store all this information in each and every QuickTime movie. Instead, the time source and code are dynamically linked in as components, while the movie data remains in a QuickTime movie file.

When a movie is opened in an application like MoviePlayer, the movie file is opened first, followed by a NewMovieXXX call (such as NewMovieFromFile). The major purpose of the NewMovieXXX call is to

84

BILL GUSCHWAN (AppleLink ANGUS) hung out with Robert Schumann to discuss their symphonic feats. Robert: "Angus, I understand you compare your jobs to symphonies." Angus: "I guess so, though I'd rather compare operas to pasta. You know, Wagner is lasagna, Mozart is fettucine, Verdi is ravioli, . . ." Robert: "So on your opera pasta scale, how do you rate my symphonic music?" Angus: "Linguini." Robert: "Yo mama!" Angus: "Listen, Mr. Concerto Psycho Ward, at least my mother knows the meaning of life beyond success. Can't say the same about your

wife." Robert: "You mean my beloved Clara." Angus: "Yep, Clara, the dogcow. Well, gotta go, I hear Symphony No. 2." Robert: "Before you go, what's the key?" Angus: "C Major." Robert: "No, what's the key?" Angus: "As Wittgenstein says, the key to life is that language is a game." Robert: "No, what's the key?" Angus: "Oh, it's the key to my new office in PIE DTS." Robert: "Then let the music begin, allegro." Angus: "Pass the parmesan." •

dynamically link to all the components listed in the movie resource and return a handle to this “new” data structure.

When a `NewMovieXXX` call is made, QuickTime invokes the Component Manager to load the handlers described in the media. A clock component is loaded first (type 'clok', subtype 'micr'). Then the media handler for each track is brought in (type 'mhlr'). If you have video and sound, for example, video and sound media handlers are loaded (subtypes 'vide' and 'soun', respectively).

You may notice that the media handlers open other media handlers to do chores for them. The video and sound media handlers open the standard media handler (type 'mhlr', subtype 'mhlr'), which is a private, high-throughput media handler. The text media handler, though, opens the base media handler (type 'mhlr', subtype 'gnrc'). The base media handler is a public, general-purpose media handler with a lower throughput that is nevertheless fast enough for text.

Next a data handler is loaded. Note that at present there's only one kind of data handler (type 'dhlr', subtype 'alis') supporting streams of data from HFS files. If necessary, a decompressor component is loaded for video; its type depends on the compression format.

Thus, a media handler and a data handler are loaded for each track. QuickTime movies use data handlers and media handlers to load, interpret, and manage the movie data. The alias data handler is responsible for opening and closing data files and for reading and writing sample data. It doesn't understand the format of the data but merely hands off the data to the media handler to interpret.

The media handler is the component that's responsible for interpreting data retrieved from the data handler and for scheduling the rendering of this data at the correct time during movie playback. For example, the text media handler interprets text samples and renders the text track in the movie based on the current time value of the movie. The media handler interfaces with

the data handler using file offsets and with the rest of QuickTime through a time value. Thus, a major media handler chore is to convert time values into file offsets.

You now know how and why a QuickTime movie dynamically links with its media handlers. With that background on QuickTime component architecture behind us, we now embark on the process of customizing the text media handler to speak its text.

IN SEARCH OF A BASE COMPONENT

The first step in customizing a component is to identify the base component — the component to start with.

Not all components can be customized. There are two requirements. First, the component must implement the target request; that is, it must allow another component instance to intercept all its messages. To determine whether a particular component instance implements the target request, you can use the call

```
ComponentFunctionImplemented(myComponentInstance,  
                             kComponentTargetSelect)
```

The second requirement is that a component must have a public API before it can be inherited from. When a component is called, it's passed the routine's selector in the `ComponentParameters` structure. The component parses this selector and jumps to the appropriate function. If there's no public API, you can't know the parameters and behavior of any of the component's routines and thus can't override them. The interface file `QuickTimeComponent.h` contains the APIs for all public QuickTime components.

To speak text as it streams by, we'll want to customize the Apple text media handler, which both implements the target request and has a public API. The Apple text media handler itself is a derived media handler that uses the services of the base media handler supplied by Apple. Consequently, its interface is defined in the `MediaHandlers.h` file. (For more on the intricacies of derived media handlers, see the “Somewhere in QuickTime” column in *develop* Issue 14.)

To see information about a track in a hierarchical manner, you can use Dumpster, a QuickTime tool that's included on this issue's CD. •

You can watch the components of a movie load if you set A-trap breaks as outlined in the section “Breaking on Common Component Manager Routines” in the article “Inside QuickTime and Component-Based Managers” in *develop* Issue 13. •

IN SEARCH OF THE ROUTINES TO OVERRIDE

Our next step is to find the routines to override. In our example, one routine we need to override is the routine in the Apple text media handler where the text is rendered. In addition to rendering the text, we want to grab the text and speak it.

A media handler is normally called in response to a `MoviesTask` call. `MoviesTask` is the QuickTime workhorse routine that gives time to the media handler to get the data and render it. In turn, `MoviesTask` calls the `MediaIdle` routine to do the bulk of the processing in a media handler. `MediaIdle` is the main routine in `MediaHandlers.h`. Thus, `MediaIdle` is the main routine we want to override. Additionally, we'll need to override the `MediaInitialize` routine, which supplies us with an initial reference to the media.

CREATING A DERIVED COMPONENT

So far we've chosen a base component from which to derive our customized component, and we've identified the routines we want to override. Now we're ready to take the third step of writing a new component that targets the base component and overrides the identified routines. If you're curious about the design of the generic derived component, you can investigate it on the CD. I'm only going to point out a couple of things about its design before moving on to discuss what you need to do to make your own derived component.

To capture or not to capture. You have two possible approaches when deriving a component. First, you can simply open and target a component, which lets your component use the services of that component. The component is still available to other clients, but you're using an instance of it. Second, in addition to targeting the component, you can capture it. The base component will then be replaced by your component in the component registration list and will no longer be available to clients (although current connections are retained). The `CaptureComponent` routine returns a special ID so that the captured component can still be used by your component.

We'll use `CaptureComponent` because we want to replace the functionality of all instances of the text media handler (conceptually, you can think of capturing as patching). However, targeting without capturing is just as effective — and it has a few advantages: it doesn't require you to keep track of the captured component's ID, and it allows clients looking for a specific component to be successful.

Let's walk through the steps you'd take to make your own derived component using the generic code on this issue's CD, which are the same steps used to create our text-speaking example. You need to make changes in specific places: the component resource, the global data file, the `OpenComponent` routine, and the override routines.

Changing the component resource. The first thing to change is the resource file for the component. The essential part of this file is the component description, which is a structure that describes the component type, subtype, manufacturer, and flags. The Component Manager looks at this information when it's handling an application's request for a component. You want the right information here so that QuickTime will grab your derived component instead of the base component.

You should change the component type and subtype to match those of the component you're inheriting from. In our example, when a QuickTime movie with a text track is opened, QuickTime asks the Component Manager for a text media handler, which has type `'mhlr'` and subtype `'text'`. Since we want QuickTime to grab our derived component instead, we need to make its type and subtype the same.

In our case, we have to change the component manufacturer to match that of the base component as well. This isn't the ideal situation, because it would be most desirable for each component to have a unique manufacturer. But clients may look for a component of a specific manufacturer and won't grab your derived component if its manufacturer is different. Because it would be better to be able to identify a derived

86

QuickTime 1.6 adds a new Component Manager selector, `componentWantsUnregister`, that you can take advantage of when you want to free a captured component. Set `componentWantsUnregister` in the `componentRegisterFlags` field. When the captured component is unregistered, your derived component can call `UncaptureComponent` and dispose of the global memory. •

To identify a captured component in a debugger, you can use the **thing** dcmd. The component ID of a captured component will begin with two periods (`..`). •

component, it's strongly suggested that component clients always perform a default search, avoiding asking for specifics other than type and subtype.

You may also need to set the `componentFlags` field, which identifies specific functionality for a component. For example, video digitizers use `componentFlags` to identify the type of clipping the digitizer performs, among other things.

If you don't know how a client searches for a component, you can find out by running that application and trapping on `FindNextComponent`. The last parameter pushed on the stack is the component description, and you can find its values in a debugger (see "Inside QuickTime and Component-Based Managers" in *develop* Issue 13). In our example, we know that QuickTime performs a simple type and subtype search for a text media handler, so we only have to change the type and subtype in the component resource.

The global data file. The `ComponentData.h` file contains the declaration of the data structure for each component instance and the global component data structure. You'll need to fill out a component description structure describing your chosen base component, which will be used to ask the Component Manager to find it.

Now you're left with defining the global data for your derived component. The generic capturing component on this issue's CD has one item that's shared across all its instances: a reference to the captured component. If you need data that's shared across instances, declare it here, but in general you shouldn't need it.

The data local to each instance is allocated in the `OpenComponent` routine. By default, three component instances will be kept track of: a self copy, a delegate copy, and a target copy. These instances will be stored for you, and you won't need to do any work. The target copy is the instance of a component that may capture yours. If your component calls itself, it should use this instance in case the target overrides the routine.

The other data that you allocate is specific to the type of your derived component. For our example, we'll allocate room for a speech channel, a media reference, a handle to the text to be spoken, and a `StdTTSPParams` structure, which is filled out by the Standard Text to Speech dialog. This dialog lets the user choose voice, pitch, and modulation.

The `OpenComponent` routine. `OpenComponent` performs three major operations. First, it allocates storage for each instance. Second, it checks for QuickTime, the Text-to-Speech Manager, and other dependencies; if they're not installed, the component can't open and an error is returned. Note that software dependencies are checked here instead of in `RegisterComponent` to bypass possible load order conflicts. Finally, `OpenComponent` captures the Apple text media handler and stores a reference to it in the component globals.

The `override` routines. Now it's time to implement the `override` routines. You'll need to get the selectors for the routines from the original component's header file.

In our example, we look at `MediaHandlers.h` and find the `MediaInitialize` routine. The selector has a constant, `kMediaInitializeSelect`. We need to make the parameters of our `override` routine match those of the `MediaInitialize` routine.

```
pascal ComponentResult MediaInitialize
    (PrivateGlobals **storage,
     GetMovieCompleteParams *gmc)
```

`MediaInitialize` performs these tasks: it stores the media reference from the `GetMovieCompleteParams` structure in our private storage; it queries the user for a voice with the Standard Text to Speech dialog; and, with this information, it allocates and sets up the speech channel.

Next we implement the `MediaIdle` routine, which has a selector of `kMediaIdleSelect`. Our `MediaIdle` looks like this:


```
pascal ComponentResult MediaIdle
    (PrivateGlobals **storage, TimeValue
     atMediaTime, long flagsIn, long *flagsOut,
     const TimeRecord *movieTime)
```

This routine retrieves the media sample for the time passed in and then speaks it. The important parameter is `atMediaTime`, which contains the current time value of the media for the movie. We get the media sample for that time using `GetMediaSample`, and then we use the nifty new Text-to-Speech Manager to speak.

In this case, we'll use `SpeakText`, which takes three parameters: a `speechChannel` (allocated earlier in the `OpenComponent` routine), a pointer to the beginning of the text that we want to speak, and the length of the text. `SpeakText` is an asynchronous routine, so it won't hold up processing (or the movie) while it speaks. On the other hand, the text can't be disposed of until speaking is finished. To accommodate this requirement, a reference to the text is stored in our instance storage, and the text is disposed of when the component closes.

LETTING USERS LINK AND UNLINK COMPONENTS

Thanks to dynamic linking, a large number of users can easily take advantage of new functionality provided by customized components. Three methods can be used to register and unregister components. First, a component is registered at system startup if the component resides in the System Folder, or in the Extensions folder of the System Folder. To unregister this component, a user can remove it and reboot. Second, an application can dynamically register components as needed, and then unregister them when finished. Third, you can use the

drag-and-drop applications *Komponent Killer* and *Reinstaller II* included on this issue's CD. Using these applications, you don't have to reboot. (Of course, your typical user won't do it this way; this method is for you, the programmer.)

EXCITING PROSPECTS

Now you know how to customize a component using a derived component, which will be dynamically linked at run time and thus can extend systemwide functionality. Just think of the possibilities! You can override the movie controller and implement an Apple event handler. And you can override other base components as well. Fiddle around with the generic derived component on the CD to get an idea of the exciting prospects before you.

REFERENCES

- "Somewhere in QuickTime: Derived Media Handlers" by John Wang, *develop* Issue 14.
- "Inside QuickTime and Component-Based Managers" by Bill Guschwan, *develop* Issue 13.
- "Techniques for Writing and Debugging Components" by Gary Woodcock and Casey King, *develop* Issue 12.
- "Be Our Guest: Components and C++ Classes Compared" by David Van Brink, *develop* Issue 12.

FLOATING WINDOWS: KEEPING AFLOAT IN THE WINDOW MANAGER

These days having floating windows in an application is like having an air bag in a car; you're not cool if you don't have at least one. Because system software doesn't support floating windows in the Window Manager, myriad floating window implementations abound, ranging from the straightforward to the twisted. This article presents a library of routines providing standard, predictable floating window behavior that most applications can readily use.



DEAN YU

Floating windows are windows that stay in front of document windows and provide the user easy access to an application's tools and controls. Ever since the introduction of HyperCard, most Macintosh programmers have been in love with floating palettes and frequently use them. This would be fine if there were an official way to implement floating windows, but there is no such beast. This article offers a solution.

Currently, the most popular way of implementing floating windows is to patch various Window Manager routines so that they behave correctly when floating windows are present. But patching traps has always been problematical. Patches often make assumptions about how a particular routine behaves or when it will be called. If system software or a third-party extension suddenly uses the patched routine where it has never been used before, compatibility problems can arise. Often, patches subtly alter the behavior of a routine — for example, by using a register or setting a condition code. This makes it difficult for Apple to extend (or even fix!) the Macintosh API and still maintain a high level of compatibility with existing applications.

You can just as easily implement floating windows by avoiding the use of high-level Window Manager routines that generate activate events; instead, you can use lower-level routines almost exclusively. It's much less likely that the code will break (or cause other code to break) when Apple makes changes to the system software. The reason for this is simple: it's much less likely for system software engineers to change the fundamental behavior of a Macintosh Toolbox routine than it is for them to use that

DEAN YU subscribes to the Taoist philosophy that that which is meant to happen, eventually will happen. In fact, he believes this is the only reason he landed his job as a Blue Meanie in the first place. During his two-and-a-half year stint in the System Software group, Dean has worked on System 7 and System 7.1 and has gone to Cancún. Apple's Rumor Monger claims that Dean left Apple in January to avoid working on a

project that installs 69 files in the System Folder, and that, more recently, he went to Las Vegas and married a Marilyn Monroe look-alike. Dean denies these rumors, of course, but not very vehemently. •

routine in some new and different way. Under this second implementation method, the application becomes a proper client of the Toolbox, using the routines that are available rather than trying to reengineer them. The floating windows library described in this article and provided on this issue's CD follows this philosophy.

STANDARD FLOATING WINDOW BEHAVIOR

Developers implementing floating windows should follow certain rules to ensure the “consistent user experience” that we're always harping about. Don't worry if there seem to be a lot of things to keep in mind; the routines in the library do most of the hard work for you.

ORDER OF ON-SCREEN INTERFACE OBJECTS

As more and more things appear on users' screens, it becomes very important to define a front-to-back order in which interface objects appear. This alleviates confusion and prevents the neophyte user from being scared away when things start flying thick and fast on the screen. Within an application, the order of windows and other on-screen objects from back to front should be as follows (see Figure 1):

- Document windows and modeless dialogs
- Floating windows
- Modal dialogs and alerts
- System windows
- Menus
- Help balloons

If you thought that floating windows would be as far back as they are, you get a gold star. The rationale for putting modal dialogs in front of floating windows stems from the normal use of these windows: floating windows are most frequently used as tool palettes. The user picks a tool, color, or something similar from the palette and then performs an operation on the active document. When a modal dialog appears, the application needs more information from the user before it can proceed. The tools in the floating window should not be available because they can't be used in the dialog.

Incidentally, system windows are windows that can appear in an application's window list but aren't directly created by the application. These windows appear in front of all windows created by the application. Examples of system windows include notification dialogs, the PPC Browser, and input method windows.

APPEARANCE OF FLOATING WINDOWS

The physical appearance of the HyperCard floating palette has become the de facto standard look for floating windows. The description of floating windows that follows

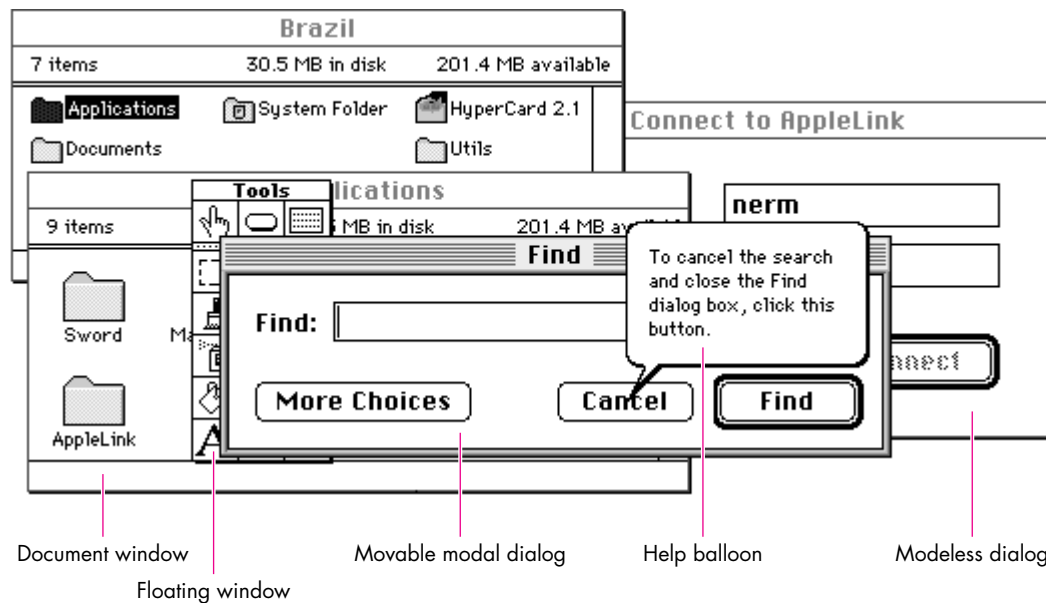


Figure 1
Order of Windows on a Screen

is based on this look. There's at least one popular program that uses the standard document window as a floating window. Don't do this; it only confuses the novice user.

Unlike document windows, floating windows are all peers of each other. That is, there's no visual cue to the user of any front-to-back order unless the floating windows actually overlap each other; they all float at the same level. Because of this equality, the title bars of floating windows are almost always in an active state. The exception to this rule occurs when a modal window is presented to the user; since this type of window appears above floating windows on the screen, the background of the title bar of each visible floating window turns from its dotted pattern to white to indicate an inactive state (see Figure 2).

A floating window can have a close box, a zoom box, and a title. The use of size boxes in floating windows is not recommended. The title bar of a floating window should be 11 pixels high or 2 pixels higher than the minimum height of the primary script's application font, whichever is greater. The title of a floating window should be in the application font, bold, and its size should be the greater of 9 points and the smallest recommended point size for that script system. Floating windows should have a 1-pixel drop shadow that starts 2 pixels from the left and top of the window.

the frontmost document window. If things were left up to the Window Manager, only the frontmost floating window would receive the required deactivate event.

To avoid this problem, you shouldn't use the Window Manager routines `SelectWindow`, `ShowWindow`, and `HideWindow` since they implicitly generate activate and deactivate events. In addition, you shouldn't use `SendBehind` to move the front window further back in the pile of windows on the screen or to make a window frontmost, because that routine also generates activate events.

Instead, use lower-level routines like `BringToFront`, `ShowHide`, and `HiliteWindow` to simulate the higher-level calls. Additionally, instead of dispatching activate events in your application's main event loop, you should activate or deactivate a window as its position in the window list changes. Here's how a replacement to `SelectWindow` might look (see "This Is Not Your Father's Window Manager" for more information on this routine):

```
pascal void SelectReferencedWindow(WindowRef windowToSelect)
{
    WindowRef      currentFrontWindow;
    WindowRef      lastFloatingWindow;
    ActivateHandlerUPP activateProc;
    Boolean         isFloatingWindow;

    if (GetWindowKind(windowToSelect) == kApplicationFloaterKind) {
        isFloatingWindow = true;
        currentFrontWindow = (WindowRef) FrontWindow();
    }
    else {
        isFloatingWindow = false;
        currentFrontWindow = FrontNonFloatingWindow();
        lastFloatingWindow = LastFloatingWindow();
    }

    // Be fast (and lazy) and do nothing if you don't have to.
    if (currentFrontWindow != windowToSelect) {

        // Selecting floating windows is easy, since they're always active.
        if (isFloatingWindow)
            BringToFront((WindowPtr) windowToSelect);
        else {

            // If there are no floating windows, call SelectWindow as in the
            // good ol' days.
            if (lastFloatingWindow == nil)
                SelectWindow((WindowPtr) windowToSelect);
        }
    }
}
```

```

else {

    // Get the activate event handler for the window currently in
    // front.
    activateProc = GetActivateHandlerProc(currentFrontWindow);

    // Unhighlight it.
    HiliteWindow((WindowPtr) currentFrontWindow, false);

    // Call the activate handler for this window to deactivate the
    // window.
    if (activateProc != nil)
        CallActivateHandlerProc(activateProc,
                                uppActivateHandlerProcInfo, currentFrontWindow,
                                kDeactivateWindow);

    // Get the activate event handler for the window that's being
    // brought to the front.
    activateProc = GetActivateHandlerProc(windowToSelect);

    // Bring it behind the last floating window and highlight it.
    // Note that Inside Macintosh Volume I states that you need to
    // call PaintOne and CalcVis on a window if you're using
    // SendBehind to bring it closer to the front. In System 7,
    // this is no longer necessary.
    SendBehind((WindowPtr) windowToSelect,
               (WindowPtr) lastFloatingWindow);
    HiliteWindow((WindowPtr) windowToSelect, true);

    // Now call the window's activate event handler.
    if (activateProc != nil)
        CallActivateHandlerProc(activateProc,
                                uppActivateHandlerProcInfo, windowToSelect,
                                kActivateWindow);
    }
}
}
}

```

Activate events and the frontmost document window. Other cases that the Window Manager doesn't handle well occur when the frontmost document window is closed or when a new document window is created in front of other document windows. If floating windows are present, these document windows don't get the needed activate and deactivate events, since the application is essentially removing or creating windows in the middle of the window list. Your application needs to send the

THIS IS NOT YOUR FATHER'S WINDOW MANAGER

You may have noticed that the `SelectReferencedWindow` routine doesn't strictly define how to do certain things. There are two reasons for this. The first is the advent of PowerPC architecture. When you write code that has the potential of running on several different runtime architectures, it should be generic, especially if you don't know what's lurking on the other side of a procedure pointer. The 68000 and PowerPC architectures handle procedure pointers differently: on a 680x0 machine, a `ProcPtr` points to the entry point of a procedure, whereas on a PowerPC, a `ProcPtr` points to a routine descriptor. It would be nice if source code that calls procedure pointers didn't have to worry about the proper calling convention for a particular platform and the proper magic would happen at the flip of a compile switch. The solution that we use in system software is the `CallProcPtr` macros defined in our interface files, which expand to different things depending on the platform we're compiling for. For

the `ActivateHandlerUPP` (for **U**niversal **P**rocedure **P**ointer) type used in `SelectReferencedWindow`, the definitions shown below are needed.

The second reason for generality in the code is the future. We would like to move the Macintosh operating system into the 1990s to get preemptive multitasking and separate address spaces. This means a move toward opaque data structures: accessor functions will be provided, so you won't be able to access fields of a data structure directly. In the future, data structures like `WindowRecords` may no longer be created in your application's address space, so you'll get a reference to a window instead of an absolute address. The floating window API follows this philosophy; all calls take a `WindowRef` type instead of a `WindowPtr`, and all fields of a window's data structure are accessed with an accessor function. This is all for the best. Really.

```
typedef pascal void (*ActivateHandlerProcPtr)(WindowRef theWindow, Boolean activateWindow);
enum {
    uppActivateHandlerProcInfo = kPascalStackBased | kParam1FourByteCode | kParam2TwoByteCode
};

#if USES68KINLINES
typedef ActivateHandlerProcPtr ActivateHandlerUPP;
#pragma parameter CallActivateHandlerProc(__A0)
pascal void CallActivateHandlerProc(ActivateHandlerUPP activateHandler, WindowRef theWindow,
                                   Boolean activateWindow) = 0x4E90; //jsr (A0)
#define CallActivateHandlerProc(activateHandler, activateHandlerProcInfo, theWindow, \
                                activateWindow) \
    CallActivateHandlerProc(activateHandler, theWindow, activateWindow)
#else
typedef UniversalProcPtr ActivateHandlerUPP;
#define CallActivateHandlerProc(activateHandler, activateHandlerProcInfo, theWindow, \
                                activateWindow) \
    CallUniversalProc(activateHandler, activateHandlerProcInfo, theWindow, activateWindow)
#endif
```


right activate events to the right windows. The floating windows library routines `ShowReferencedWindow` and `HideReferencedWindow` generate the appropriate activate and deactivate events for you.

Activate events and modal windows. When a modal window is to appear, you should send deactivate events to all visible floating windows and to the active document window. When the user dismisses the modal window, send activate events to those windows. Instead of overloading `SelectReferencedWindow` with yet another case, it's easier to surround calls to `Alert` or `ModalDialog` with calls to deactivate and activate the floating windows and the first document window.

Here's what the code would look like:

```
short PresentAlert(short alertID, ModalFilterProcPtr filterProc)
{
    short alertResult;

    DeactivateFloatersAndFirstDocumentWindow();
    alertResult = Alert(alertID, filterProc);
    ActivateFloatersAndFirstDocumentWindow();

    return alertResult;
}
```

THE FLOATING WINDOW API

The floating windows library supplies the routines and accessor functions described below. Each routine description tells how to use it in an application and, when necessary, describes its parameters in detail.

The floating window API uses the `WindowRef` type in the place of a `WindowPtr`. This is in anticipation of the situation in which memory for a window's data structure is no longer allocated in the application's address space. (See "This Is Not Your Father's Window Manager.") At present, a `WindowRef` is interchangeable with a `WindowPtr`, and a parameter of type `WindowRef` can be passed to existing Window Manager routines. A typecast is needed because a `WindowRef` points to a structure that contains a `WindowRecord` plus other fields.

Creating and disposing of windows. The routines described in this section — `NewWindowReference`, `GetNewWindowReference`, and `DisposeWindowReference` — should be used instead of `NewWindow`, `GetNewWindow`, and `DisposeWindow`. You can use these new routines for any type of window, not just floating windows. Note that you should use them together; for example, `DisposeWindowReference` should be used to dispose of any windows created by `NewWindowReference` or `GetNewWindowReference`.

```
pascal OSErr NewWindowReference(WindowRef *windowReference, const Rect
    *boundsRect, ConstStr255Param title, Boolean visible, WindowAttributes
    attributes, WindowRef behind, long refCon, ActivateHandlerUPP
    activateHandler);
```

NewWindowReference creates a floating window, document window, or dialog window. On machines with Color QuickDraw, it creates a color window; on machines without Color QuickDraw, it creates a window with a black-and-white grafPort.

The windowReference parameter returns a reference to the new window. If a window could not be created, nil is returned. The boundsRect, title, visible, and refCon parameters are identical to the parameters you would normally pass to NewWindow or NewCWindow.

The behind parameter specifies the window that the new window should be created behind. It's similar to the behind parameter that's passed to NewWindow, except that -1 has the following special meaning: if a floating window is being created, -1 means the new window will be created in front of all other windows; if a document window is being created, -1 means the new window will be created behind any existing floating windows.

Unlike NewWindow, which establishes an appropriate WDEF resource based on the window definition ID passed as a parameter, NewWindowReference establishes an appropriate window definition function based on the attributes parameter, which describes the desired physical attributes. The following values have been defined for the attributes parameter:

```
enum {
    kHasCloseBoxMask =          0x00000001,
    kHasZoomBoxMask =          0x00000002,
    kHasGrowBoxMask =          0x00000004,
    kHasModalBorderMask =      0x00000010,
    kHasThickDropShadow =      0x00000020,
    kHasDocumentTitlebarMask = 0x00001000,
    kHasPaletteTitlebarMask =  0x00002000,
    kHasRoundedTitlebarMask =  0x00004000,

    // Attribute groupings

    kWindowGadgetsMask =       0x0000000F,
    kWindowAdornmentsMask =    0x00000FF0,
    kWindowTitlebarMask =      0x000FF000,

};
typedef unsigned long WindowAttributes;
```

The values of the attributes parameter can be combined, except only one title bar value can be used. For example, `kHasCloseBoxMask + kHasZoomBoxMask + kHasGrowBoxMask + kHasDocumentTitlebarMask` describes the appearance of a standard document window.

Finally, the `activateHandler` parameter is a pointer to the routine that's called whenever the window is activated or deactivated. You should always supply this routine, because the main event loop doesn't receive activate events when the floating windows library is used. Activate event handlers have the following prototype:

```
pascal void (*ActivateHandlerProcPtr) (WindowRef theWindow, Boolean
    activateWindow);
```

The `theWindow` parameter is the window that should be activated or deactivated. The `activateWindow` parameter specifies whether the window should be activated or deactivated: `true` means activate, `false` means deactivate.

`NewWindowReference` can return the following errors:

- `kUndefinedTitlebarTypeError`: Invalid values in the attributes parameter, or more than one title bar attribute is specified in the attributes parameter.
- `kWindowNotCreatedError`: Not enough memory to create the window.
- `kInvalidWindowOrderingError`: The `behind` parameter specifies creating a floating window behind an existing document window, or a document window in front of a floating window.

```
pascal OSErr GetNewWindowReference(WindowRef *windowReference, short
    windowResourceID, WindowRef behind, ActivateHandlerUPP
    activateHandler);
```

`GetNewWindowReference` creates a window based on a resource template. On machines with Color QuickDraw, it creates a color window. On machines without Color QuickDraw, it creates a window with a black-and-white `grafPort`.

The `windowReference` parameter returns a reference to the new window. If a window could not be created, `nil` is returned. The `windowResourceID` parameter is the resource ID of the `WIND` resource that describes the window. The visible field in the `WIND` resource should be `false`.

The `behind` parameter specifies the window that the new window should be created behind. As in `NewWindowReference`, if `-1` is specified for this parameter, document windows are created behind any existing floating windows.

GetNewWindowReference can return the following errors:

- `kWindowNotCreatedError`: Not enough memory to create the window, or the specified WIND resource cannot be found.
- `kInvalidWindowOrderingError`: The meaning of this is the same as for `NewWindowReference`.

```
pascal void DisposeWindowReference(WindowRef windowReference);
```

`DisposeWindowReference` frees the memory used by a window created with `NewWindowReference` or `GetNewWindowReference`.

Displaying windows. The routines described in this section affect how windows look and how they're ordered on the screen.

```
pascal void SelectReferencedWindow(WindowRef windowToSelect);
```

`SelectReferencedWindow` replaces `SelectWindow`; it brings a window as far forward as it should come when the user clicks in it. Selecting a floating window makes it the absolute frontmost window on the screen. Selecting a document window makes it the frontmost document window, but it remains behind all floating windows.

```
pascal void HideReferencedWindow(WindowRef windowToHide);
```

`HideReferencedWindow` replaces `HideWindow` to hide a window. As in `HideWindow`, if the frontmost window is hidden, it's placed behind the window immediately behind it, so when it's shown again, it will no longer be frontmost. This is also true for document windows even if floating windows are visible.

```
pascal void ShowReferencedWindow(WindowRef windowToShow);
```

This routine replaces `ShowWindow` to make a hidden window visible again. If the window is frontmost when it's shown, the previously active window is deactivated.

```
pascal void DeactivateFloatersAndFirstDocumentWindow(void);
```

Before presenting a modal window to the user, applications should call this routine to unhighlight any visible floating windows and the frontmost document window, and to send deactivate events to these windows. At this point, all visible windows in the window list can be treated as normal windows, and the modal dialog or alert can be brought up with the traditional calls.

```
pascal void ActivateFloatersAndFirstDocumentWindow(void);
```

After the user dismisses a modal window, the application should call `ActivateFloatersAndFirstDocumentWindow` to restore the highlight state of any

visible floating windows and the frontmost document window. This routine also sends an activate event for each of these windows. When called in the background, `ActivateFloatersAndFirstDocumentWindow` hides any visible floating windows by calling `SuspendFloatingWindows`.

```
pascal void SuspendFloatingWindows(void);
```

When an application with visible floating windows receives a suspend event, it should call `SuspendFloatingWindows` to hide its floating windows. This routine remembers the current visibility of a floating window so that only the current visible floating windows are revealed on a subsequent call to `ResumeFloatingWindows`. If a movable modal dialog is frontmost when this routine is called, floating windows are not hidden because the application is in a modal state. However, if the dialog goes away while the application is in the background, the floating windows will be hidden automatically because `ActivateFloatersAndFirstDocumentWindow` calls `SuspendFloatingWindows`.

```
pascal void ResumeFloatingWindows(void);
```

Applications should call `ResumeFloatingWindows` when a resume event is received. Any floating windows that were visible when `SuspendFloatingWindows` was called are made visible again. `ResumeFloatingWindows` also activates the frontmost document window.

Utility routines. These routines provide all the other functions an application might need to operate smoothly with floating windows.

```
pascal ActivateHandlerUPP GetActivateHandlerProc(WindowRef theWindow);
```

`GetActivateHandlerProc` returns a pointer to the routine that handles activate and deactivate events for the specified window. If the window doesn't have a handler routine, `GetActivateHandlerProc` returns nil.

```
pascal void SetActivateHandlerProc(WindowRef theWindow, ActivateHandlerUPP  
    activateHandlerProc);
```

`SetActivateHandlerProc` sets a new routine to handle activate and deactivate events for the specified window. It replaces any existing handler routine for this window.

```
pascal void DragReferencedWindow(WindowRef windowToDrag, Point startPoint,  
    const Rect *draggingBounds);
```

`DragReferencedWindow` drags a window around, ensuring that document windows stay behind floating windows. Like `DragWindow`, `DragReferencedWindow` doesn't bring a window forward if the Command key is held down during the drag.

```
pascal WindowRef FrontNonFloatingWindow(void);
```

FrontNonFloatingWindow returns a reference to the first visible window that's not a floating window. Usually, this is the first visible document window. However, if a modal dialog is visible, it returns a reference to the dialog window.

```
pascal WindowRef LastFloatingWindow(void);
```

LastFloatingWindow returns a reference to the floating window that's furthest back in the window list, whether it's visible or not. Normally, the floating windows library uses this routine internally, although applications can use it to determine where the floating window section of the window list ends. If there are no floating windows in the window list, LastFloatingWindow returns nil.

Hangin' with the Get/Setters. In an effort to move toward more flexible and architecture-independent data structures, the library includes routines that get and set several WindowRecord fields. The library supplies the accessor functions only for the fields the floating window routines need to get at, however. The ambitious reader can also create accessor functions for the other WindowRecord fields that aren't provided by the floating windows library. Accessor functions have been provided for these fields: windowKind, visible, hilited, strucRgn, and nextWindow.

THE SAMPLE PROGRAM AND THE SOURCE

On this issue's CD, there's a floating windows program that doesn't do much more than exercise the routines from the floating windows library. It shows how floating windows interact with other types of windows, including alerts, movable modal dialogs, and document windows. The floating window definition procedure on the CD is taken from one of the many game programs I've never finished; it works well enough for demonstration purposes, although anyone can write a better one.

The complete MPW C source for the floating windows library is in the files WindowExtensions.c and WindowExtensions.h. This code was written so that most applications could start using the routines with a minimum of effort. (You may have to change the resource ID of the floating window WDEF that's defined in WindowExtensions.h.) Just remember that your mileage may vary.

SEAT CUSHIONS AND OTHER FLOTATION DEVICES

OK, I admit it. I did have a private agenda when I set out to write this article. The way I figure it, for every application developer I convince to implement floating windows without patching traps, I save myself a few hours in MacsBug. The most compelling argument I could think of was to write the code for the floating windows library routines so that no one else would have to. If you use the supplied library routines, you don't have to worry about any of the details on how floating windows

behave, and you can concentrate on making your applications the envy of all your friends who use Windows.

The floating windows library described in this article isn't the be-all and end-all of floating windows. The THINK Class Library and MacApp — as well as AppsToGo in the Sample Code folder on the CD — provide support for floating windows within an entire application framework. The floating windows library presented here has the advantage of being a standalone library that can be linked into your home-grown application. For the stout of heart who want to implement their own floating windows, this article lays out a road map of the gotchas and pitfalls in creating floating windows on top of the Window Manager.

I touched briefly on making source code more platform independent. As Apple takes the Macintosh experience cross-platform, there's a big potential for source code maintenance to become hellish as different machine architectures create subtle differences in the runtime environment. By factoring out assumptions about data structures and the underlying chip architecture, your applications can move cross-platform more quickly and less expensively.

The API of the floating windows library hints at what the Toolbox will look like in a few years time. While I can't predict when the Window Manager will finally support floating windows, use of the API described in this article will make for a smoother transition when that day finally comes.

THANKS TO OUR TECHNICAL REVIEWERS

C. K. Haun, Nick Kledzick, Kevin MacDonell, Eric Soldan •

WORKING IN THE THIRD DIMENSION

Macintosh users have a lifetime of experience seeing and manipulating objects in three dimensions. Many developers are taking advantage of this experience by adding three-dimensional elements to their human interfaces. 3-D effects can heighten the ease of use, realism, and visual appeal of your application. This article discusses why 3-D effects add value to Macintosh applications and describes an easy way to add 3-D effects to applications created with MacApp.



**JAMIE OSBORNE AND
DEANNA THOMAS**

Three-dimensional effects can add life to any user interface. A 3-D interface is inviting to users. It offers them tactility and can make the user interface elements they work with stand out. When implemented correctly, the 3-D interface helps users differentiate between the important contents of a window and the background. The result is a friendlier, more accessible interface to your application.

This article tells you about some basic 3-D design principles that you can use and describes one way to implement them. The accompanying code on this issue's CD contains a set of adorners and classes that make it easy to bring your MacApp applications into the third dimension.

WHY A 3-D LOOK?

Developers have been adding color and 3-D buttons to their products for a couple of years now. Users like 3-D effects not only because they're nicer to look at but also because they help define user interface elements within the workspace. There's a clear message that both users and developers want 3-D effects, and developers aren't holding off until Apple provides guidelines and tools to implement them.

We've seen many developers come up with their own implementations of gray windows and 3-D buttons. Even within Apple there are several ideas about what a 3-D user interface should look like. Although this article doesn't present an official Apple 3-D interface, it does describe designs and approaches that we developed while

JAMIE OSBORNE (AppleLink JWO) is a best-selling novelist stuck inside a software engineer's body. When he isn't working on secret decoder rings in Apple's Enterprise Systems Division, he can be found watching *Star Trek* with his friends Darmok and Jilad, and his kitten. Before coming to Apple, he was an undergraduate at Dartmouth College, where he discovered and was

subsequently sucked into the world of Macintosh programming. He has yet to escape. •

implementing a 3-D interface and that we think can help you with your own 3-D interface design.

THE 3-D LOOK

In our three-dimensional look, windows have a light gray background instead of the standard white background. Darker shades of gray and white on the edges of user interface elements give the illusion that some elements are chiseled into the background while others appear in bas-relief.

There are two special cases: Modal dialog boxes (windows with the definition ID of `dBoxProc`) have a gray background, but do not have the chiseled effect because the color version of the window already has a 3-D border. Scrollable document windows (definition ID of `documentProc`) do not have a gray background at all.

The 3-D look offers the user two advantages. First, it creates a clearer work environment. White interface elements, such as text fields, checkboxes, and pop-up menus, stand out from the light gray background and call attention to themselves. The chiseled appearance, though subtle and unobtrusive, effectively communicates the division of elements within the window.

The second advantage to the 3-D look is that it lets the user work on a more tangible, tactile surface. The chiseled look gives strong clues that invite interaction. The slight depression of text fields, for example, invites the user “into” the field to edit text. The shading around buttons makes them look as if they project slightly from the gray background. This makes buttons appear pressable, and they react appropriately when the user clicks them. Our implementation of these elements strives to maintain the crisp, clean graphical elegance that the Macintosh is known for.

The background gray we use for windows is the lightest gray in the Macintosh palette. Its RGB value is 61166, 61166, 61166 (hex 0xEEEE). In small areas, this gray is subtle; it appears to be darker in large areas that have only a few text fields or other graphical elements.

Chiseled lines define the edges of windows, text fields, and checkboxes. They also help group related items within a window. To create a chiseled effect on the light gray background, use white and the fourth gray value from the Macintosh palette. The RGB value of this medium light gray is 43690, 43690, 43690 (hex 0xAAAA). The way you use these colors determines whether an object comes toward the user or recedes into the background.

To chisel an object into the gray background, use gray shading on the object’s top and left edges and white on its bottom and right edges. To make an object project from the gray background, reverse the order: use white on the top and left edges and gray on the bottom and right edges. Note that the color of the top and left edges, whether

white or gray, always extends to the far corner pixels. Figure 1 shows both kinds of shading.



Figure 1
Shading Items for 3-D Effects

If you keep in mind that the imaginary light source on the desktop comes from the upper left corner of the screen, you can always determine where to place the highlight and shadow colors.

In windows, the background itself should appear to come forward by one pixel. To create this effect, draw a white line on the top and left edges of the window and a gray line on the bottom and right edges. Figure 2 shows a window with the background drawn this way and a scrolling list chiseled into the background.

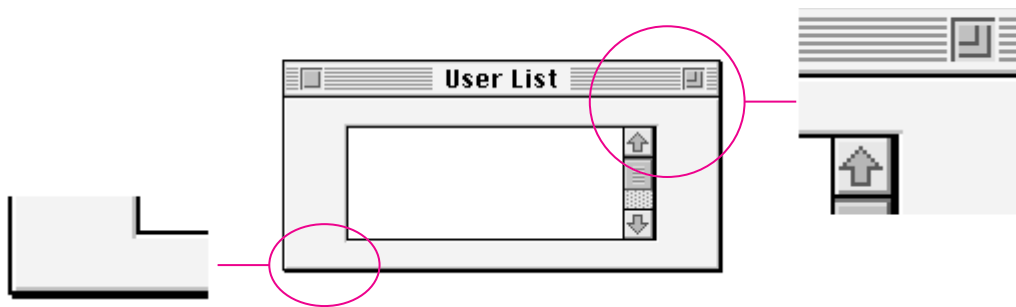


Figure 2
A Window With 3-D Effects

Though these chiseled effects are subtle, they do give a sense of depth, even if it's almost subliminal in some cases. Remember, we don't want to hit anyone over the head with these graphical enhancements. The goal is to add elegance and ease of use to the work environment, not to scream out "Look at me!"

ICON BUTTONS

Icon buttons are buttons that use graphics to describe what they do. The page orientation controls in the standard Page Setup dialog box are good examples of early

icon buttons. One of the great things about icon buttons is that they give pictorial clues at a glance. The user can usually understand what the buttons do from the way they're clustered and from their surrounding context. Another advantage of icon buttons is that they're more accessible and easier to see than menu commands. This makes icon buttons ideal for frequently used commands.

It's very easy to create poorly designed icons. If you decide to use icon buttons in your application, take the extra time to do plenty of user testing to make sure your users think your icons mean what you think they mean. This is especially important if you decide not to use text labels for your buttons.

Our icon buttons are square or rectangular, and the button is a slightly darker shade of gray so that it's easily distinguished from the light gray background. The illusion of height invites the user to press a button. Figure 3 shows a few examples of icon buttons.

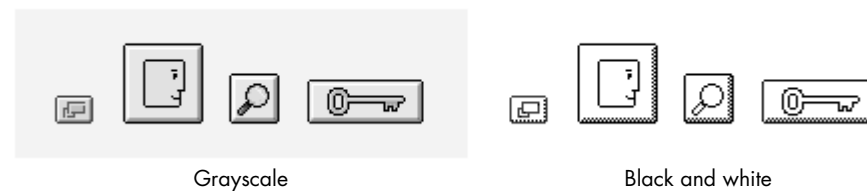


Figure 3
Some Icon Buttons

Because icon buttons are dynamic elements with specific characteristics, they need to maintain a distinct appearance even in black and white, just as checkboxes and radio buttons do. On a black-and-white screen, we use a 50% dither pattern along the bottom and right edges of an icon button to give the button height (see Figure 3). When the user clicks the button, the entire button inverts, maintaining a subtle 3-D look.

To achieve a consistent 3-D effect, our icon buttons follow certain guidelines. We always use a 2- to 3-pixel margin between the icon boundaries and the edge of the surface of a button. The frame and shading that give the icon button its 3-D appearance extend four pixels in each direction. Figure 4 shows a closeup of an icon button's shading.

When you design your icon buttons, don't forget the additional pixels that you'll need for the shading. If you want to use a 32 x 32 icon in a button, the minimum button size is 40 x 40. If you want the entire button, including its shading, to fit in a 32 x 32 space, the icon should fit in roughly a 22 x 22 area so that it sits comfortably within the button.

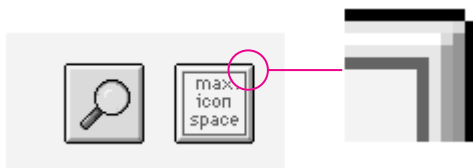


Figure 4
Icon Button Detail

As you design your icons, try to keep the image as free form as possible. In general, the image shouldn't follow the outline of the button. A square icon can hide the appearance of a square button and diminish the 3-D effect when the user presses the button. The more free form the image, the clearer it is within the button. The icon buttons in Figure 3 are good examples of free-form images.

Our icon buttons can appear in one of three states: available, selected, and unavailable. Figure 5 shows what these states look like.

- In the available state, the button shows the full 3-D shading, and its icon is displayed in the normal way. When the user clicks the button, it goes into the selected state.
- In the selected state, the button and its icon darken, similar to a selected icon in the Finder. Since it's a 3-D image, the shading changes to make the icon recede into the surface, giving the impression that it's pushed in. If the button behaves like a radio button, it stays in this state until the user selects a different button. If the button behaves like a push button, it pops back out when the user releases the mouse button.
- In the unavailable state, meaning the operation that the button represents is not available, the entire button is dimmed to make it appear flat against the surface. When the user takes some action that makes the button's operation available, it changes to the available state.



Figure 5
Icon Button States

You can change the icon in the button depending on context. Dynamic buttons give the user visual information about what effect the operation will have. For instance, suppose you have a list that can show user names, folders, and documents at the same time. When the user selects a folder or a document from the list, the icon button for the Open operation displays a folder icon or a document icon; the folder case is shown in the window on the left in Figure 6. When the user selects a user name from the list, the button changes to display the standard user face icon, as shown on the right in Figure 6.

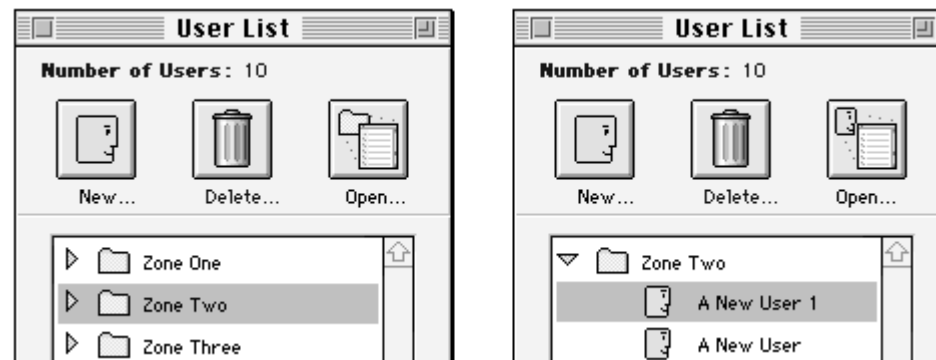


Figure 6
A Window Containing a Dynamic Button

You need to be especially careful when designing dynamic icon buttons. While the image in the icon may change to specialize the button, you don't want to change the meaning of the button. Changing the icon from open folder to open document is useful, but changing the meaning of the button from "open item" to "quit application" would confuse the user.

THE NAUGHTY BITS

This issue's CD includes code that makes it easy for you to add these 3-D effects to your MacApp application, and includes a sample application, with source code, that shows you how to use the code. There are some adorners for drawing gray backgrounds and chisels in windows and around text and list boxes, 3-D versions of TControls such as T3DCheckBox and T3DRadio, and two C++ stack-based objects to help with drawing. The code requires MacApp 3.0.x and the resulting application requires System 7.

The 3-D adorning classes on the CD are TGrayBackgroundAdorner, T3DGrayBackgroundAdorner, TWhiteBackgroundAdorner, T3DFrameAdorner, T3DLineTopAdorner, T3DLineBottomAdorner, T3DLineLeftAdorner, and

T3DLineRightAdorner. The 3-D TControl classes are T3DButton, T3DCheckBox, T3DRadio, and T3DIconButton.

The easiest way to incorporate the 3-D adorners and classes into your application is to use a view editor, such as AdLib or ViewEdit. Attach the adorners to your views the same way you would attach any other adorer. For best performance, insert the line adorners after the Draw adorer in the view's adorer list. One of the background adorners should replace the Erase adorer (before the Draw adorer) in your window's adorer list.

For the classes, create a control 'View' resource of the type you want, and insert "3D" into the name. For instance, TCheckBox becomes T3DCheckBox, TRadio becomes T3DRadio, and so on. T3DIconButton is an exception because there is no TIconButton class. To create a T3DIconButton, you'll need an icon suite ('ICN#' resource). Next, create a TControl and change the class name to T3DIconButton. Then, set the user value of the T3DIconButton to the ID of the icon suite. If you install the adorners or classes this way, be sure to call InitU3DDrawing in your initialization code so that the linker doesn't dead-strip any adorners that you don't explicitly reference in your code.

Another way to use the adorners is to create them procedurally and use the AddAdorner method to add them to a view. If you do it this way, you can create a single global instance of each adorer you intend to use and pass a pointer to the global adorer when you call AddAdorner. MacApp uses this technique for the common adorners such as TDrawAdorner and TEraseAdorner. You may still want to call InitU3DDrawing in your initialization function in case you decide to use the first method for some of your views.

You can create the 3-D classes (such as T3DCheckBox and T3DRadio) procedurally as you would any other MacApp view.

STACKING THE DECK IN YOUR FAVOR

The MacApp classes on the CD use two C++ stack-based objects to help you draw the 3-D effects. CGraphicsState objects save and restore the drawing state, and CDrawPerDevice objects let you customize your drawing routines for different pixel depths. For more information on stack-based objects, see "Stack-Based Objects."

When a drawing routine changes the graphics state, as most of the 3-D adorners and classes do, it's a common courtesy to restore the state to the way you found it when your routine is done. CGraphicsState saves important characteristics of the graphics state, such as the foreground color, the background color, the pen state, and the text style. All you have to do is declare an instance of a CGraphicsState in your function's local variable list, and forget it. C++ takes care of everything else, as in the following example.

AdLib is a view editor sold by MADA. You may purchase a copy by calling MADA at (408)253-2765 or by sending an AppleLink to MADA. •

STACK-BASED OBJECTS

Stack-based objects are a powerful feature of C++ that lets you use function scope to perform routine actions in your code. They work because of constructors and destructors.

To create a stack-based object, create a C++ class with a constructor and a destructor. In the constructor, add the code that you want to execute when an instance of the object is created. For example, the constructor can save the graphics state, set the cursor to a watch, or lock down handles. In the destructor, reverse the process: restore the state, reset the cursor, or unlock the handles.

Since the object is created on the stack as a local variable, C++ frees the memory it occupies when the function ends. But before the object is freed, its destructor executes.

The CGraphicsState class is a simple illustration of this technique. When you declare an object of type CGraphicsState, the constructor saves the graphics state in the object's data members. When the function ends, and the destructor is called, it restores the graphics state from its saved data.

In theory, you can use the constructors and destructors of stack-based objects to do anything you want; in practice, you have to be careful. Obviously, your destructor should reset any changes that your constructor made, so it's important that they're in sync. If your constructor allocates memory, you should be prepared to handle a failure. Since constructors can't return a value, you'll need to provide some way to find out whether the operation was successful.

```
MyClass::DrawIt() {
    short          aLocal;
    CGraphicsState theGState;
    . . .
    // Do some drawing.
    . . .
}
```

When the function ends, CGraphicsState's destructor restores the graphics state.

Sometimes you want your drawing routine to behave differently depending on the pixel depth of a monitor. Our stack-based object, CDrawPerDevice, lets you specify how your drawing routine should react to monitors with different pixel depths. CDrawPerDevice performs the same function as the DeviceLoop routine (see the article "DeviceLoop Meets the Interface Designer" in *develop* Issue 13), but CDrawPerDevice is easier to use with MacApp. To use the DeviceLoop routine with C++ or with Object Pascal, you need to create a static drawing routine that you can pass to DeviceLoop. Since you can't make the Draw method of a TView subclass static, you need to create a second drawing function. If you had to do this for all your drawing classes, it would get messy and wasteful. CDrawPerDevice doesn't require static draw routines.

To use CDrawPerDevice, declare an instance of it and initialize it with a CRect that specifies the QuickDraw drawing area. Then do your drawing in a while loop that calls CDrawPerDevice::NextDevice, like this:

```
MyClass::DrawIt(VRect area) {
    CGraphicsState    theGState;
    CRect              aQDArea;
    short              pixelSize;

    this->ViewToQDRect(area, aQDArea);
    CDrawPerDevice device(aQDArea);
    while (device.NextDevice(pixelSize)) {
        if (pixelSize > 2) {
            // Do some color drawing.
            this->DrawColor();
        }
        else {
            // Do some black-and-white drawing.
            this->DrawBW();
        }
    }
}
```

Note that you create actual instances of CGraphicsState and CDrawPerDevice on the stack (that is, as local variables in the function). Don't instantiate pointers to these objects.

Correct		Incorrect	
CGraphicsState	myState;	CGraphicsState	*myState;
CDrawPerDevice	drawing(aQDArea);	CDrawPerDevice	*drawing;
			new drawing(aQDArea);

DECORATING YOUR WINDOWS WITH ADORNERS

An adorning is a handy class that MacApp uses to draw views. Instead of putting special drawing code in the Draw method of a class, you can use an adorning to do the drawing for you. For example, a line adorning might draw a single black line along the top edge of a view. You can create an adorning to draw something, like a line, and then add it to any view you want.

The basic 3-D adorning is TGrayBackgroundAdorning. As you might have guessed, it colors the background of its view gray. This adorning calls EraseRect on the viewRect

with `kLightGray`. Using `EraseRect` instead of `PaintRect` in gray has a useful side effect: because any view without an explicit `TDrawingEnvironment` inherits the foreground color and the background color from its superview, all subviews of a view with a `TGrayBackgroundAdorner` will erase with gray by default. This is especially important when you have `TStaticText` items. Without the gray `TDrawingEnvironment` they inherit, they would draw ugly white boxes around the text.

`T3DGrayBackgroundAdorner` is a subclass of `TGrayBackgroundAdorner` that adds the chiseled effect described earlier in this article. Its `Draw` routine first calls the inherited method, which fills the `viewRect` with `kLightGray`. Then it draws the gray and white lines around the edges. These lines are not drawn if the monitor is in black-and-white mode. To draw a chiseled gray line that appears as a black line in black-and-white mode, use `T3DLineTopAdorner`, `T3DLineLeftAdorner`, `T3DLineBottomAdorner`, or `T3DLineRightAdorner`.

The `T3DFrameAdorner`, used for `TEditText` and list views, is an interesting case. Unlike the other frame adorners, which tend to draw a one-pixel-wide line around your view or portions of your view, `T3DFrameAdorner` draws two-pixel-wide lines to achieve its 3-D effect. This isn't a problem for `TEditText`, but it means that you need to inset your `TListView` two pixels from the `TView` or `TControl` that encloses the `TListView`. Then add the `T3DListFrameAdorner` to the *enclosing* `TView` instead of the `TListView`. You should also add a `TWhiteBackgroundAdorner` to your list view or edit text (before the `Draw` adorner) so that the text appears on a white background. Another way to get the same effect is to set the drawing environment background for the list view or edit text to white.

Note that the 3-D adorners draw the chiseled effect only on devices whose pixel depth is 2 or more. None of the 3-D adorners described here, except the line adorners described above, show up in black and white. This decision to hide 3-D effects in black and white follows the Macintosh Human Interface Guidelines recommendation that applications not try to simulate 3-D effects in black and white.

TO STAY IN CONTROL YOU HAVE TO HAVE CLASS

It would be nice if you could give 3-D effects to controls simply by adding adorners. To do it that way, you would need to draw over the CDEF. This is not a recommended programming practice, since CDEFs may change, or a user might have installed a special CDEF such as Greg's Buttons. Any assumptions your adorner makes about the way a control looks may turn out to be false. That's why we use MacApp classes instead of adorners to handle the drawing of 3-D controls.

The 3-D control classes are still easy to use because they inherit from their 2-D ancestors. On 1-bit devices, the 3-D classes simulate the standard two-dimensional CDEF drawing. Since icon buttons need to maintain a distinct button appearance

even in black and white, T3DIconButton does simulate a 3-D effect even on 1-bit devices.

The text for checkboxes, radio buttons, and regular push buttons is normally drawn on a white background because that's the way the System CDEF works. The TControl subclasses change the effect to draw on a gray background and, where appropriate, to add a 3-D effect. T3DCheckBox, for example, draws black text on a gray background and adds the chiseled effect to the checkbox itself. You can customize the drawing colors by setting the fForegroundColor and fBackgroundColor fields.

You can use two different kinds of 3-D buttons: T3DButton, descended from MacApp's TButton, and T3DIconButton from our code. Class T3DButton just inherits from TButton to draw in 3-D.

Class T3DIconButton uses the TIconSuite class and icon suites ('ICN#' and 'ics#' resources) to draw icon buttons. T3DIconButton uses CDrawPerDevice to draw buttons differently on monitors set to different pixel depths. For example, if you're drawing on a 4-bit monitor, T3DIconButton uses the 'icl4' or 'ics4' resource for your icon, but if you're drawing on an 8-bit monitor, it uses 'icl8' or 'ics8'. Also, instead of inverting the icon when the button is hit, T3DIconButton masks the icon with the kSelected mask just as the Finder does.

ADDING THE 3-D CLASSES TO YOUR MACAPP APPLICATION

Here's how you add the 3-D adorners and 3-D control classes to your MacApp application:

1. Drag the files in the 3D Drawing folder on the this issue's CD to your project folder.
2. Edit your *MYourApp.cp* file as follows: In main, add "extern InitU3DDrawing();" after your includes, and add "InitU3DDrawing();" where the other initialization routines are.
3. Edit your *YourApp.MAMake* file to include the files you added from the CD. The *Demo3D.MAMake* file is a good example of how to do this.

WHY BOTHER WITH A THIRD DIMENSION WHEN WE HAVE TWO PERFECTLY GOOD ONES?

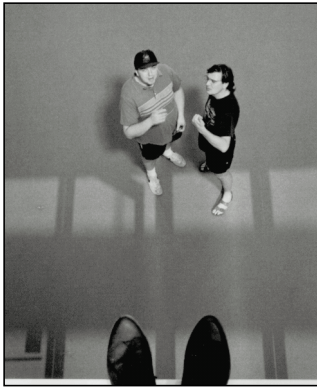
"So," you ask, "why should I bother with all this stuff? Does it really make my application any easier to use?" The answer is: probably. There have been no studies that definitively prove a 3-D interface is better than a 2-D interface. What we hear is

a cry for more 3-D elements in the Macintosh interface. The subtle effects described in this article are enough to add depth and tactility to the workspace. This results in ease of use and clarity of view for the user. It's also a lot more fun without being distracting.

“Why not just use a custom CDEF like Greg’s Buttons?” The 3-D effects from this article weren’t created arbitrarily. They reflect a lot of work done to date by some of Apple’s visual designers. CDEFs are also more difficult to write than adorners and TControl subclasses. Furthermore, CDEFs define only the way controls are drawn. You can’t use a CDEF to create a chiseled effect and a gray background for your windows, for example.

“This all looks great with MacApp, but what if I’m using something else?” There’s nothing intrinsic about the 3-D effects that makes them difficult to implement in any application framework. We chose to use MacApp because it made sense for our work. You could easily adapt these techniques to fit the framework you’re using.

We hope that someday 3-D will become the rule, and that it will be just as easy to make a window three dimensional as it is now to make it colorized. Until then, the methods we’ve described in this article should save you time and frustration when making your applications 3-D. And we’re confident that those who use your applications will appreciate and benefit from your effort.



VIEW FROM THE LEDGE

TAO JONES

The brain trust that makes up the heart, soul, and spleen of *develop* recently got together to have a few beers, pat each other on the back in a team-building-while-not-being-too-aggressive way, and just generally — as Albert Einstein once said — “figure it all out.”

Two interesting things came out of the meeting. First, it's very difficult to pat each other on the back in a team-building-while-not-being-too-aggressive way after you've had a couple of beers; and second, *develop* has a serious flaw. A flaw as wide, as gaping, and as socially repugnant as the space between Alfred E. Neuman's teeth.

As by now I'm sure you've discovered, *develop* is the *best* source for “how to” articles on Apple technology. It's the literary equivalent of an overprotective older brother in a strange neighborhood. Yet all those back issues of *develop*, combined with that big stack of CDs you save but never use, are doing *nothing* to help you survive in the day-to-day workplace. You may be able to write C++ in your sleep, but it's not going to do you any good if you don't know socially important things like who your company's CEO is or the finer points of water cooler etiquette. The raw philanthropic nature of this journal, combined with the necessity to fill a couple of extra pages, compels us to offer some useful office survival tips along with the gripping technical articles *develop* has always delivered.

From this point forward for eternity (or until I'm fired, or until the hate mail reaches unbearable levels), I'll be your tour guide through the political jungle of the modern office. Like any good companion, I'll be pointing out the nasty baboons from the safety of our digital van, helping you step around the virtual guano while we roam the electronic terra firma, bellowing at you when you try to feed your little sister to the lions, and pretending to know what I'm talking about when I don't have a clue.

To offer the ultimate in customer service, and to provide snappy repartee, I'll be answering questions sent in by inquisitive readers and hand-picked by our crack staff. Rest assured your questions will be held in the utmost confidence, unless they have good blackmail potential. Everyone “fortunate” enough to have their questions printed will receive an incredibly cheap, yet heartwarming collectible, gift. Interestingly enough, questions arrived even before this column was announced. Leaks are everywhere, I guess.

Dear Tao,

As a product manager at a major U.S. software company, I often find myself losing in negotiations with my colleagues for our company's internal resources. I always go into meetings prepared and well rehearsed, and I always come out with less than I'd hoped for.

The situation is growing desperate. I have yet to be promoted, and I'm afraid I'll need to call in the loans I've made to my children so I can pay off my BMW.

Please help,

Distressed in Denver

Dear Distressed,

You say you're a product manager yet you got your point across in fewer than ten pages. My guess is you're being too direct with people in the workplace. A rule of

TAO JONES is the pen name of an Apple employee so afraid of speaking directly that he can't even get up the nerve to introduce himself here. •

Tao Index: In the computing industry, people who refer to themselves as “guru” or “wizard” usually aren't. •

thumb: When dealing with people in the workplace, never be direct.

There are several disadvantages to saying what's on your mind. One is that people will know what you're thinking. If you're dealing with someone who isn't direct, they'll automatically have an edge on you; they'll know what you know, but not vice versa. If you're dealing with someone who is truly malevolent — the kind of person who bites the heads off marshmallow bunnies before eating the rest, or worse, someone who wears suspenders — your directness can be very detrimental to what you're trying to accomplish.

As you've already learned, being direct can be especially harmful when dealing with management. Let's say you have some idea that you present honestly and forthrightly to your boss. When your boss responds with, "It basically seems like a good idea; let's see if we can work something out," what this really means is you're heading straight into a nightmare commonly referred to as "negotiation." This is bad news in a big way, because Boss School has taught your manager to (1) verbally whittle your idea into a headless monstrosity doomed for failure, and then (2) either take credit for your superhuman struggles in making the plan succeed or, more likely, crucify you mercilessly for failure.

Just like everything else in the '90s, the answer to your problem lies in a combination of inner awareness, macrobiotic diets, shaky mutual funds, and self-help. Not being direct will seem unnatural at first, so I give you this as your homework: Put a marble on a table. Study it closely until you really begin to understand what that marble is all about. Then, for the next half hour, describe the marble without using the words "round" or "sphere." Once you think you've got it, ask some friends in and describe the marble to them. If they have no idea what you're talking about, you'll know that you're well on your way to honing one of the most important of your office survival skills.

As for calling in those loans from your children: Mom, I *told* you that I'd pay you back when I got a chance.

Dear Tao,

I have an employee who is perpetually late and always uses the excuse "my dog got loose." I've always been a bit suspicious, and upon checking I discovered that he doesn't even own a dog! What should I do?

Sign me,

Flea-Bitten

Dear Flea,

As a manager you're probably already aware that chronic lateness is usually only a symptom of some bigger underlying problem. Situations like this are sensitive, extremely volatile, and prone to disaster if not handled by properly trained professionals.

First, you should make a detailed account *in writing* of all the incidents as they happen. Then call both the local animal shelter and the police department, stating that you believe one of your employees is running an illegal dog-racing operation. Don't forget to mention that you think it's likely your underling will "do something" (make sure to use that exact phrase) with the dogs if he finds out that the authorities are after him. As you watch events unfold, it may seem like a painful and arduous process. But believe me, once it's all over, your employee will be so choked with emotion that he won't be able to find the words to thank you.

RECOMMENDED READING AND LISTENING

- *Gojira* by Mark Jacobson. The *autobiography* of everyone's favorite Japanese monster.
- *Cool Tricks* by John Javna. It's high time you learned how to tie balloon animals, isn't it?
- *Dead Man's Party* by Oingo Boingo. Play it loud, and when your boss comes in, say "It's symbolic."

116

Tao would like to thank Sarcasmo for rewriting most of this column without trying to claim authorship, Cindy Jasper and Caroline Rose for their unnerving faith, and Mr. and Mrs. Jones for his existence. •

Tao needs questions to keep from dropping into a nasty blue funk. Please send them to AppleLink DEVELOP. •

Q *Will QuickDraw and QuickDraw GX coexist, or will QuickDraw GX replace QuickDraw?*

A QuickDraw isn't leaving with the introduction of QuickDraw GX. It's here to stay. Among other reasons, QuickDraw functions are used extensively by the Macintosh Toolbox and applications. For more information, see the article "Getting Started With QuickDraw GX" in this issue of *develop*.

MACINTOSH

Q & A

Q *Will display cards that offer QuickDraw acceleration be affected by QuickDraw GX?*

A Current QuickDraw acceleration cards aren't affected by QuickDraw GX. QuickDraw GX doesn't use any of the current QuickDraw calls and its presence won't affect applications that use only QuickDraw. QuickDraw acceleration cards also won't accelerate QuickDraw GX.

Q *The LaserWriter driver before the QuickDraw GX version has an option to print in Color/Grayscale or Black & White. Why isn't this option in the QuickDraw GX LaserWriter driver?*

A The Color/Grayscale option was added to the LaserWriter driver only for compatibility reasons. At the time, some applications couldn't deal with the color option (specifically with cGrafPorts), so a Black & White mode was also provided. The Black & White mode exhibited the same functionality as the earlier LaserWriter driver 5.2.

Because most applications are color compatible now, the option was removed from the QuickDraw GX printer drivers. Some people reported that the option let them print faster when they chose Black & White. This was true because of quirks in the earlier LaserWriter driver version. Under QuickDraw GX, this shouldn't be a problem. If it turns out to be a problem for your driver, you could incorporate black-and-white threshold printing into your "rough draft mode" code. The QuickDraw GX LaserWriter driver always prints in color.

Q *When a QuickDraw GX PostScript printer driver generates a file, will it work for Level 1 and Level 2 printers?*

A The LaserWriter driver bundled with QuickDraw GX produces a flavor of PostScript that we call "portable." This flavor is meant to work on the widest range of printing devices, be they Level 1 or 2, color or black and white.

Q *What PostScript Level 2 features does the QuickDraw GX printing mechanism take advantage of?*

Kudos to our readers who care enough to ask us terrific and well thought-out questions. The answers are supplied by our teams of technical gurus in Apple's Developer Support Center; our thanks to all. Special thanks to Pete ("Luke") Alexander, Mark Baumwell, Joel Cannon, Matt Deatherage, Tim Dierks, Nitin Ganatra, Bill Guschwan, Mark Harlan, C. K. Haun, Dave Hersey, Rich Kubota, Jim Luther, Joseph Maurer,

Kevin Mellander, Don Moccia, Ed Navarrete, Guillermo Ortiz, Faith Pai, Dave Radcliffe, Brigham Stevens, Steve Strassmann, Dan Strnad, John Wang, and Dave Wells for the material in this Q & A column. •

A The Level 2 features used in QuickDraw GX mostly have to do with patterns, text, and bitmaps:

- QuickDraw GX patterns are converted into Level 2 pattern dictionaries when going to a Level 2 printer. The actual PostScript code emitted by the driver differs little with respect to patterns when going to Level 1 or Level 2. However, the procedures defined in the header do something entirely different on a Level 2 printer.
- Line layout in QuickDraw GX takes advantage of the **xshow**, **yshow**, and **xyshow** operators when it makes sense to do so.
- The Level 2 rectangle operators are used in Level 2, though this happens in the procedures defined in the header rather than in PostScript code from the driver.
- On Level 2 printers, the indexed color spaces are used for printing bitmaps up to eight bits deep.
- The plan is to use Level 2 device-independent color when possible.

Q *Why can't I get NewMessageGlobals to work in my gxInitialize message for my QuickDraw GX printing extension? The global data I try to initialize isn't being accessed correctly.*

A You shouldn't call NewMessageGlobals from any routine in which you access your global data. Otherwise, because of optimization that your compiler may perform, the data references can be invalid. Instead, use an approach like this:

```
extern long A5Size(void);
extern void A5Init(void *);
typedef struct GlobalType {
    StringHandle aString;
} GlobalType;
GlobalType myGlobals;

/* This routine sets the initial values of our global data. */
OSErr InitGlobalData()
{
    OSErr err;
    // Initialize our globals.
    myGlobals.aString = GetString(r_myStringID);
    err = ResError();
    if (!err) DetachResource(myGlobals.aString);
    return err;
}
```

```

/* Our override for the gxInitialize message. */
OSErr MyInitialize()
{
    OSErr err;
    // Create an A5 world, then go initialize our global data.
    err = NewMessageGlobals(A5Size(), A5Init());
    if (!err) err = InitGlobalData();
    return err;
}

```

A detailed explanation of the problem accompanies the Kabooms printing extension sample on this issue's CD.

Q *A QuickDraw GX printing extension I've already written works fine, but when I install my latest creation, it doesn't show up in the print dialog — only my old printing extension does. Both extensions are the same except for a few lines of code in their gxDespoolPage message overrides. What's going on?*

A Your printing extensions shouldn't have the same creator type. QuickDraw GX requires unique creator types for drivers and printing extensions, just as the Finder does for applications. A creator type must be unique because QuickDraw GX uses it to build its chain of message handlers. If two printing extensions have the same creator, there's no way to determine which is which in the chain. You can register creator types for your printer drivers and printing extensions with the Developer Support Center (AppleLink DEVSUPPORT).

Q *Will the SndPlayFromDisk routine in the new Sound Manager version 3.0 work on Macintosh models without the Apple Sound Chip? For the current Sound Manager, is there a good way to tell whether a machine supports SndPlayFromDisk?*

A Sound Manager 3.0 supports SndPlayFromDisk on all models. Playing from disk is based on the performance of the machine's SCSI device more than the Sound Manager's performance. In earlier versions of the Sound Manager, SndPlayFromDisk works only on Apple Sound Chip machines.

The System 7.1 interface has a play-from-disk Gestalt flag in the sound attributes selector. If this flag bit is set, play from disk is supported. If you're not running System 7.1, you'll have to guess based on whether the version of the system being used has an Apple Sound Chip. So first check for the new Gestalt flag and the presence of Sound Manager version 3.0 or later, which supports this flag. Older versions of the Sound Manager don't report whether play from disk is supported, so you'll have to check for the Apple Sound Chip.

Q *Is the Component Manager bundled with QuickTime? In other words, will the users of the application I'm designing have to buy QuickTime in order to get the Component Manager?*

A The Component Manager was first introduced as part of the QuickTime 1.0 system extension. This makes the presence of QuickTime a requirement for system software versions earlier than System 7.1. The Component Manager is now part of System 7.1, giving the extended functionality to any Macintosh application running on System 7.1 regardless of the presence of QuickTime.

Q *How does the Component Manager handle multiple segments?*

A The Component Manager doesn't automatically support multiple code segments. In fact, it assumes that all the code is in one segment. The 'thng' resource allows you to specify one segment. Therefore, if you load other code segments from your main segment, they should be loaded and locked in the system heap. There are several ways you can do this; the register routine probably is the best place to do it. Since the register routine is called only once at registration time, this allows your component to completely load the remaining code segments into the system heap. Components should ensure that A5 (or A4) is set appropriately before any intersegment calls. For more information, see "Managing Component Registration" in this issue of *develop*.

Q *How can I get answers to my Macintosh Common Lisp questions?*

A Macintosh Common Lisp technical support is available on the Internet at the following addresses (add "@internet#" for the corresponding AppleLink addresses):

- bug-mcl@cambridge.apple.com, which reaches the MCL team
- info-mcl@cambridge.apple.com, which reaches MCL users at large

You can join the info-mcl list in one of the following ways:

- Contact info-mcl-request@cambridge.apple.com on the Internet to add your address to the list.
- Join INFO.MCL\$ on AppleLink by contacting ST.CLAIR.
- Check comp.lang.lisp.mcl on Internet netnews.

MCL utilities and sample source code are available in these places:

- on the MCL 2.01 CD-ROM (available through APDA)

- on the *Developer CD Series* disc (Tool Chest edition) and the *develop Bookmark* CD
- on the Internet, by anonymous ftp from the location `cambridge.apple.com:/pub/MCL2/contrib`

Q *Can I assume that the value of the ColorSync CWorld parameter returned by the CWNewColorWorld routine isn't null if the routine was successful? I'd like to determine whether a color world exists by checking the variable for null.*

A You can assume that if no error is returned by CWNewColorWorld, the CWorld parameter will be a valid handle. In other words, it won't be null if no error is returned.

Q *The icons that appear in our application's menu lists are very, very small. It looks like they're 8 x 8 (scaled) instead of the standard 16 x 16. Can you tell from our test code why this is happening?*

A This was a wild one! What's causing the weird behavior is that you have a 'cicn' resource with ID = 256 that's smaller than 32 x 32. When the MDEF finds such an icon, it uses it to size the area to use for the menu icons. So, for your application the solution is either to change the 'cicn' to be larger (32 x 32) or to give it a different ID.

Q *I've installed a resource error procedure (ResErrProc) but it gets called with an error -192 (resNotFound) every time I release a resource. Why is this happening and what can I do to fix it?*

A If you install a ResErrProc, the Process Manager switches it out when switching to another application. The problem you're having is that the Process Manager doesn't switch it out while in Process Manager code itself. If you break into the debugger in your ResErrProc and examine register A5, you'll find it's not your A5; it is in fact the Process Manager's.

What has happened is that the Process Manager has patched ReleaseResource for its own mysterious purposes. An error occurs, which the Process Manager believes it can ignore, but because your ResErrProc is still installed, it gets called and reports the error. To avoid this, when your ResErrProc gets called, check to see whether register A5 is equal to the low-memory global CurrentA5. If the values aren't equal, you can assume your application wasn't responsible for the error, so you can ignore it.

The code might look like this:

```

void MyResErrorProc (void)
{
    long  A5;
    A5 = SetCurrentA5 ();
    if (A5 == *(long *)CurrentA5)    /* If they're equal, we're in
                                     current app code */
        Debugger();                /* ...or whatever you want to do here */
    else                             /* Not in current app code */
        SetA5 (A5);                /* Restore old A5 */
    return;
}

```

Similar problems can occur if you make calls that make other Resource Manager calls. The secondary calls may produce errors that are inconsequential to the primary call, but your ResErrProc gets called anyway. This makes ResErrProcs of limited use, so use any information reported by a ResErrProc carefully.

Q *We can't apply the rotation and skew effects to a QuickTime 1.5 movie. We've created an identity matrix, applied RotateMatrix to the matrix, set the matrix to the movie using SetMovieMatrix, and played the movie. The movie didn't rotate but the movieRect rotated and the movie scaled to the movieRect. Is there anything wrong with what we're doing?*

A Rotation and skew will give you correct results for matrix operations but they haven't been implemented into QuickTime movie playback yet. Scaling and offset transformations now work with movies and images; rotation and skew are important future directions. Meanwhile, you can accomplish rotation and skewing by playing a movie to an off-screen GWorld and then using QuickDraw GX to display the rotated or skewed off-screen GWorld.

Q *The demo programs on the final QuickTime for Windows CD won't run without locking up, because the QTInitialize function doesn't return. Any ideas?*

A The final QuickTime for Windows exhibits the behavior you describe when the display board is configured in a way that QuickTime for Windows doesn't recognize. XGA and SuperVGA are modes that could cause this problem. To check whether this is the problem, edit the QTW.INI file to include the following:

```

[Video]
Optimize = Driver

```

The default is Hardware. This could be causing the problem you describe.

Q *While updating our application to run correctly under the Kanji version of System 7.1, we've noticed that the space bar sends two key-down events (values \$81 and \$40) as opposed to one (\$20). To determine whether the space bar was hit, should we use the key code from the event record instead of the character code?*

A This is new with Kanji 7.1 and is still a controversial issue. Codes \$81 and \$40 correspond to the two-byte space character, which is different from the standard ASCII space character (different width).

You can't use the key code from the event record, as it has been lost during KanjiTalk's event processing. Moreover, even if it were there (or if you got it through a GetKeys call) you shouldn't use it; the space bar has different key codes on different keyboards!

So, the only reasonable workaround in your case seems to be to expand your space bar key-down check to compare with \$81 (the first byte of a two-byte character) followed by \$40, in addition to comparing with the standard ASCII \$20.

Q *What kind of resources are available to incorporate WorldScript capability to our programs?*

A The first thing to do to support WorldScript is to make sure that your product takes full advantage of the Script Manager. The most up-to-date and comprehensive reference for learning about it is *Inside Macintosh: Text*. Be sure to check the QuickDraw Text and Text Utilities chapters, because much of the old Script Manager functionality has been moved into those areas. Other references are *Guide to Macintosh Software Localization* and *Localization for Japan*. These are all available through APDA. (If compatibility with system software versions before System 7.1 is important to your application, see the earlier documentation: the Script Manager and Worldwide Software Overview chapters of *Inside Macintosh Volumes V and VI*, and *Macintosh Worldwide Development: Guide to System Software*.) See also the Text Services Manager chapter of *Inside Macintosh: Text*.

In general, all the rules for being Script Manager-compatible apply, but with a few new twists. WorldScript is still a very young technology and there isn't a lot of detailed information yet. Some human interface issues have not matured to the point of being ready to standardize. This means that you might find yourself breaking new ground as you add WorldScript support to your application.

The Japanese Language Kit is the first new product to take advantage of WorldScript. All the non-Roman system software uses WorldScript but doesn't really take full advantage of the possibilities.

The main advantage of WorldScript is that it supports multiple scripts in a system. The old model was that there could be a Roman script plus one other optional script, and Roman was always the secondary script, unless it was the only script. Now, any script can be the primary script, and there can be multiple secondary scripts. (Roman is still required to be available.) This makes it possible to have Japanese (and in the future, other scripts) installed as secondary scripts in any system.

Because many of the Macintosh Toolbox and operating system routines don't pass script or font information with text, it's difficult to display multiscrypt text properly. Menus, dialogs, window titles, and filenames are problem areas. A short-term solution is the Language Kit Extension, which was introduced in the Japanese Language Kit. It dynamically changes the system fonts according to the region code in each application's 'vers' resource. This allows localized applications to display menus, dialogs, window titles, and filenames correctly. Multiple applications running concurrently can have independent system fonts.

The Language Kit Extension also extends the Views control panel to control the current script and font for the entire file system. This allows the user to display, create, and edit filenames for any script installed. Note that this solution supports only one script at a time, and fundamental changes will have to be made to the Toolbox to support more than one script concurrently.

A technical overview of potential internationalization problem areas is given in the "Internationalization Checklist" on this issue's CD. Internationalization tips are also provided in "Writing Localizable Applications" in Issue 14 of *develop*. Here are some areas requiring special attention:

- Font names should be displayed in menus in their appropriate script and font.
- The system (primary) script may not be the same as the application script.
- Script system resources (date/time/currency) should be referenced explicitly.
- Text should be tagged with script and font information so that it can be displayed in its appropriate script and font.

Styled text and QuickDraw GX are multiscrypt-capable and should be used as much as possible. Unstyled text documents limit text to one script (character set), which is undesirable.

Ideally, the localized version of a product should be independent of the scripts and languages it supports. With Macintosh multiscrypt support and today's cultural diversity within nations, users want "foreign language-capable" applications that let them mix writing systems and use familiar regional formatting conventions.

Here are a few tests to see how well you're doing:

- Does the base version of your application work on all localized versions of system software (such as KanjiTalk, Arabic, German)?
- Install the Japanese Language Kit and see whether your application supports multiscrypt text. Try using the two-byte Japanese script with a relatively simple script such as German, and then with a bidirectional script such as Arabic.

Q *We're planning to distribute a custom font with our application, by including the FOND and NFNT resources in its resource fork. Are there any problems we might be overlooking by using this technique?*

A There are things you must be aware of when embedding fonts in applications. First, no ID conflict resolution is done. Since the system caches some fonts for performance reasons, if your font has the ID of a font in the system, you can't predict whether you'll get your font or the system's font. Second, you must use NFNT instead of FONT resources for bitmap fonts, or the system will sometimes get the wrong font strike. This is described in the Macintosh Technical Note "Fond of FONDS" (Text 21). You should never try to override a font in the System file with one in your application; the name of your custom font should be different from all other font names. If you don't want your font to show up in menus, give it a name beginning with a period or "%" so that AddResMenu will ignore it.

While most printer drivers and the system take pains to find fonts wherever they may be, some software components don't, and we can't promise this will work with all printers or all software. While installing fonts in an application should work for most purposes, it's preferable all around for you (and your customers) to install your fonts in the system. Of course, you should always ask for your font by name in your code.

Q *When I try to use the MDEF mDrawMsg to draw the contents of the pop-up box for a pop-up menu, if the system justification is right to left and I send it mDrawMsg with a rectangle that's too short, the system hangs unless I move the left edge of the rectangle off to infinity and use clipping to constrain the drawing. Is there a better way to do this?*

A There may be an easier way to achieve the same result. The standard MDEF has been modified to support two new messages to do exactly what you need. In fact, this was added to support the System 7 pop-up menu CDEF. For an MDEF whose entry point is defined as

```
void pascal MyMenu (short message, MenuHandle theMenu, Rect
    *menuRect, Point *hitPt, short *whichItem);
```


the two new messages are `mCalcItemMsg` and `mDrawItemMsg`. In response to `mCalcItemMsg`, the MDEF calculates the bounding rectangle of a menu item:

- Input parameters:
 - `message = mCalcItemMsg = 5`
 - `theMenu = handle to pop-up menu`
 - `menuRect.top` and `menuRect.left` = screen position at which to draw menu
 - `hitPt` = not used
 - `whichItem` = menu item to calculate the size for
- Output:
 - `menuRect.bottom` and `menuRect.right`, calculated for correct size for `whichItem`

In response to `mDrawItemMsg`, the MDEF draws an item in a menu:

- Input parameters:
 - `message = mDrawItemMsg = 4`
 - `theMenu = handle to pop-up menu`
 - `menuRect` = rectangle to draw item in
 - `hitPt` = not used
 - `whichItem` = menu item to draw
- Output:
 - Draws item `whichItem` of menu `theMenu` in rectangle `menuRect`. `menuRect` is determined by first calling the MDEF with `mCalcItemMsg`.

These messages may be ignored by other menu definition procedures; they're optional. If you find that your rectangle wasn't resized after an `mCalcItemMsg` call, the MDEF has probably ignored your size request, and you should do it manually. The standard MDEF will continue to support these options.

Q *We create an alias for a file when File Sharing is off, so the alias doesn't contain server and zone information. Then we turn on File Sharing and resolve the alias. The file, of course, is found but the alias isn't marked for an update. Shouldn't it be updated?*

A According to *Inside Macintosh: Files*, `wasChanged` is set to `TRUE` only on aliases created by `NewAlias` (not `NewAliasMinimalFromFullPath` or

NewAliasMinimal) when it sees that key information has changed. The key information is:

- name of the target
- directory ID of the target's parent
- file ID or directory ID of the target
- name and creation date of the volume on which the target resides

Since none of that changes when you turn File Sharing on or off, the wasChanged flag isn't set to TRUE. If you're really worried about it, just call UpdateAlias every time you use the alias (unless you think this would be a major performance hit). Or maybe you should update it only if you notice that it doesn't have server information and the server is now turned on (to check for this, call PBHGetVolParms and check the bHasPersonalAccessPrivileges bit).

Q *When I try to compile some C++ code that I'm porting from another platform, I get the error message shown below. Can you shed some light on what the error means? I can avoid it by removing labels, but sometimes it's difficult to fix. What's the real cure?*

```
# 4:14:35 PM ----- Executing build commands.
      CPlus -s tops1 -model far -sym on -mf test.cp
      Set Echo 0
File "test.cp"; line 704 # sorry, not implemented: label in block
with destructors
```

A The error message is correct; labels in a scope that locally defines objects with destructors aren't supported. The most appropriate fix would be for AT&T to fix their CFront code to support this, but that's not going to happen right away.

The variety of twisted code paths that become possible when labels and gotos are used probably gave the compiler programmers conniptions. When code can contain constructors and destructors, the C++ compiler has to be extremely careful to construct each object only once and destruct each object only once. This probably was a problem for the C++ compiler if gotos were allowed, so the compiler programmers must have chickened out and didn't write the necessary code (at least, not yet). The only solution to this error is either to avoid the label and goto constructs (which you've been trying to do) or to keep any objects with destructors out of the block that contains the label.

Q *I want to use sprintf in a standalone code resource, but I'm having trouble linking my resource because sprintf apparently requires data-to-code references. I get the error shown below; what can I do to avoid this?*

```
### While reading file "HD:MPW:Libraries:Libraries:Runtime.o"
### Link: Error: Data to Code reference not supported (no Jump
Table). (Error 59)
### Link: Errors prevented normal completion.
```

A Unfortunately, a great deal of the standard C library uses globals, which necessitates your creating an A5 world as described in the Macintosh Technical Note “Stand-Alone Code, *ad nauseam*” (Platforms & Tools 35). Additionally, some of the global data used is initialized with function pointers. For example, if you had the following global variable defined

```
ProcPtr gMyProcedure = (ProcPtr)MyFunc;
```

where MyFunc is a function of yours, you can see that the global variable would have to be initialized with a pointer to the function before your code started executing. MPW’s loader supports only pointing variables such as these at A5-relative references — that is, into the jump table. It cannot initialize them to the PC-relative reference that a standalone code segment requires. This means that you can’t link any code resource that requires a data-to-code reference — thus the error.

Fortunately, there’s a workaround. Because the standard library is written very generally and uses a lot of indirection, the linker believes `sprintf` might use some of the standard output calls; however, it never does, and since it’s these calls that necessitate the use of the data-to-code references, if we can fool the linker into accepting an innocuous substitute, we won’t require the data-to-code references that are causing us such agony now. To cut to the chase, add the following lines to your code somewhere:

```
size_t fwrite (const void *, size_t, size_t, FILE *) { return 0; }
flsbuf() {}      // These calls won't actually be called by sprintf.

fcvt() {}        // These calls are used only for floating-point %f
ecvt() {}        // and %e.
```

The first two calls override `fwrite` and `flsbuf`; these calls will no longer cause the linker to believe that you require the data-to-code references; because they’re never called by `sprintf`, replacing them isn’t a difficulty. The second two calls are only conveniences; they stub out the standard library’s code for converting floating-point arguments. If you never use `%f` or `%e` in your `sprintf` calls, these will reduce the size of your compiled code.

To get these overriding functions to mask the versions in the StdCLib.o library, make sure that the object file containing these stubs appears on the Link command line before the StdCLib.o library.

Q *Why does calling `FSpCreateResFile` return an `afpAccessDenied` error (in `ResErr`) when the destination folder is an AppleShare drop folder?*

A Using any of the `CreateResFile` calls in a drop box is a useless exercise. Here's why: The access privileges within a drop box are very limited write-only access. This lets you create new files and then perform a small set of operations on the file while it's *empty* (no bytes in either the data or the resource fork of the file). So, while the file is empty, you can open it (either fork) with write-only access (`PBHOpenDeny` or `PBHOpenRFDeny`), and write the file's attributes (`PBHSetFInfo` or `PBSetCatInfo`), including Desktop Manager comments. Once either fork has a single byte of data written to it, you can no longer open or change the file's attributes. This makes it possible to copy a file into a drop folder, but not to manipulate or delete a file (containing data) that's already in the drop folder. You can also move a file or directory into a drop folder if that file or directory is already on the same server volume.

So, when you call `CreateResFile` on a drop folder, a new file is created and data is added to the resource fork of the file. Since the file now has data in one of the forks, you cannot open the file or change any of the file's attributes. `FSpCreateResFile` fails because after performing the `CreateResFile` operation that writes data into the resource fork, it attempts to set the file's attributes (creator, `fileType`, and `scriptTag`).

For a quick explanation of drop folder access privileges and rules, see the Macintosh Technical Note "Creating Files Inside an AppleShare Drop Folder" (Files 18). For a complete explanation of what AFP access privileges you need to perform specific operations on an AppleShare (AFP) server, see the section "Directory Access Control" in *Inside AppleTalk* starting on page 13-31.

Q *When I try to send an `OpenSelection` Apple event to the Finder, I get a -903 error after calling `AESend`. Could you tell me what causes this error and how I can overcome it?*

A The application is returning the -903 error because it isn't completely set up to send and receive Apple events. This is why you're getting the error only after calling `AESend`. The problem is that the `isHighLevelEventAware` bit isn't set in your `SIZE` resource, meaning that your application doesn't send or receive high-level events. For more information on setting the `isHighLevelEventAware` bit, see *Inside Macintosh: Macintosh Toolbox Essentials*, pages 2-115 through 2-119 (or *Inside Macintosh* Volume VI, pages 5-14 through 5-17).

Q *When we launch our Macintosh application, the system heap grows by huge amounts. It seems to be filling up with font-related resources. What's causing this to happen?*

A Your application is probably calling RealFont for each font installed in the system. Trying to build lists of font information at startup nearly always balloons the system heap to enormous sizes if you have lots of fonts.

When you call RealFont, the system must load the FOND resource and walk through the association table to see if there's a bitmap for it. If there's a TrueType font, the Font Manager loads the 'sfnt' resource and examines it. A value inside TrueType fonts says "I don't render smaller than this resolution"; if the requested resolution is smaller than that value, RealFont returns FALSE even on a TrueType font. For most of Apple's TrueType fonts the value is about 6 points — calling RealFont(2, Helvetica) will return FALSE.

So the system is loading every FOND and 'sfnt' looking to see whether the fonts are real. The main solution is not to call RealFont until you're sure you need the information. For example, don't outline Size menus until just before they're accessed or until a new font is chosen, to prevent this kind of problem.

Q *I just wrote an installer script for the Apple Installer version 3.4. After the installation is complete and I click the Quit button, the Finder insists on mounting both of the disks that were used in the installation. It gives me the annoying dialog "Please insert the disk Install 1" and "Please insert the disk Program 1." Before I did the installation, only the Install 1 disk was mounted by the Finder. Why is this happening?*

A This situation typically occurs when the installer disk sets were created under System 7 and the installation is happening under system software version 6.0.x. Its cause is that the system is trying to update the second disk's desktop. The solution in this case is to insert the disk into a system running version 6.0.x before performing the installation.

Another possible cause for this problem is that the second disk opens to display a window by default. When the Installer quits, there's still "residue" Finder information to be written to the second disk, which has since been ejected. The solution in this case is to insert the second disk with the write tab lock disabled, and close any open window associated with that disk.

Q *I received an error code of -151 from NewGWorld when creating a very large off-screen bitmap. Does this mean not enough memory? If so, can I count on it not to change in future versions of the system? It's not listed as one of the possible errors in Inside Macintosh Volume VI.*

A The error -151 is cTempMemErr, "failed to allocate memory for temporary structures," or in other words, there wasn't enough temporary memory available for NewGWorld. NewGWorld returns this error after it receives a memFullErr

from TempNewHandle. (See *Inside Macintosh: Memory* or the Memory Management chapter of *Inside Macintosh* Volume VI for more information about temporary memory.) This was inadvertently left out of *Inside Macintosh* Volume VI but does appear in the MPW interface files. You can count on this error code in future versions of system software.

Q *SetStylHandle appears to dispose of only the TEstylHandle in the TextEdit record and not the handles in the TEstylRec in system software version 7.0.1. Since my application calls SetStylHandle often, I'm leaving lots of little handles around in my heap. Can I safely dispose of the handles in the current TEstylRec before calling SetStylHandle?*

A SetStylHandle does indeed orphan the related handles in your heap, a potential problem for applications using SetStylHandle. This won't be fixed in the foreseeable future. Your application will be fine disposing of the handles itself. Here's a routine you can use instead of SetStylHandle; it correctly disposes of the substructures in the TEstylRec before calling SetStylHandle:

```
void FixedSetStylHandle(TEStylHandle newstyle, TEHandle hte)
/*
    This function avoids orphaning the substructures of a
    TEstylRec by disposing of them before calling SetStylHandle.
*/
{
    register TEstylHandle oldstyle;
    /* Get the old-style handle so that we can clean it up. */
    oldstyle = GetStylHandle(hte);
    /* Dispose of the substructures first. */
    DisposeHandle((**oldstyle).styleTab);
    DisposeHandle((**oldstyle).lhTab);
    DisposeHandle((**(**oldstyle).nullStyle).nullScrap);
    DisposeHandle((**oldstyle).nullStyle);
    /* Now we can install the new style. */
    SetStylHandle(newstyle, hte);
}
```

Q *I was hiking in a remote part of the Rocky Mountains and could have sworn that I saw the familiar Macintosh beach-ball cursor. Is this possible?*

A Yes; that's the same symbol that's used to signify a ranger station. It could also be that you were hallucinating. (By the way, *develop*'s Editor-in-Cheek tells us that the beach-ball cursor can also be found on the reverse of an Eastern Caribbean \$5 bill; she took a special field trip down there just to check this out.)

Have more questions? Need more answers?
Take a look at the Macintosh Q&A Technical
Notes on this issue's CD and on the Dev Tech
Answers library on AppleLink. •

KON & BAL'S PUZZLE PAGE

I'M HERE TO SERVE

See if you can solve this programming puzzle, presented in the form of a dialog between Konstantin Othmer (KON) and Bruce Leak (BAL). The dialog gives clues to help you. Keep guessing until you're done; your score is the number to the left of the clue that gave you the correct answer. These problems are supposed to be tough. If you don't get a high score, at least you'll learn interesting Macintosh trivia.



**KONSTANTIN OTHMER
AND BRUCE LEAK**

- BAL Here's one for you, KON: I've got this server that I store builds on. It's a Macintosh ci running System 7.1 with File Sharing enabled, hooked up over Ethernet. The server runs an experimental MPW tool that spins the beach-ball cursor (don't they all?) waiting for a build to appear; then it renames the build with the next release number and makes it available to the public. Early on we had some problems with the tool and it would occasionally crash the server, but that seems to be working now.
- KON So what's the problem?
- BAL When I use the Finder to copy my latest build out there, it crashes the server. Yet when I use the duplicate command as part of the MPW build script for my project, it works.
- KON I thought you did all your builds in THINK C.
- BAL Of course, but I use MPW for file management just like the guys on the Finder team.
- KON Chalk one up for MPW.
- BAL But that's not all. When I use the Finder to copy other files out to the server it works fine. Solve me, KON. Over.
- KON I'm suspicious of that MPW shell script running on the server. It's continuously making file system calls while File Sharing is trying to copy your file over in the background. There must be some contention

KONSTANTIN OTHMER AND BRUCE LEAK

spent some time in the county slammer this summer, BAL for bungee jumping off a Cupertino freeway overpass and KON for losing the landmark "KON versus Stockton Board of Health" tenants-rights trial. They were able to get reduced sentences by working as waiters in "Pro Bono," the prison's nouvelle cuisine restaurant,

(see photo) and faithfully adhering to the *Macintosh Human Interface Guidelines*. •

caused by MPW trying to rename the file while it's still in the process of being copied.

BAL Come on, KON, what do you think we're running here? This is System 7 we're talking about. Don't you think we would get something like that right on the seventh try?

KON No comment.

100 BAL If I stop the MPW shell script running on the server or even quit MPW completely, the server still crashes when I use the Finder to copy my build to it.

KON Are the Finder copies that crash file-size dependent?

95 BAL Size doesn't seem to matter. The Finder can successfully copy files that are the same size, smaller, and bigger than mine without crashing the server.

KON So it's the server that's crashing?

BAL Yeah, the progress bar comes up on the client machine, the copy starts, and then almost immediately the server crashes and the client machine hangs in XPP's two-minute penalty box.

KON There's an updated version of the StopXPP dcmd that works with System 7 for all those MacsBugphiles. Install that baby and you're golden. When the client hangs because the server has gone away, just drop into MacsBug and type StopXPP to force the connection to time out immediately. Any other problems?

90 BAL Yeah, yeah, yeah. I've already got that dcmd. My machine isn't hung anymore, but the server is still crashed and your point total is still falling.

KON You packrat! You stole one of those rev J Apple Ethernet cards from Van Brink's garage. It's dropping bits, same as it ever did, and the way the Finder is slicing up your bit stream chokes it.

85 BAL Sorry, it's rev K of the card — the one that only drops bits when you're running in 32-bit mode — but the ci has only 8 megs of RAM and is running in 24-bit mode.

KON OK, I'll try Third-Party Cards for \$200.

80 BAL KON, I use only Apple equipment. Throw the card out altogether, use LocalTalk, and it still happens. Get a life.

KON Are there any other cards in the ci that are dropping bits? Perhaps the Apple ci cache card?

75 BAL No cards, but if it makes you feel any better, I'll have Vanna swap in a new ci, keeping only the hard disk; it still happens.

- KON I don't want a ci anymore. How 'bout we try an fx, or maybe even a Classic?
- 70 BAL Vanna's getting a little tired here. Bad news: it still happens on all of them.
- KON What if we try one of those NuTek boxes?
- 65 BAL Fine. It crashes on boot. Next.
- KON And swapping out the hard drive fixes it?
- 60 BAL Swapping out the hard drive stops the server from crashing. Any chance I can solve this problem and preserve my existing hardware investment?
- KON So there's something wrong with your system software or the File Sharing software on that drive. I'll reinstall all that stuff and watch it work beautifully. Problem solved.
- 55 BAL *Not!* After you reinstall System 7.1 and all the bogus networking disks, and reinstall MacsBug and the dcmds from the virgin copies that came with that *Debugging Macintosh Software* book, ISBN #0-201-57049-1, it still happens.
- KON I never really trusted installing over existing corrupt software. So I throw everything in the trash, empty the trash, and reinstall fresh.
- 50 BAL It still happens.
- KON Maybe the volume allocation is messed up: you've got some bad I-nodes, circular B-tree references, or some other HFS mumbo jumbo. I'll confer with Dr. Norton and see what he thinks.
- 45 BAL Doctors Norton, Feldman, and Bruffey, Disk FirstAid, and others all give it a clean bill of health, though a few file dates were wrong and they've been fixed.
- KON So maybe some part of the media went bad, producing bad sectors or ambiguous data. I'll do a media test using Silverlining.
- 40 BAL The media is fine. OK, KON. Quit screwing around. It's not a hardware problem.
- KON But when I replaced the hard disk, the problem went away. What remains after I throw all the files away? Well, the driver persists. I'll update the driver with the utility software that came with the drive.
- 35 BAL It still happens. It doesn't have anything to do with the driver.
- KON I'll try SneakerNet!
- 30 BAL Well, transferring the file by floppy disk doesn't work either, using the Finder on the server. Surprisingly, this most reliable means of

networking doesn't work even though you aren't using high-density floppies. In this case paper clips don't even help.

KON OK, I reformat the drive and reinstall the software fresh.

25 BAL Now it works. So what's the bug?

KON Well, the reformatting changed the interleave, and there was some weird timing problem hosing you.

20 BAL Enough grasping at straws. Remember it only happens with my file or copies of my file. Get those rusty old gears turning.

KON I'll chop the file into pieces and see if those still crash the server when I copy them. I'll strip out all the CODE resources with ResEdit and try that file.

15 BAL The file is pretty small now, and it still crashes.

KON I strip out all the data by setting the EOF.

10 BAL It still crashes. Look, I can build it from scratch, I can build it on other machines, I can change the code that's in it, other people can build it on their machines, it happens with Ethernet, SneakerNet, LocalTalk, you name it — it still crashes.

KON And people thought we'd run out of these puzzles.

BAL I got a great one for next time, too, but you've got to finish this one first.

KON So what makes my file my file? If it's not the name, the data in it, or the creation date, it's got to be the container info — the icon, the file type, the creator, that stuff.

BAL Your puzzle, KON.

KON The Desktop Manager previously used only by AppleShare is used by the Finder in System 7, right?

5 BAL Yeah, there are two invisible data files: Desktop DB and Desktop DF. Desktop DF has the all the data for the icons, and Desktop DB has information for all files: the document-to-application binding, the file-to-icon binding, and the file-to-comment binding. For each application creator, it keeps a list of all the applications of that type with the newest one first. That way when you double-click a document, the Finder launches the newest version of the application that handles that document.

KON I get it. The desktop database got corrupted by the experimental MPW tool you had, and when the Finder tries to update the database, it reads bad data and chokes.

SCORING

75–100 Excellent! Link your résumé to DEVELOP.

50–70 Not bad. Link your résumé to MACTECHMAG.

25–45 Hmmm. Better sit on that résumé for a while.

5–20 At least you beat KON's score! ♦

BAL It worked when copying files via MPW because MPW doesn't try to keep the desktop database up to date.

KON So you should be able to rebuild the desktop database by holding down the Command and Option keys while booting. Then everything will work great.

BAL I knew the AppleShare servers are the most reliable in the business, and I couldn't believe we got one to crash.

KON Well, if you want to write sleazy MPW tools that you haven't fully debugged, you get what's coming to you.

BAL In fact, for ultimate security on a network server, you should lock it in a room and not run any weird stuff that isn't endorsed by your AppleShare administrator.

KON Right! But if you insist on running untested code, you get what you pay for.

BAL Nasty.

KON Yeah.

INDEX

A

accelerator cards, Macintosh
Q & A 117
activate events, floating windows
and 92-96
ActivateFloatersAndFirstDocu-
mentWindow, floating windows
and 99-100
AddAdorner, 3-D effects and 109
AdLib, 3-D effects and 109
adorners, 3-D effects and
111-112
AESend, Macintosh Q & A 129
Alert, floating windows and 96
alerts, floating windows and 90,
96
Alexander, Pete ("Luke") 6
aliases, Macintosh Q & A
126-127
Apple events, Macintosh Q & A
129
AppleScript, QuickTime and 77
AppleShare 5, 129
Apple Sound Chip, Macintosh
Q & A 119
ashow (PostScript), QuickDraw
GX and 62
asynchronous routines, new
information on 5
attributes, QuickDraw GX and 8
awidthshow (PostScript),
QuickDraw GX and 62

B

background adorners, 3-D effects
and 109
base components, QuickTime and
85
Bézier curves, QuickDraw GX and
53-54
bitmaps, Macintosh Q & A
130-131
bitmap shape 8
Black & White option, Macintosh
Q & A 117

BringToFront, floating windows
and 93
By Page Setup, QuickDraw GX
and 11, 30, 31

C

CaptureComponent, QuickTime
and 86
CDEFs, 3-D effects and 112
CDrawPerDevice, 3-D effects and
109, 110, 111, 113
CGraphicsState, 3-D effects and
109, 110, 111
charpath (PostScript), QuickDraw
GX and 70
child view port, QuickDraw GX
and 20-21
Chooser, QuickDraw GX and 10
Collection Manager, QuickDraw
GX and 35
collections, QuickDraw GX and
35
Color/Grayscale option,
Macintosh Q & A 117
color profile object, QuickDraw
GX and 56
Color QuickDraw, floating
windows and 97, 98
ColorSync, Macintosh Q & A
121
common color library, QuickDraw
GX and 18
complete message override,
QuickDraw GX and 38
component load order resource,
QuickTime and 79-80
Component Manager
Macintosh Q & A 120
QuickDraw GX and 34-35
QuickTime and 74-83
components 74-83, 84-88
constructors, 3-D effects and 110
context switching 5, 92
asynchronous routines and 5
floating windows and 92

For a cumulative index to all issues of
develop, see this issue's CD. •

control classes, 3-D effects and
112–113
copypage (PostScript),
QuickDraw GX and 52
Count1Resources, QuickTime and
79
CreateThePageOfGXShapes,
QuickDraw GX and 29
custom fonts, Macintosh Q & A
125
CWNewColorWorld, Macintosh
Q & A 121

D

derived components, QuickTime
and 86
desktop printers 10
despooling 36
destructors, 3-D effects and 110
“Developing QuickDraw GX
Printing Extensions” (Weiss)
34–50
device communications 36
DeviceLoop, 3-D effects and 110
Device Manager, asynchronous
routines and 5
display cards, Macintosh Q & A
117
DisposeWindow, floating windows
and 96
DisposeWindowReference,
floating windows and 96, 99
Document Setup, QuickDraw GX
and 11, 30, 31
document structuring conventions,
QuickDraw GX and 65
DragReferencedWindow, floating
windows and 100
DragWindow, floating windows
and 100
Draw, 3-D effects and 109, 110,
111, 112
drivers, Macintosh Q & A 117
dynamic linking, QuickTime and
84–88

E

Encapsulated PostScript,
QuickDraw GX and 52
EnterGraphics, QuickDraw GX
and 15
'eopt' resource, QuickDraw GX
and 46, 47–48
Erase adorners, 3-D effects and
109
EraseRect, 3-D effects and
111–112
Event Manager, floating windows
and 92

F

FailedForward_GXCountPages,
QuickDraw GX and 41
File Manager, asynchronous
routines and 5
file servers, KON & BAL puzzle
132–136
File Sharing 5, 126–127
filled objects, QuickDraw GX and
70
Finder
KON & BAL puzzle
132–136
Macintosh Q & A 129, 130
QuickDraw GX and 10
QuickTime and 82
3-D effects and 107, 113
FindNextComponent, QuickTime
and 87
“Floating Windows: Keeping
Afloat in the Window
Manager” (Yu) 89–102
FOND resource, Macintosh
Q & A 125, 130
font resources, Macintosh Q & A
125, 129–130
fonts 68–69, 125, 129–130
Forward_GXCountPages,
QuickDraw GX and 41, 45
ForwardMessage, QuickDraw GX
and 45

ForwardThisMessage, QuickDraw
GX and 45
4-Up printing extension,
QuickDraw GX and 34–50
framed objects, QuickDraw GX
and 70
FrontNonFloatingWindow,
floating windows and 101
FSpCreateResFile, Macintosh
Q & A 129

G

geometric shape 7
Gestalt 14, 77
Get1IndResources, QuickTime
and 79
GetActivateHandlerProc, floating
windows and 100
GetMediaSample, QuickTime and
88
GetNewWindow, floating
windows and 96
GetNewWindowReference,
floating windows and 96, 98,
99
GetNextEvent, asynchronous
routines and 5
“Getting Started With QuickDraw
GX” (Alexander) 6–33
glyphs, QuickDraw GX and 60,
61–62, 68–69
'gnht' resource, QuickTime and
83
GraphicsBug, QuickDraw GX and
14, 33
Greg’s Buttons, 3-D effects and
112
grestore (PostScript), QuickDraw
GX and 59
Guschwan, Bill 84
GXCountPages, QuickDraw GX
and 39, 40, 41, 45
GXDespoolPage, QuickDraw GX
and 39, 40, 41–45, 48

GXDisposeGraphicsClient,
 QuickDraw GX and 32, 33
 GXDisposeShape, QuickDraw
 GX and 32
 GXDrawShape 22, 25, 28, 29, 63
 GXExitPrinting, QuickDraw GX
 and 32
 GXGetGraphicsError,
 QuickDraw GX and 13
 GXGetJob, QuickDraw GX and
 40
 GXGetJobError, QuickDraw GX
 and 30
 GXGetPictureParts, QuickDraw
 GX and 59
 GXGetShapeDrawError,
 QuickDraw GX and 12
 GXIgnoreGraphicsNotice,
 QuickDraw GX and 13
 GXIgnoreGraphicsWarning,
 QuickDraw GX and 13
 GXImagePage, QuickDraw GX
 and 39
 GXLoad, QuickDraw GX and 16
 GXMoveShape, QuickDraw GX
 and 59
 GXNewGraphicsClient,
 QuickDraw GX and 15
 GXNewShape, QuickDraw GX
 and 32
 GXPrimitiveShape, QuickDraw
 GX and 70
 GXRenderPage, QuickDraw GX
 and 39
 GXSetPictureParts, QuickDraw
 GX and 59
 GXSetShapeTextSize, QuickDraw
 GX and 61
 GXSetShapeType, QuickDraw
 GX and 23, 70
 GXSetStyleRunFeatures,
 QuickDraw GX and 27
 GXSpoolPage, QuickDraw GX
 and 39
 GXUnload, QuickDraw GX and
 16

GXValidate, QuickDraw GX and
 17

H

handles, Macintosh Q & A 131
 help balloons, floating windows
 and 90
 HideReferencedWindow, floating
 windows and 96, 99
 HideWindow, floating windows
 and 93, 99
 HiliteWindow, floating windows
 and 93
 human interface, 3-D effects and
 103–114

I

icon buttons, 3-D effects and
 105–108
 icons, Macintosh Q & A 121
 image file 36
 InitPrinting, QuickDraw GX and
 30
 INITs, QuickTime and 77, 79–82
 InitU3DDrawing, 3-D effects and
 109
 ink object, QuickDraw GX and 8,
 56
 Installer, Macintosh Q & A 130
 installer scripts, Macintosh Q & A
 130

J

Johnson, Dave 71
 Jones, Tao 115
 jump table, QuickDraw GX and
 45–46

K

Kanji 7.1, Macintosh Q & A 123
 key-down events, Macintosh
 Q & A 123
 Komponent Killer, QuickTime
 and 88

“KON & BAL’s Puzzle Page”
 (Othmer and Leak) 132–136
 kshow (PostScript), QuickDraw
 GX and 62

L

LaserWriter driver, Macintosh
 Q & A 117
 LastFloatingWindow, floating
 windows and 101
 layout shape, QuickDraw GX and
 60, 61, 62
 Leak, Bruce 132
 line adorners, 3-D effects and 109
 line groups 72–73
 line layouts, QuickDraw GX and
 26–29
 Lipton, Daniel 51
 LoadComponents, QuickTime
 and 80
 loader component, QuickTime
 and 82–83
 LoaderINIT, QuickTime and 80,
 81, 82, 83
 ‘load’ resource, QuickDraw GX
 and 46, 48

M

Macintosh Common Lisp,
 Macintosh Q & A 120–121
 Macintosh Easy Open,
 QuickTime and 77
 Macintosh Q & A 117–131
 MacsBug, QuickDraw GX and 14
 makefont (PostScript),
 QuickDraw GX and 68
 “Managing Component
 Registration” (Woodcock)
 74–83
 math types, QuickDraw GX and
 52–53
 matrix transformations,
 QuickDraw GX and 54–55
 MDEFs, Macintosh Q & A
 125–126

- MediaIdle, QuickTime and 86, 87–88
- MediaInitialize, QuickTime and 86, 87
- memory
 - Macintosh Q & A 130–131
 - QuickDraw GX and 16, 17
- Memory Manager 5, 15, 16, 33
- message class 34
- message handler 34
- Message Manager, QuickDraw GX and 34, 37, 38, 45
- message objects 34
- ModalDialog, floating windows and 96
- modal dialogs, floating windows and 90, 96
- modal windows, floating windows and 96
- modeless dialogs, floating windows and 90
- movie controller component, QuickTime and 74
- MoviePlayer, QuickTime and 84
- MoviesTask, QuickTime and 86
- MultiFinder 5, 15
- multiple segments, Macintosh Q & A 120

N

- NewCWindow, floating windows and 97
- NewGWorld, Macintosh Q & A 130–131
- NewMessageGlobals, Macintosh Q & A 118–119
- NewMovieFromFile, QuickTime and 84
- NewWindow, floating windows and 96, 97
- NewWindowReference, floating windows and 96, 97, 98, 99
- NFNT resource, Macintosh Q & A 125

- nonsquare pens, QuickDraw GX and 67–68
- notices, QuickDraw GX and 12, 13

O

- Object Pascal, 3-D effects and 110
- objects, QuickDraw GX and 70
- oblique text, QuickDraw GX and 68–69
- off-screen bitmaps, Macintosh Q & A 130–131
- OpenComponent, QuickTime and 86, 87, 88
- OpenSelection, Macintosh Q & A 129
- Osborne, Jamie 103
- Othmer, Konstantin 132
- outline text, QuickDraw GX and 69
- 'over' resource, QuickDraw GX and 46, 47
- overrides, QuickDraw GX and 37
- owner count, QuickDraw GX and 8

P

- page layout, QuickDraw GX and 51–70
- PaintRect, 3-D effects and 112
- partial message override, QuickDraw GX and 37
- paths, QuickDraw GX and 70
- PDEF, QuickDraw GX and 37
- picture, QuickDraw GX and 8
- plane groups 72
- point groups 72
- pop-up boxes/menus, Macintosh Q & A 125–126
- portable digital document 10, 36
- PostScript 11, 51–70, 117
- PPC Toolbox, asynchronous routines and 5

- primitive shape, QuickDraw GX and 70
- Print dialog, QuickDraw GX and 11
- printer drivers, Macintosh Q & A 117
- PrinterShare GX 10, 36, 39
- printing extensions 11, 34–50, 118–119
- Printing Manager 30, 35, 37, 40
- Print One Copy, QuickDraw GX and 30, 31
- Process Manager, asynchronous routines and 5

Q

- Q & A, Macintosh 117–131
- QTInitialize, Macintosh Q & A 122
- QuickDraw
 - Macintosh Q & A 117
 - QuickDraw GX and 6, 8, 11, 51
 - 3-D effects and 111
- QuickDraw GX
 - compared to PostScript 51–70
 - developing printing extensions 34–50
 - getting started with 6–33
 - Macintosh Q & A 117, 118–119
 - symmetry and 71
- QuickDraw GX heap 15, 16
- QuickDrawGXInit, QuickDraw GX and 15
- “QuickDraw GX for PostScript Programmers” (Lipton) 51–70
- QuickTime
 - customizing components 84–88
 - Macintosh Q & A 120, 122
 - registering components 74, 77, 78

QuickTime for Windows CD,
Macintosh Q & A 122
QuickTime INIT 77

R

RegisterComponent, QuickTime
and 87
Reinstaller II, QuickTime and 88
resource fork, Macintosh Q & A
125
Resource Manager, QuickTime
and 79
ResumeFloatingWindows, floating
windows and 100
RotateMatrix, Macintosh Q & A
122
rotational symmetry 72
rotation effect, Macintosh Q & A
122
rotocenter 72

S

scalefont (PostScript),
QuickDraw GX and 60–61
'scop' resource, QuickDraw GX
and 46, 48–50
ScrollRect, QuickDraw GX and
21
SCSI Manager, asynchronous
routines and 5
segments, Macintosh Q & A 120
SelectReferencedWindow, floating
windows and 96, 99
SelectWindow, floating windows
and 93, 99
SendBehind, floating windows and
93
Send_GXCountPages,
QuickDraw GX and 45
SetActiveHandlerProc, floating
windows and 100
SetGraphicsLibraryErrors,
QuickDraw GX and 16
SetGraphicsLibraryNotices,
QuickDraw GX and 16

SetMovieMatrix, Macintosh
Q & A 122
setpagedevice (PostScript),
QuickDraw GX and 52, 64
SetShapeCommonColor,
QuickDraw GX and 18
SetStylHandle, Macintosh Q & A
131
shape object, QuickDraw GX and
7, 22–30, 56
show (PostScript), QuickDraw
GX and 60, 62
ShowHide, floating windows and
93
showpage (PostScript),
QuickDraw GX and 52, 56
ShowReferencedWindow, floating
windows and 96
ShowWindow, floating windows
and 93, 99
ShutdownProgram, QuickDraw
GX and 33
skew effect, Macintosh Q & A
122
SndPlayFromDisk, Macintosh
Q & A 119
“Somewhere in QuickTime”
(Guschn) 84–88
Sound Manager, Macintosh
Q & A 119
space groups 73
SpeakText, QuickTime and 88
stack-based objects, 3-D effects
and 109–111
StackSpace, asynchronous routines
and 5
standalone code resources,
Macintosh Q & A 127–128
stroke (PostScript), QuickDraw
GX and 70
strokepath (PostScript),
QuickDraw GX and 70
style object, QuickDraw GX and
8, 56

SuspendFloatingWindows,
floating windows and 100
symmetry, Johnson ponders
71–73
symmetry group 72
synchronous drivers, asynchronous
routines and 5
SysBeep, QuickTime and 77
System 6, asynchronous routines
and 5
System 6.0.7, QuickTime and 77
System 7.0
asynchronous routines and 5
QuickTime and 77
3-D effects and 108
System 7.0.1 77, 131
System 7.1 78, 81, 123
System CDEF, 3-D effects and
113
system heap, Macintosh Q & A
129–130
system windows, floating windows
and 90

T

T3DButton, 3-D effects and 109,
113
T3DCheckBox, 3-D effects and
109
T3DFrameAdorner, 3-D effects
and 108–109, 112
T3DGrayBackgroundAdorner,
3-D effects and 108–109, 112
T3DIconButton, 3-D effects and
109, 113
T3DLineBottomAdorner, 3-D
effects and 108–109, 112
T3DLineLeftAdorner, 3-D effects
and 108–109, 112
T3DLineRightAdorner, 3-D
effects and 109, 112
T3DLineTopAdorner, 3-D effects
and 108–109, 112
T3DListFrameAdorner, 3-D
effects and 112

T3DRadio, 3-D effects and 109
 tag list, QuickDraw GX and 8
 tag objects 8, 56
 TButton, 3-D effects and 113
 TControl, 3-D effects and 108, 109, 112, 113
 TDrawingEnvironment, 3-D effects and 112
 TeachText, QuickDraw GX and 36
 TEditText, 3-D effects and 112
 tessellation 71
 text, QuickDraw GX and 22-26, 51-70
 TextEdit, Macintosh Q & A 131
 text media handler, QuickTime and 84-88
 text shape, QuickDraw GX and 60
 Text-to-Speech Manager, QuickTime and 84-88
 TGrayBackgroundAdorner, 3-D effects and 108-109, 111, 112
 'thld' resource, QuickTime and 79
 'thng' resource, QuickTime and 77, 79, 80, 83
 Thomas, Deanna 103
 three-dimensional effects, MacApp and 103-114
 TIconSuite, 3-D effects and 113
 tile shapes, Johnson ponders 71-73
 Time Manager, asynchronous routines and 5
 TListView, 3-D effects and 112
 transform object, QuickDraw GX and 8, 56
 TrueType GX font, QuickDraw GX and 10
 TStaticText, 3-D effects and 112
 TView, 3-D effects and 110, 112
 TWhiteBackgroundAdorner, 3-D effects and 108-109, 112

Type 1 GX font, QuickDraw GX and 10
 typographic shape 7

U

user interface, 3-D effects and 103-114

V

validation, QuickDraw GX and 17
 “Veteran Neophyte, The” (Johnson) 71-73
 view device 8
 ViewEdit, 3-D effects and 109
 view editors, 3-D effects and 109
 “View From the Ledge” (Jones) 115-116
 view group 8
 view port objects, QuickDraw GX and 56
 view ports 8, 18-21
 'View' resource, 3-D effects and 109

W

WaitNextEvent, asynchronous routines and 5
 warnings, QuickDraw GX and 12, 13
 WDEF resource, floating windows and 97
 Weiss, Sam 34
 Window Manager, floating windows and 89-102
 windows, QuickDraw GX and 18-22
 WIND resource, floating windows and 98, 99
 Woodcock, Gary 74
 “Working in the Third Dimension” (Osborne and Thomas) 103-114
 WorldScript, Macintosh Q & A 123-125

X

xshow (PostScript), QuickDraw GX and 62
 xyshow (PostScript), QuickDraw GX and 62

Y, Z

y-axis, QuickDraw GX and 55
 yshow (PostScript), QuickDraw GX and 62
 Yu, Dean 89