OpenDoc Memory Management • Safe Printing • Source Code Control

develop

The Apple Technical Journal

Issue 28 December 1996

Coding Your Object for Advanced Scriptability

New QuickDraw 3D Geometries

QuickDraw GX Line Layout: Bending the Rules

MacApp Debugging Aids

Chiropractic for Your Misaligned Data

$10.00

develop

EDITORIAL STAFF

ART & PRODUCTION

ISSN #1047-0735.

Printed on recycled paper by Stream International, USA

THINGS TO KNOW

develop, The Apple Technical Journal, a quarterly publication of the Apple Developer Relations group, is published in March, June, September, and December. It provides developers of Apple-platform products with technical articles and code that have been reviewed for robustness by Apple engineers.

All issues of *develop*, along with the code they describe, can be found on the *develop Bookmark* CD, the Reference Library edition of the *Developer CD Series*, and the Internet. The code is updated regularly, so always use the latest version.

This issue's CD. Subscription issues of *develop* are accompanied by the *develop Bookmark* CD. This CD contains a subset of the materials on the *Developer CD Series*, which is part of the Apple Developer Mailing available through the *Apple Developer Catalog*. The CD also contains Technotes, sample code, and other documentation and tools (these contents are subject to change). Items referred to as being on "this issue's CD" are located either on the Bookmark CD or on the Reference Library or Tool Chest edition of the *Developer CD Series*.

Subscriptions and back issues. You can subscribe to *develop* through the *Apple Developer Catalog* (see below, or use the subscription card in this issue). Back issues, in addition to being available electronically, can also be ordered through the catalog. The one-year U.S. subscription price is $30 (for four issues and four *develop Bookmark* CDs), or U.S. $50 in other countries. Back issues are $13 each. These prices include shipping and handling. For Canadian orders, the subscription price includes GST (R100236199).

WHERE TO FIND US

What	Who/where
develop on the Web and ftp	http://www.devworld.apple.com/develop/ ftp://ftpdev.info.apple.com/Developer_Services/Periodicals/develop/
Bookmark CD contents	http://www.devworld.apple.com/ ftp://ftpdev.info.apple.com/Developer_Services/
Technotes on the Web	http://www.devworld.apple.com/dev/technotes.shtml
Editorial comments or suggestions	Caroline Rose, crose@apple.com, (408)974-0544 fax
Technical questions about *develop*	Dave Johnson, dkj@apple.com, (408)974-0544 fax
Article submissions	develop@apple.com (ask for our Author's Kit)
Apple Developer Catalog (*develop* subscription, back issues, or other products)	http://www.devcatalog.apple.com order.adc@applelink.apple.com 1-800-282-2732 U.S., 1-800-637-0029 Canada (716)871-6555 internationally, (716)871-6511 fax *Apple Developer Catalog*, Apple Computer, Inc., P.O. Box 319, Buffalo, NY 14207-0319
Subscription changes or queries	order.adc@applelink.apple.com *Please be sure to include your name, address, and account number as they appear on your mailing label.*

ARTICLES

COLUMNS

EDITOR'S NOTE

CAROLINE ROSE

Regular readers of *develop* may know that, while I always use the latest hardware and software at work, my home system is woefully out of date — at least it was until I recently upgraded to a Power Mac. The initial setup wasn't too hard, once I got used to flipping back and forth between the manual for the computer, the manual for the monitor, and the online updates; when all else failed, I relied on common sense. But when I moved beyond hardware setup into software installation, it seemed as if I was expected to know much more than I did. The modem software installer, for example, would just name a software module and say "click OK if you want to install this," with no mention of which module(s) provided the basic modem capabilities (which is all I wanted). What to do? I recently had this same feeling while trying to learn a new e-mail application at work: though I immediately saw how to address mail to anyone on any network in the known world with barely a keystroke, I couldn't tell how to simply enter an address for someone here at Apple. In these and many similar cases I've encountered recently, I couldn't figure out how to perform basic application functions without the intervention of an experienced user. The manuals and online help were somewhat helpful, but they were limited by the design of the software, which was the problem in the first place. Common sense was no help at all.

Naturally I griped about this to my "friends in the industry." I think one of them hit the nail on the head when he said the problem is that too many products are being designed by experts who, consciously or not, design for experts. Designing with experts in mind ends up complicating everything, even the features that should be simple. Bud Tribble, when he managed the software group at NeXT, used to tell programmers, "Simple things should be simple, and complex things should be possible." It seems increasingly true these days that designers are trying to make complex things simple, but as a result are making simple things complex. Design by experts for experts is not the answer: developers need to find out what real users want, and focus on *their* needs.

Is *develop* guilty of a similar problem? When I looked at the feedback we gathered at Apple's Worldwide Developers Conference this year, I noticed that some *develop* readers are asking for more entry-level articles, saying that a lot of what we publish is over their heads. While we're limited by what types of articles are submitted to us for publication, the *develop* Review Board does get to say which ones are accepted or not, and the Board is largely made up of "expert" programmers. Do we consequently tend to decide in favor of the more advanced articles? We'll keep an eye out for this from now on.

I often recall the days when Steve Jobs envisioned that the Macintosh would be as easy to use as a home appliance. Sure, we don't want to go back to that first oversimplified product, but maybe we should all ponder whether we've gone a bit too far in the opposite direction. I'd like to believe there's still a place for common sense.

Caroline Rose
Editor

CAROLINE ROSE (crose@apple.com) has been a Mac enthusiast ever since she started writing the original *Inside Macintosh* in 1982. After a reorganization that suddenly changed the entire managerial hierarchy above her (up to and including Steve Jobs), she left Apple, but like so many other formerly disgruntled employees, she eventually returned. This year, again, a reorg happened that changed the entire managerial hierarchy above Caroline. At least there's some stability in her home life, where her cat Cleo remains the boss after 15 long years.

LETTERS

GAME OUT OF CONTROL

When running the code from Philip McBride's article "Game Controls for QuickDraw 3D" in Issue 27 of *develop*, I noticed in MySetCameraData that the call to Q3Camera_SetPlacement gets fouled up after a while. The fix shown below was needed.

```
void MyGetCameraData(
   DocumentPtr theDocument,
   TQ3CameraObject theCamera)
{
   TQ3CameraPlacement cameraPlacement;
   . . .
   Q3Vector3D_Cross(
      &theDocument->zVector,
      &theDocument->yVector,
      &theDocument->zVector);

   // I added this:
   Q3Vector3D_Normalize(
      &theDocument->xVector,
      &theDocument->xVector);
}
```

— Flip Phillips

You made the right call. QuickDraw 3D requires all vectors to be of unit length (normalized). So any changes to vectors that could make them not normalized should be followed by a call to normalize those vectors (using Q3Vector3D_Normalize as you've done). If QuickDraw 3D didn't have this requirement, it would have to normalize vectors itself internally, which would take away from its efficiency.

— Philip McBride

A SIDE ISSUE

In Steve Falkenburg's article "Planning for Mac OS 8 Compatibility" in *develop* Issue 26, I read the following:

> *After some poking around, we figured out that it was assuming that InitWindows called InitMenus as a side effect. The completely new Mac OS 8 implementations of windows and menus no longer have this behavior.*

However, *Inside Macintosh* Volume V states, under the heading "InitWindows" on page 208:

> *Since the menu bar definition procedure ('MBDF') actually performs these calculations, InitWindows now calls InitMenus directly. InitMenus has been modified so that it can be called twice in a program without ill effect.*

I advise the Mac OS 8 engineers to read the "You're never too smart to read the manual" section of the Veteran Neophyte column in the same issue of *develop* ;-).

— Reinder Verlinde

Your point is well taken. However, it's worth mentioning that Inside Macintosh: Macintosh Toolbox Essentials (the more current documentation) says no such thing, and in fact says explicitly that InitMenus should always be called after InitWindows. As always, depending on a side effect, even a documented one, is dangerous.

— Dave Johnson

MORE GOOD STUFF

Look on this issue's CD and *develop*'s Web site for Martin Minow's article, "Timing on the Macintosh," which was recently updated to include a few Java timing techniques. You'll also find some scripting vocabulary advice from Cal Simone.

THINK THESE LETTERS ARE BORING?

Then why not write one of your own? We welcome your letters to the editor, especially regarding articles published in *develop*. Write to Caroline Rose at crose@apple.com or, if technical *develop*-related questions, to Dave Johnson at dkj@apple.com. 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). Address subscription-related queries to order.adc@applelink.apple.com.

Coding Your Object Model for Advanced Scriptability

Implementing an Apple event object model makes your application scriptable — that is, it enables users to control your application with AppleScript or some other OSA-compliant language. You can provide anything from a basic implementation of the object model that handles only simple scripts to a full implementation that can handle the most complex scripts. This article will help you do the latter. It will show you how to write object accessors and handlers that process lists of objects, alert you to some common pitfalls, and suggest other features you can add for completeness.

RON REUTER

You've decided to give your users an alternate interface for controlling your application by implementing an Apple event object model. You've read Richard Clark's article, "Apple Event Objects and You," in *develop* Issue 10 to get an overview of the Apple event object model and how to support it. As you've begun to think about your scripting vocabulary, you've absorbed Cal Simone's article, "Designing a Scripting Implementation," in *develop* Issue 21 and his According to Script column in Issue 24. You've checked out the portions of *Inside Macintosh: Interapplication Communication* that apply. You've read and understood the *Apple Event Registry*, which defines the primary events and objects that you should support in a scriptable application, and have paid particular attention to the Core, Text, and QuickDraw Graphics suites.

With this basic knowledge, you're ready to read this article. Here you'll learn how to structure your code to handle more complex user scripts. After a brief review of the components of an object model implementation, I'll focus on object accessors and show you how to handle script statements that require your code to act on a list of objects. Then I'll describe in detail how to deal with three big "gotchas" that are bound to trip you up unless you know about them. Finally, I'll tell you about some other goodies you can implement for the sake of completeness. All of this is illustrated in the sample application Sketch, which demonstrates object model support for a subset of the QuickDraw Graphics suite. The code for Sketch, which accompanies the article on this issue's CD and *develop*'s Web site, contains many functions that you can use when you get ready to code the object model for your own application.

RON REUTER (rlreute@uswest.com) is a software developer for USWEST Media Group, "a leading provider of Yellow Pages and interactive multimedia information services" or, as it used to be known, the phone company. He's spent the last two years as technical lead on a team that's developing a graphics package to be used across a fourteen-state region to build ads for USWEST Yellow Page directories. Programming is Ron's sixth career; he's also been a offset press operator, a furniture maker, a luthier, a traveling jewelry salesman, and a Zen student — but he'd rather be dancing beneath the diamond sky with one hand waving free.•

COMPONENTS OF AN OBJECT MODEL IMPLEMENTATION

The components of an object model implementation are outlined in the "Apple Event Objects and You" article and discussed in great detail in *Inside Macintosh: Interapplication Communication*. Here I'll briefly review the basic terms and concepts to refresh your memory and to show how they apply in our sample program.

When a script statement asks your application to perform an action on some object, such as closing document 1, the object specifier (document 1) must be resolved — that is, the representation of the specified object must be located in memory. Your application resolves the object specifier by way of an object accessor function that converts the object specifier into a token. The token is then passed to an event dispatcher for that object. I'll describe each of these components before reviewing the process of resolving object specifiers and dispatching events.

Object accessors are functions you write and install in an accessor table. These functions are called by the Object Support Library (OSL) function AEResolve when the Apple Event Manager needs to find some object in your application's data structures. Object accessors receive a container, an object specifier for an object to locate inside that container, and a result parameter into which a token is placed. When you install accessors, you tell the Apple Event Manager that your application knows how to find a certain kind of object in a certain kind of container. For instance, you know how to find a rectangle object in a grouped graphic object or a word object in a paragraph object. (More on containers in a minute.)

A *token* is an application-defined data structure that is populated in your object accessors and is passed later to your object's event dispatcher code, where it's used to find the object that an Apple event will be applied to. The structure and content of a token are private to the application; neither the Apple Event Manager nor the OSL attempts to interpret or use the contents of a token. The Sketch sample application uses a single token structure, shown below, for all of its objects. Note that some fields aren't used for all object types and that the token doesn't contain the object's data or a data value; it contains information about how to locate the object later. You can use a single token structure in your implementation, or you may want to design a unique token structure for each object you support.

```
typedef struct CoreTokenRecord {
    DescType    dispatchClass;      // class that will handle an event
    DescType    objectClass;        // actual class of this object
    DescType    propertyCode;       // requested property code,
                                    // or typeNull if not a property token
    long        documentNumber;     // unique ID for the document, or 0
    long        elementNumber;      // unique ID for the element, or 0
    WindowPtr   window;             // used for window objects only
} CoreTokenRecord, *CoreTokenPtr, **CoreTokenHandle;
```

Event dispatchers are application-defined functions that you call after you've called AEResolve and your object accessors have returned a token for the target of the Apple event. You call your event dispatchers and pass the token you created, the original Apple event, and the reply Apple event you received in your Apple event handler. The event dispatcher examines the Apple event, extracts the event ID, and passes its parameters on to a specific event handler for the token object. The token is used to identify the object or objects that the Apple event should act on.

Apple event handlers are functions you write and install that receive a specific Apple event and a reply Apple event. Your event handlers extract parameters from the Apple event, process the event using those parameters, and place the result in the reply Apple event.

A single handler can be installed to handle many events. For example, one handler can receive all events in the Core suite, except the Create Element event, if you specify kAECoreSuite for the event class and typeWildCard for the event ID. Because Create Element passes an insertion location instead of an object specifier in the direct object parameter, Sketch installs a separate handler, AECreateElementEventHandler, to handle this event for all Core suite objects.

The Sketch sample code resolves object specifiers and dispatches Core suite Apple events by using the object-first approach. The object-first flow of control proceeds as follows:

1. In the Core suite event handler, extract the parameter for the direct object of the Apple event. Except in the case of the Create Element event, this is a reference to some object the user is trying to access or modify.
2. Call AEResolve, which calls one or more of your object accessor functions, each of which finds the requested object in a specified container and then returns a token. AEResolve successively calls object accessors until one of the following three conditions is met: you return a token for the specified object; you return an error code; or AEResolve finds a container-element combination for which there's no installed accessor.
3. Examine the token to determine what kind of object it references and then send the original event, the reply event, and the token to the event dispatcher for that type of object.
4. In the object's event dispatcher, extract the event ID and dispatch the event and the token to the object's event handler.
5. In the object's event handler, apply the event to the object or objects referenced by the token and return the results in the direct object of the reply Apple event. You usually just unpack the parameters in your event handler and then call lower-level functions you've written to do the application-specific work.

Figure 1 shows how this approach is applied as Sketch processes the script statement

```
set fill color of rectangle 1 of document 1 to blue
```

Note that Sketch has one file for each type of scriptable object. Figure 1 shows fragments of three files:

- AECoreSuite.c, which receives all Apple events from the Core suite, resolves the direct object parameter, and dispatches the token and the Apple event to the dispatcher for a specific object type
- OSLClassDocument.c, which contains accessors, a dispatcher, and event handlers for document objects
- OSLClassGraphicObject.c, which contains accessors, a dispatcher, and event handlers for all graphic objects

OBJECT ACCESSORS AND YOUR CONTAINMENT HIERARCHY

How you implement your object model will depend largely on the nature of your data and on your containment hierarchy. Your containment hierarchy specifies the objects you support, how script statements should address those objects, and which objects are contained by which other objects. Contained objects are called *elements* of the container object. Each object also usually contains one or more *properties*, which

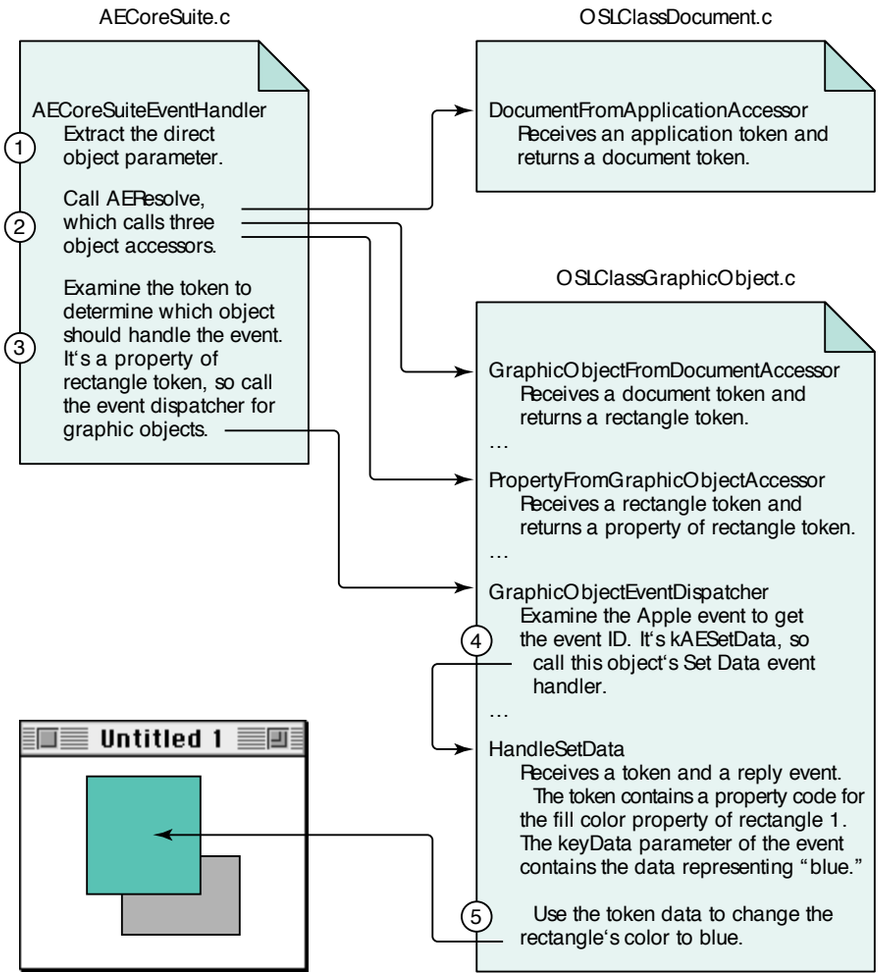


Figure 1. How Sketch processes **set fill color of rectangle 1 of document 1 to blue**

represent that object's characteristics, such as font size or color. While an object can contain many elements of a particular type, it contains only one of each of its properties. A script identifies the object to inspect or change by way of an *object reference*, which specifies the object's location in the containment hierarchy.

Sketch has the containment hierarchy shown in Figure 2. The application object can contain both windows and documents. Documents, in turn, contain objects defined in the QuickDraw Graphics suite, such as rectangles, ovals, graphic lines, and graphic groups. Graphic groups can contain any object from the QuickDraw Graphics suite, including other graphic groups.

The following complete script navigates through Sketch's containment hierarchy from top to bottom to get a property of an object:

```
tell application "Sketch"
   tell document "Sales Chart"
      tell rectangle 1
         get fill color
      end tell
   end tell
end tell
```

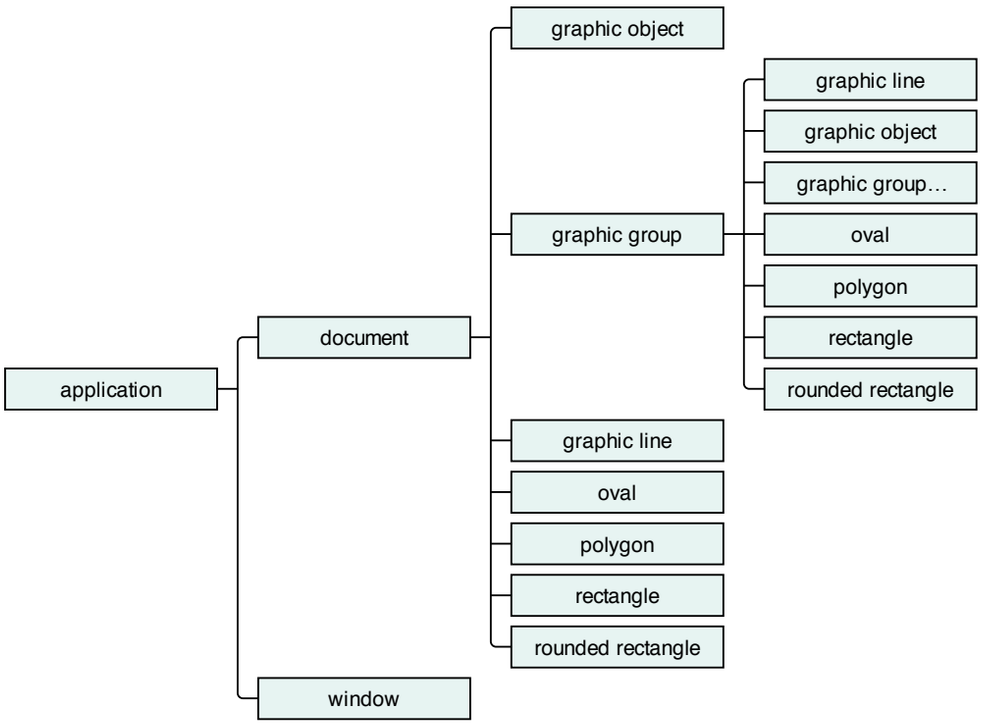


Figure 2. The Sketch containment hierarchy

Throughout the rest of this article, I'll usually show script fragments consisting of a single statement instead of complete scripts.

The desired object can be specified in one of several ways in a script statement, as you'll see later in the discussion of key forms. Theoretically, for each container-element combination in your containment hierarchy, you need an object accessor function that can find the element type in its container type. In reality, you frequently can get by with a single object accessor function that can handle many container-element pairs, rather than having to write and install a separate function for each one.

The *Apple Event Registry* lists the elements that can be contained within each object it defines. Recursive definitions occur frequently in the Registry. For example, the word object in the Text suite can contain characters, words, lines, paragraphs, and text. While it seems reasonable that a word can contain a character, when would a word contain a line or a paragraph? Suppose the script asked to do something to words 1 through 200. This is an example of a range specifier, which we'll look at in more detail later. Your application might resolve this range specifier into a list of 200 word objects. Because there could be many paragraphs within that range, asking for paragraph 2 of words 1 through 200 would make sense. It's to support range specifiers that every text object is required to be an element of every other text object.

The upshot of this is that to support the word object in the Text suite, you would need to write object accessors to resolve all these possible containment scenarios: word-from-character, word-from-word, word-from-line, word-from-paragraph, word-from-text, and either word-from-document (for a text editor that supports one large text object per document) or word-from-graphic-text (for a drawing application that supports many text boxes per document). As mentioned earlier, though, you frequently can get by with a single object accessor function that can handle many container-element pairs. Sketch, for example, uses just two object accessors to support all objects in the QuickDraw Graphics suite: GraphicObjectFromDocumentAccessor and

GraphicObjectFromGroupAccessor, both of which call GraphicObjectAccessor to do the real work of finding a graphic object.

OBJECT ACCESSORS AND KEY FORMS

Script statements can ask for an object or a collection of objects in a variety of ways. They can ask for a single object by its unique ID, by name, or by its absolute or relative position in a container. A script can also ask not for an object, but for some property of an object, such as the fill color of a rectangle or the font of a paragraph. A script statement can ask for more than one object by using the word **every**, by specifying a range between some object and some other object in a container, or by specifying a test that the objects must satisfy. The method that's used to reference an object or objects in a script determines the keyForm parameter that an object accessor function will receive when it comes time to resolve the object specifier.

When an object accessor receives one of the simple key forms and associated key data types listed in Table 1, it returns a descriptor containing a token that references a single object in your application. When it receives one of the complex key forms and associated key data types listed in Table 2, it returns a descriptor containing a list of tokens, each of which references a single object.

Table 1. Simple key forms

Key form	Key data type	Key data value
formUniqueID	typeLongInteger	A unique number
formPropertyID	typeEnumerated	An identifier declared in your 'aete' resource
formName	typeIntlText	The name of an object, such as a document
formAbsolutePosition	typeLongInteger typeAbsoluteOrdinal	A positive or negative number kAEFirst, kAEMiddle, kAELast, kAEAny
formRelativePosition	typeEnumerated	kAENext, kAEPrevious

Table 2. Complex key forms

Key form	Key data type	Key data value
formAbsolutePosition	typeAbsoluteOrdinal	kAEAll
formRange	typeRangeDescriptor	See section "Handling formRange."
formTest	See section "Handling formTest and formWhose."	
formWhose	See section "Handling formTest and formWhose."	

Note that not all key forms are appropriate for all classes — a rectangle might not have a name, for example, and some objects, such as a word or a paragraph, might not have a unique ID. According to *Inside Macintosh: Interapplication Communication*, if a key form isn't supported for an object in one of your containers, you should return errAEEventNotHandled. But you might want to return a more specific error code, such as errAEBadKeyForm or errAENoSuchObject.

HANDLING SIMPLE KEY FORMS

Handling the simple key forms is mostly straightforward. Table 3 shows some examples of script fragments using simple keys and their results. For these examples,

assume the script is looking at a text block that contains the words “Hi there” in 12-point Helvetica type. For the first two examples, formAbsolutePosition is the key form; for the third example, the key form is formPropertyID, and for the fourth example it’s formRelativePosition.

Table 3. Script fragments using simple keys and their results

Script fragment	Result type	Example result
word 2	word	"there"
character 1 of word 2	character	"t"
size of word 1	number (font size)	12
word before word 2	word	"Hi"

Although formRelativePosition is a simple key form, there’s one aspect of handling it that might not be obvious. The container parameter that your object accessor receives in this case is a reference not to a container but to an object inside a container in relation to another object inside that container. In other words, if a script asks for an object before or after another object in a container, as in

```
get name of the window after window "Sales Chart"
```

your object accessor will receive a keyForm parameter of formRelativePosition and a keyData parameter that contains a constant, either kAENext or kAEPrevious. Your accessor must then find the object either before or after the “contained” object. This means that to handle formRelativePosition, you’ll have to install an accessor that gets an object of one type from another object of the same type.

Although the containment hierarchy for Sketch shows that windows don’t contain other windows, you *will* need a window-from-window accessor installed to handle formRelativePosition. If your accessors can find an object in a container, finding an object either before or after that object should be relatively easy, as long as you remember to install the accessor. Here’s how Sketch installs the accessor for its window object:

```
error = AEInstallObjectAccessor(cWindow, cWindow,
    NewOSLAccessorProc(WindowFromApplicationAccessor), 0L, false);
```

HANDLING EVERY

If a script asks for every one of a certain kind of object, your accessor will receive a keyForm parameter of formAbsolutePosition and a keyData parameter with a descriptor type of typeAbsoluteOrdinal and a value of kAEAll, and you’ll return a descriptor that represents a collection of objects. The Sketch application returns an AEList of tokens that reference each object. Some examples of script fragments using **every** and their results are shown in Table 4. Again, assume the script is looking at a text block that contains the words “Hi there” in 12-point Helvetica type.

Each **every** specifies another list level: one **every** will return a list, two will return a list of lists, and so on. Consider, for instance, this statement that navigates through the Text suite hierarchy:

```
get every character of every word of every paragraph of every document
```

Table 4. Script fragments using **every** and their results

Script fragment	Result type	Example result
every word	List of words	{"Hi", "there"}
character 1 of every word	List of characters	{"H", "t"}
every character of every word	List of list of characters	{{"H", "i"}, {"t", "h", "e", "r", "e"}}
font of character 1 of every word	List of strings	{"Helvetica", "Helvetica"}
size of every character of every word	List of list of numbers	{{12, 12}, {12, 12, 12, 12, 12}}

An application could handle this statement by returning a descriptor containing a four-level list of character tokens. Alternatively, an application could return a flat list (a single-level list of objects all concatenated together), but I don't recommend this practice because it assumes that the information about the deep structure that's thrown away won't be needed for any subsequent processing in the script, and there's really no way to know that reliably.

AEResolve and your individual object accessors have no way to know how deep a list will end up being, but your code that handles the Apple event after the object resolution has been completed must do the right thing with a descriptor referencing a single object and with a descriptor that contains arbitrarily deep lists of such objects.

HANDLING FORMRANGE

If the script asks for objects between some object and some other object in a container, your object accessor for that container will receive a keyForm parameter of formRange. There are many ways to specify a range of objects in a script:

```
get the fill color of rectangles 1 through 3
get the location of windows from window "Hello" to window 4
get the bounds of graphic objects from oval 1 to rectangle 3
```

Note that the beginning and ending objects can be specified with different key forms and that they might even be two different object types, as in the third example. Regardless of how they're specified, you need to resolve the two object specifiers and return a descriptor containing a list of tokens for the objects from the first through the last object in the range.

Your object accessors are called three times to completely resolve a formRange statement. On the first call to an object accessor, you receive a key form of formRange and key data that contains a typeRangeDescriptor record. In Sketch, this information is passed on to the ProcessFormRange function, shown in Listing 1. ProcessFormRange begins by coercing the range record into a regular record, which will then contain two object specifiers. Next, it extracts the first descriptor from the record and calls AEResolve, which calls your object accessors again to get a token for the first object in the range. Finally, ProcessFormRange extracts the second descriptor and calls AEResolve again to get a token for the last object in the range. ProcessFormRange is called from your object accessor, and when it returns you'll have tokens for the two boundary objects in the range. Your object accessor then builds a list of all objects in the range and returns that list in the result token.

HANDLING FORMTEST AND FORMWHOSE

If the script asks for objects that satisfy some test, such as

```
get the fill color of every rectangle whose rotation is 45
```

Listing 1. Resolving the boundary objects for a range request

```
OSErr ProcessFormRange(AEDesc *keyData, AEDesc *start, AEDesc *stop)
{
   OSErr    error;
   AEDesc   ospec = {typeNull, NULL};
   AEDesc   range = {typeNull, NULL};

   // Coerce the range record data into an AERecord.
   error = AECoerceDesc(keyData, typeAERecord, &range);
   if (error != noErr) goto CleanUp;

   // Resolve the object specifier for the first object in the range.
   error = AEGetKeyDesc(&range, keyAERangeStart, typeWildCard, &ospec);
   if (error == noErr && ospec.descriptorType == typeObjectSpecifier)
      error = AEResolve(&ospec, kAEIDoMinimum, start);
   if (error != noErr) goto CleanUp;
   AEDisposeDesc(&ospec);

   // Resolve the object specifier for the last object in the range.
   error = AEGetKeyDesc(&range, keyAERangeStop, typeWildCard, &ospec);
   if (error == noErr && ospec.descriptorType == typeObjectSpecifier)
      error = AEResolve(&ospec, kAEIDoMinimum, stop);

CleanUp:
   AEDisposeDesc(&ospec);
   AEDisposeDesc(&range);
   return error;
}
```

you'll return a descriptor containing a list of tokens referencing those objects. Fortunately, once you've added support for list processing, you only need to install two functions to gain the incredible power of **whose** statements: an object-counting function and an object-comparison function. The object-counting function counts the number of objects of a specified class in a specified container. Let's say that your document has three rectangles that are rotated to 45 degrees, and another three that aren't rotated. When the OSL calls your counting function, you return 6, the total number of rectangles in the document container. Now the OSL knows that it has to call your object-comparison function six times, once for each rectangle.

The object-comparison function is given two descriptors and a comparison operator and returns true if the two descriptors satisfy the comparison operator, or false if they don't. For the example above, one descriptor will be an object specifier, such as rotation of rectangle 1, and the second descriptor contains the raw data, 45. You need to resolve the first descriptor, a formPropertyID reference, to get the rotation value for that object. Then you use the comparison operator to compare the resolved property value with the raw comparison data. If the comparison is valid, you return true; otherwise, you return false. When you return true, the OSL adds the token representing the rectangle under consideration to a list of objects that satisfy the test. To make sure the OSL handles formTest and formWhose for you in this way, be sure to specify kAEIDoMinimum as the second parameter to AEResolve.

Because you can have only one counting function and one comparison function installed, they need to be able to work with all of your container types and all the

object types you support. The good news is that if you've added support for basic object model scriptability, you've already got most of the functions spread around that do most of the work you'll need to do in your counting and comparison callbacks. Sketch includes both an object-counting function and an object-comparison function, plus a variety of comparison functions for different data types.

Depending on the OSL to handle **whose** clauses in this way has one drawback — it can be inefficient when there are a large number of objects. The OSL will call your accessors to find each object and then it will apply the comparison to each one. If you find that this is too slow, you can go the extra mile and handle resolution of **whose** clauses yourself. For details, see "Speeding Up **whose** Clause Resolution in Your Scriptable Application" by Greg Anderson in *develop* Issue 24.

INTRODUCING THE THREE BIG GOTCHAS

Handling the key forms and the lists your object accessors can return goes a long way toward making an object model implementation capable of handling complex user scripts. But there's more you need to do — namely, you have to know about the three big gotchas so that you can avoid getting into trouble with them.

I first encountered the gotchas while I was taking the "Programming Apple Events" course at Apple Developer University. The instructor, James Sulzen, was showing us some slides when he boldly exclaimed, "And here is the most important slide in the course!" It was the slide listing the gotchas I'm about to describe. But I didn't discover just how correct his pronouncement was until sometime later, when I'd read the Registry several times over and had started implementing an object model for a high-end graphics package. Simply stated, the gotchas are these:

- Any Apple event parameter can be an object specifier.
- Any resolution can return either a descriptor containing a token for a single object or a descriptor containing a list of tokens.
- The meaning of a token's contents must be preserved during the execution of an Apple event that uses that token.

I'll explain each of these and describe what you need to do in your code to deal with them.

GOTCHA #1: THE "ANY PARAMETER" GOTCHA

Any Apple event parameter can be an object specifier.

To help you grasp the implications of this gotcha, let's look first at a script that results in sending your application a keyData parameter that's *not* an object specifier:

```
set stroke size of rectangle 1 of graphic group 2 to 3
```

In your 'aete' resource, you've included the QuickDraw Graphics suite that defines a rectangle object and its stroke size property. When a script sends the above statement to your application, your accessors will be called to find rectangle 1. In this example, accessors for document-from-application, group-from-document, and rectangle-from-group will be called. The last accessor, the one that actually finds the rectangle, returns a token that will allow your event handler to find this specific rectangle later. Next, since AEResolve has done its work, your Core suite dispatcher examines the type of object the token refers to and dispatches it to the appropriate object's event dispatcher.

Your event dispatcher looks at the Apple event ID and determines that it's a Set Data event, so it calls the object's Set Data event handler, passing in the token returned from your object accessor, the original event, and the reply event. In the object's event handler, you examine the token to determine that it references the stroke size property of a particular rectangle, and you examine the Apple event to extract the keyData parameter, which contains the value 3. Finally, you update the data structure that represents that rectangle, setting the stroke size to 3, and probably do something to generate an update event so that the screen is redrawn to show the rectangle's new visual appearance.

Now, suppose the user typed a slightly different statement:

```
set stroke size of rectangle 1 to the stroke size of oval 2
```

This time the keyData parameter isn't a simple number like 3 but is instead an object specifier, stroke size of oval 2. There's only one way to convert this to a value to use to set the stroke size of rectangle 1 — you have to resolve the keyData parameter. You first have to resolve the object specifier to acquire a token that references the stroke size of oval 2, and then, since you need the actual value of that property for the Set Data event, you must use that token and emulate a Get Data event to extract that value from oval 2.

How to deal with gotcha #1. Again, gotcha #1 says *any* parameter to an Apple event can be an object specifier. Since this is the case, we might as well write a generic function that extracts parameters from an Apple event and that can handle parameters that contain raw data as well as parameters that contain object specifiers. Sketch uses this approach, calling its ExtractKeyDataParameter function from its Set Data event handlers.

The ExtractKeyDataParameter function, shown in Listing 2, extracts the key data from the Apple event without changing its form. It then passes that data to the ExtractData function (Listing 3), which looks at the descriptor type and calls AEResolve if it determines that the source parameter contains an object specifier. ExtractData can receive an object specifier, an object token, a property token, or raw data (text, number, and so on); it converts whatever it receives into raw data and returns that. Besides being called from ExtractKeyDataParameter, it's also called by the OSLCompareObjectsCallback function, which is used to resolve **whose** clauses.

Listing 2. Extracting the keyData parameter from an Apple event

```
OSErr ExtractKeyDataParameter(const AppleEvent *appleEvent,
    AEDesc *data)
{
    OSErr     error = noErr;
    AEDesc    keyData = {typeNull, NULL};

    error = AEGetKeyDesc(appleEvent, keyAEData, typeWildCard, &keyData);
    if (error == noErr)
        error = ExtractData(&keyData, data);

    AEDisposeDesc(&keyData);
    return error;
}
```

Listing 3. Extracting raw data from a descriptor

```
OSErr ExtractData(const AEDesc *source, AEDesc *data)
{
   OSErr        error = noErr;
   AEDesc       temp = {typeNull, NULL};
   DescType     dispatchClass;

   if ((source->descriptorType == typeNull) ||
       (source->dataHandle == NULL)) {
      error = errAENoSuchObject;
      goto CleanUp;
   }

   // If it's an object specifier, resolve it into a token;
   // otherwise just copy it.
   if (source->descriptorType == typeObjectSpecifier)
      error = AEResolve(source, kAEIDoMinimum, &temp);
   else error = AEDuplicateDesc(source, &temp);
   if (error != noErr) goto CleanUp;

   // Next, determine which object should handle it, if any.
   // If it's a property token, get the dispatch class.
   // Otherwise, it's either an object token or raw data.
   if (temp.descriptorType == typeProperty)
      dispatchClass = ExtractDispatchClassFromToken(&temp);
   else dispatchClass = temp.descriptorType;

   // If it's a property token, get the data it refers to;
   // otherwise just duplicate it.
   switch (dispatchClass) {
      case cApplication:
         error = errAEEventNotHandled;
         break;
      case cDocument:
         error = GetDataFromDocumentObject(&temp, NULL, data);
         break;
      case cWindow:
         error = GetDataFromWindowObject(&temp, NULL, data);
         break;
      case cGraphicObject:
         error = GetDataFromGraphicObject(&temp, NULL, data);
         break;
      default:
         // This is raw data or a nonproperty token.
         error = AEDuplicateDesc(&temp, data);
         break;
   }

CleanUp:
   AEDisposeDesc(&temp);
   return error;
}
```

There are some circumstances where extracting raw data isn't the correct thing to do, as in

```
set selection of application "Sketch" to oval 2
```

In this case, we just want to return the token for oval 2, not some property data as in the previous example. To handle this case, ExtractData checks to make sure that the token's propertyCode field doesn't contain typeNull before we dispatch the token to one of the GetDataFrom functions. If it isn't a property token, we just return the token itself and not its data.

GOTCHA #2: THE "ANY RESOLUTION" GOTCHA

Any resolution can return either a descriptor containing a token for a single object or a descriptor containing a list of tokens.

As noted earlier, the presence of **every** in a script statement, or a range request, or a **whose** statement all require that you generate and return a descriptor containing a list of tokens. Let's look at a script statement and follow the resolution process as it calls each of our accessors in turn. Here's the statement:

```
get every character of word 2 of every line of paragraph 2 of document 1
```

Let's assume document 1 looks like this:

Hello there!¶
This text block contains three lines
and two of them are long but one
is not.¶

In the Core suite, AEResolve works from the top of the containment hierarchy down to the requested object, so in our example it first calls the document-from-application accessor, which returns a token identifying the frontmost document. I'll introduce a notation here, where a letter refers to the object type, and a number refers to an index, so "D1" means "document 1."

```
resolve "document 1" => D1
```

Next, AEResolve asks us to find a paragraph by calling our paragraph-from-document accessor, which returns a token for paragraph 2:

```
resolve "paragraph 2 of document 1" => D1P2
```

Next, AEResolve calls our line-from-paragraph accessor. Because of the **every** keyword, we must return a list of tokens:

```
resolve "every line of paragraph 2 of document 1" =>
   {D1P2L1, D1P2L2, D1P2L3}
```

Next, AEResolve asks for word 2 and calls our word-from-line accessor. In this case, however, our accessor must be able to find a word in each token in a list of line tokens. The accessor's result is a list of word tokens. The list depth doesn't change, because the statement doesn't ask for every word.

```
resolve "word 2 of every line of paragraph 2 of document 1" =>
   {D1P2L1W2, D1P2L2W2, D1P2L3W2}
```

The final resolution asks for every character of each of those three words. Because this is our second **every** in the statement, we know we're going to return a list of lists:

```
resolve "every character of word 2 of every line of paragraph 2 of
document 1" =>
   {{D1P2L1W2C1, D1P2L1W2C2, D1P2L1W2C3, D1P2L1W2C4},
    {D1P2L2W2C1, D1P2L2W2C2, D1P2L2W2C3},
    {D1P2L3W2C1, D1P2L3W2C2, D1P2L3W2C3}}
```

or, as it would be displayed as an AppleScript result:

```
{{"t", "e", "x", "t"}, {"t", "w", "o"}, {"n", "o", "t"}}
```

A list of tokens can also be accumulated by the OSL in the course of handling a **whose** clause. For example, consider the following statement:

```
resolve "every word of paragraph 2 of document 1 that contains "e"" =>
   {"text", "three", "lines", "them", "are", "one"}
```

When this statement is resolved, the OSL will call your object accessors for word 1 through word 16 of the token for paragraph 2 of document 1 and pass each word token to your object-comparison function. Those tokens that match (words that contain the letter *e* in this example) are copied into an AEList with AEPutDesc, and the original is disposed of with AEDisposeDesc. Tokens that don't match are disposed of with your token disposal callback if you've installed one, or with AEDisposeDesc otherwise.

There's a corollary to gotcha #2: *Any token list can be or can contain an empty list or lists.* Given the statement

```
get every character of word 3 of every line of paragraph 2 of document 1
```

we must deal with the fact that line 3 (the last line) of paragraph 2 contains only two words. What then should we do with "word 3 of line 3"? If this were a standalone statement, we'd feel justified in returning an errAEIllegalIndex error to let the user know that the requested word doesn't exist. However, since we're returning lists in the more complex statement, we might want to return an empty list as part of our result instead. For example:

```
{{"b", "l", "o", "c", "k"}, {"o", "f"}, {}}
```

Another example, again from the Text suite, involves words from paragraphs. Suppose paragraph 2 is empty, as in the following block of text:

Hello there!¶
¶
How are you?¶

What will you do with "get every word of every paragraph" in this case? If you decide to support empty lists or empty sublists, all of your handlers will need to be able to deal not only with a single token and arbitrarily deep lists of tokens, but also with an empty list.

How to deal with gotcha #2. Designing your object accessors and your event handlers to be list savvy enables your code to fully respond to script statements that require you to return lists of objects or to apply Apple events to lists of objects.

To handle lists, an object accessor must be able to return a descriptor containing a token that references a single object or a descriptor that contains a list of tokens. For example, a property-from-object accessor must be able to receive a list of object tokens and return a list of property tokens for those objects. For each object you support, you need one of these property-from-object accessors. In Sketch, these basically duplicate the token for the object and then stuff the requested property ID into the token's propertyCode data field.

An object's event handler must also be able to receive a descriptor that contains a single token or a descriptor that contains a list of tokens. It must then apply the event to the object referenced by each token. In addition, the event handler must apply the event to each object in a manner that addresses gotcha #3, discussed later.

If you've installed a token disposal callback function, it too must be able to handle an AEList of tokens.

The Sketch sample handles this gotcha by implementing recursion in both its object accessors and its event handlers. The basic structure of an accessor then consists of three functions. For example, for the QuickDraw Graphics suite, the property-from-object accessor uses these three functions, as shown in Listing 4:

- PropertyFromGraphicObjectAccessor — installed function that calls one of the following two static functions, depending on whether it receives a token or a token list
- PropertyFromListAccessor — always receives a list, and calls itself recursively until it finds a token that doesn't contain a list, when it calls PropertyFromObjectAccessor
- PropertyFromObjectAccessor — always receives a token for a single object, and returns a token representing a property of that object

Listing 4. Functions used by our property-from-object accessor

```
pascal OSErr PropertyFromGraphicObjectAccessor(DescType desiredClass,
    const AEDesc* containerToken, DescType containerClass,
    DescType keyForm, const AEDesc* keyData, AEDesc* resultToken,
    long refcon)
{
    OSErr    error;

    if (containerToken->descriptorType != typeAEList)
        error = PropertyFromObjectAccessor(desiredClass,
            containerToken, containerClass, keyForm, keyData,
            resultToken, refcon);
    else {
        error = AECreateList(NULL, 0L, false, resultToken);
        if (error == noErr)
            error = PropertyFromListAccessor(desiredClass,
                containerToken, containerClass, keyForm, keyData,
                resultToken, refcon);
    }
    return error;
}
```

(continued on next page)

Listing 4. Functions used by our property-from-object accessor *(continued)*

```
static OSErr PropertyFromListAccessor(DescType desiredClass,
   const AEDesc* containerToken, DescType containerClass,
   DescType keyForm, const AEDesc* keyData, AEDesc* resultToken,
   long refcon)
{
   OSErr        error = noErr;
   long         index, numItems;
   DescType     keyword;
   AEDesc       srcItem = {typeNull, NULL};
   AEDesc       dstItem = {typeNull, NULL};

   error = AECountItems((AEDescList*)containerToken, &numItems);
   if (error != noErr) goto CleanUp;

   for (index = 1; index <= numItems; index++) {
      error = AEGetNthDesc(containerToken, index, typeWildCard,
         &keyword, &srcItem);
      if (error != noErr) goto CleanUp;

      if (srcItem.descriptorType != typeAEList) {
         error = PropertyFromObjectAccessor(desiredClass, &srcItem,
            containerClass, keyForm, keyData, &dstItem, refcon);
      }
      else {
         error = AECreateList(NULL, 0L, false, &dstItem);
         if (error == noErr)
            error = PropertyFromListAccessor(desiredClass, &srcItem,
               containerClass, keyForm, keyData, &dstItem, refcon);
      }
      if (error != noErr) goto CleanUp;

      error = AEPutDesc(resultToken, index, &dstItem);
      if (error != noErr) goto CleanUp;

      AEDisposeDesc(&srcItem);
      AEDisposeDesc(&dstItem);
   }

CleanUp:
   AEDisposeDesc(&srcItem);
   AEDisposeDesc(&dstItem);
   return error;
}

static OSErr PropertyFromObjectAccessor(DescType desiredType,
   const AEDesc* containerToken, DescType containerClass,
   DescType keyForm, const AEDesc* keyData, AEDesc* resultToken,
   long refcon)
{
   OSErr      error = noErr;
   DescType   requestedProperty = **(DescType**)(keyData->dataHandle);
```

(continued on next page)

Listing 4. Functions used by our property-from-object accessor *(continued)*

```
    if (CanGetProperty(containerClass, requestedProperty)
     || CanSetProperty(containerClass, requestedProperty)) {
        error = AEDuplicateDesc(containerToken, resultToken);
        if (error == noErr) {
            resultToken->descriptorType = desiredType;
            (**(CoreTokenHandle)(resultToken->dataHandle)).propertyCode
                = requestedProperty;
            (**(CoreTokenHandle)(resultToken->dataHandle)).objectClass
                = containerClass;
        }
    }
    else error = errAEEventNotHandled;
    return error;
}
```

The event handlers use this same three-tiered mechanism to apply events to descriptors that contain either a single token or a list of tokens. For example, the Get Data event will eventually receive the property token returned by the property-from-object accessor above and deal with it as shown in Listing 5.

Again, the event handler passes the first parameter on to GetDataFromGraphicObject, which calls GetDataFromList if the parameter contains a list of tokens, or GetDataFromObject if it contains a token for a single object. Both the object accessor and the event handler use the same three-tiered mechanism to deal with either lists or single tokens. Most of the work is done, in both cases, in the third tier, and if you've already implemented simple object model scriptability, you've already written most of the code for the third tier. To support lists, you just have to add the switching code for the first and second tier, which is almost identical for all object accessors and all event handlers. Using this mechanism, fully supporting lists of any depth is nearly trivial.

Flattening lists. Sometimes, after your object resolution code has built an arbitrarily deep list of lists to satisfy the tail end of a script statement, the final resolution might require you to flatten it back into a single-level list. Sketch includes the FlattenAEList function to perform this duty:

```
OSErr FlattenAEList(AEDescList *deepList, AEDescList *flatList);
```

Here's an example of when you might use it, again from the Text suite:

```
get text of every character of every word of every paragraph ¬
   of every document
```

Since the Text class isn't required to handle either formRange or the **every** construct, you can return a string that spans from the first character in the list to the last character in the list. A function to flatten a typeAEList token from an arbitrary depth to a single list is useful for this purpose, and for use in your Apple event handlers, such as the handlers for Count and Delete. For example, the statement

```
count every character of every word of every line of every paragraph
```

Listing 5. How a Get Data event handles a property token

```
static OSErr HandleGetData(AEDesc *token, const AppleEvent *appleEvent,
   AppleEvent *reply, long refcon)
{
   OSErr    error = noErr;
   AEDesc   data = {typeNull, NULL};
   AEDesc   desiredTypes = {typeNull, NULL};

   AEGetParamDesc(appleEvent, keyAERequestedType, typeAEList,
      &desiredTypes);   // "as" is an optional parameter; don't check
                        // for error.
   error = GetDataFromGraphicObject(token, &desiredTypes, &data);
   if (error == noErr && reply != NULL)
      error = AEPutKeyDesc(reply, keyDirectObject, &data);

   AEDisposeDesc(&data);
   AEDisposeDesc(&desiredTypes);
   return error;
}

OSErr GetDataFromGraphicObject(AEDesc *tokenOrTokenList,
   AEDesc *desiredTypes, AEDesc *data)
{
   OSErr error = noErr;

   if (tokenOrTokenList->descriptorType != typeAEList)
      error = GetDataFromObject(tokenOrTokenList, desiredTypes, data);
   else {
      error = AECreateList(NULL, 0L, false, data);
      if (error == noErr)
         error = GetDataFromList(tokenOrTokenList, desiredTypes, data);
   }
   return error;
}
```

is allowed, and your accessors will return a four-deep list of characters. The Count event handler doesn't care about the structure of the list, only about the number of objects in its sublists, so rather than deal with recursion to step through the list structure you can just flatten the list and then call AECountItems to get the number of elements. This example is somewhat contrived, and although this script fragment would be processed correctly, such processing might be very slow for a large number of objects. This is a side effect of a strict object-first implementation. For some events, such as Count, you may want to write custom counting code that short-circuits your standard object resolution and dispatching mechanism.

GOTCHA #3: THE "PRESERVE A TOKEN'S MEANING" GOTCHA

The meaning of a token's contents must be preserved during the execution of an Apple event that uses that token.

You're most likely to come up against this gremlin when one of your handlers receives a list of tokens and some action needs to be performed on the objects referenced by the tokens in the list. Consider, for example, the following statement:

```
delete character 2 of every word
```

Let's say all of your text tokens are implemented by storing a beginning offset and a length, where the offset is measured from the beginning of a text block. Resolving the above statement will return a list of tokens, with offsets for character 2 of each word in the text block. Next, your handler iterates through the list of objects referenced by the tokens and deletes the character referenced in each object. The first deletion works just fine; you use the offset contained in the first token and delete character 2 of word 1. This causes every following character to move one position to the left to fill the spot vacated by the deleted character. Uh oh! Now the offsets for the remainder of your objects are all off by 1! The next deletion will use the now incorrect offsets, and character 3 of word 2 will be deleted. The next call will delete character 4 of word 3, and so on. This implementation has violated gotcha #3 — you received a single Delete event, but that single event operates on multiple objects, and although your object accessors computed the object tokens correctly at the time they were called, your handler causes the meaning of the tokens to be inaccurate each time it processes another object.

How to deal with gotcha #3. Here are several ways to solve the problem resulting from processing the script statement above:

- You could construct your tokens as offsets from the end of the text block instead of from its beginning. Then, as characters are deleted from the first word to the last, since the end of the list is shrinking also, the offsets will still be correct.
- You could have your handler keep track of the number of characters deleted so far and adjust the offsets in your tokens as you go.
- You could step through the list in reverse order, from the last token down to the first.

These methods all produce the correct results for the script statement above, but they might produce incorrect results for other valid statements. For instance, suppose your user built the word list herself and then reversed the list and sent it to your Delete handler. With the last solution above, you cleverly work from the end of the list to the beginning, but since the user has already reversed the list, you're really back to deleting from the beginning of the text block toward its end, and you experience the very problem you were trying to avoid!

If you're implementing the Text suite, pay particular attention to gotcha #3. Test your implementation with many different scripting constructs, and have people who write scripts very differently from you test it also. If necessary, you may need to first manipulate the order of the tokens in the list you receive to make sure you can preserve the meaning of those tokens until the event has been applied to each one of them.

OTHER GOODIES FOR COMPLETENESS

Now you know how to handle lists and some ways to avoid the big gotchas. But there are still a few more things you can do to make your object model implementation more complete. Specifically, you can implement a "properties" property, implement a property-from-property accessor, provide your own coercions, and return meaningful error codes.

IMPLEMENTING A "PROPERTIES" PROPERTY

You should implement a "properties" property and return a record containing all the properties for an object. This provides a real boon for the scripter, who can then set

or get several properties with a single statement, and it speeds up execution as well since it avoids the need to send many events to get or set properties one at a time.

For instance, if the script says

```
get the properties of rectangle 1
```

the Get Data event should return a record containing the name and value of each property for that object:

```
{bounds: {0, 0, 100, 200}, fill color: red, stroke size: 10, ...}
```

The script could also say something like

```
set properties of rectangle 1 ¬
   to {stroke size: 3, fill color: blue, location: {20, 40}}
```

In Sketch, the Set Data event handler looks at the property token it receives. If the token references a single property, it packages it into a record containing that property and passes the record on to the SetProperties function. If, instead, it receives a record, it just passes that record on to SetProperties. The SetProperties function always receives a record; it examines the record for each property of the object and then applies the value of each property it finds in the record to the object.

IMPLEMENTING A PROPERTY-FROM-PROPERTY ACCESSOR

If you implement the "properties" property, you should also implement a property-from-property accessor. If you don't, you won't be able to get a single property out of the property record you've already built. The first statement below will work, but the second one will generate an error:

```
get fill color of rectangle 1
get fill color of properties of rectangle 1 -- won't work
```

To get around this, the script writer will need to first assign the results to a variable and then depend on AppleScript to extract the property out of that variable:

```
set myProps to properties of rectangle 1
get fill color of myProps
```

But since one of our goals should be to make scripting intuitive and not force the script author into particular programming constructs when not absolutely necessary, both methods of asking for the property should be handled in your code.

Another reason you may need a property-from-property accessor arises when, in the process of defining your object containment hierarchy, you define two classes, make one class an element of the other class, and then realize that the container can contain one and only one instance of that element. For example, imagine a very limited drawing program that allows many graphic objects but only one text block, an instance of the QuickDraw Graphics graphic text class. If you stick with a straight containment metaphor, the script author will need to use a statement like

```
graphic text 1 of document "Graphic Chart"
```

to reference the one and only text block. But why should the scripter have to specify the index of 1 when there can be only one per document? This also invites the scripter

to ask for graphic text 2, for which you would need to return an errAENoSuchObject error.

One way to handle this case is to implement the singleton object as a property of an object rather than a contained class. In your 'aete' resource, define a property (say "label" for the above example) of type graphic text, which is a class defined elsewhere in your 'aete' resource. Now, the script statement

```
get the label of document "Graphic Chart"
```

doesn't need to specify an index, since "label" is a property. What will that statement return? You decide. You might just return the contents of the graphic text as a string. But since the "label" property also references a class, you could return the properties of the text object as a record, such as:

```
{contents: "Financial Results", font: "Times", size: 12, ...}
```

By implementing a property-from-property accessor, you can also properly resolve a statement like this:

```
get font of label of document "Graphic Chart"
```

There's an ongoing debate in the developer community about the best way to design for this single-element situation. Some developers believe that the design discussed above leads to intuitive script statements that make it easier for users to script your application. Others contend that elements and properties serve very different purposes, and that intermixing them in this way both corrupts the object design and confuses the beginning script writer. You'll have to decide for yourself how you want to handle this in your application; there may not be one best design. In any case, if you find yourself in this situation, take the advice Cal Simone gives in his According to Script columns: write down script statements as part of your design and make sure that they seem natural and intuitive before you write your code.

PROVIDING APPLICATION-SPECIFIC COERCIONS

Provide your own coercions. There are several places where these come in handy. First, the Get Data event can take an optional parameter, keyAERequestedType, a list of types that the user would like for the returned data. For instance, a fill color might be represented as one of the following:

- typeEnumerated, such as **red**
- typeChar or typeIntlText, such as "red"
- typeRGBColor, such as {32767, 0, 0}

Thus, a scripter might ask for

```
fill color of rectangle 1 as constant
fill color of rectangle 1 as string
fill color of rectangle 1 as RGB color
```

The Registry defines the type of the **as** parameter as typeAEList, indicating that the first item in the list is the user's preferred data type, the second is the user's next most preferred type, and so on. However, I haven't been able to persuade AppleScript to accept a list for this parameter. It seems as though **get fill color of rectangle 1 as string** (or **RGB color** or **constant**) should work, but it won't compile.

Note that there's a bug in AppleScript 1.1 that generates an error when you implement both lists and the kAERequestedType parameter. The following statement will expose the error:

```
get fill color of every rectangle as string
```

The **every** statement causes you to generate a list of property tokens, which is then passed to your graphic object's Get Data event handler. There, you examine each token, get the fill color from its rectangle, and convert it to a string (presumably the name of the color), as specified in the **as string** part of the statement. Since you received a list of tokens, you return a list of strings, as you should. You've done the right thing, but AppleScript isn't satisfied! It doesn't realize that you've already handled the **as string** coercion, so it tries to coerce the list of strings you returned into a string and it reports a coercion error. There's really nothing you can do in your application to work around this bug; you'll have to wait for it to be fixed in a future version of AppleScript. There *is* a way that scripts can handle the error, however:

```
tell document 1 of application "Sketch"
   try
      set colorNames to fill color of every rectangle as string
   on error number -1700 from offendingVariable
      set colorNames to offendingVariable
   end try
end tell
```

RETURNING USEFUL ERROR CODES

One last suggestion: Return a meaningful error code and error message if you don't or can't handle an event, an object, or a data type. Table 5 presents a list of some of the most common return codes, when to use them, and the error message that AppleScript generates when one of these errors occurs.

Table 5. Common error codes and examples of when you might return them

Error to return	Example of when to return it	Error message
errAEEventNotHandled (–1708)	This isn't just a catch-all error; it has specific side effects in certain situations. If your code doesn't handle an event, this is a signal for the Apple Event Manager to give any system handlers a shot at the event.	<object-reference> doesn't understand the <event> message.
errAECoercionFail (–1700)	When you can't coerce some data to the requested type.	Can't make some data into the expected type.
errAENoSuchObject (–1728)	When the requested object doesn't exist.	Can't get <object reference>.
errAENotASingleObject (–10014)	When a handler that doesn't handle lists receives a list.	Handler only handles single objects.
errAENotAnElement (–10008)	When you get a request to delete a property.	The specified object is a property, not an element.
errAENotModifiable (–10003)	When the object can *never* be modified, such as a read-only property. See also errAEWriteDenied.	Can't set <property> to <value>. Access not allowed.
errAEWriteDenied (–10006)	When the object can't *currently* be modified, such as a locked rectangle that can't be changed until it's unlocked. See also errAENotModifiable.	Can't set <property> to <value>.

(continued on next page)

Table 5. Common error codes and examples of when you might return them *(continued)*

Error to return	Example of when to return it	Error message
errAECantHandleClass (–10010)	When an event handler can't handle objects of this class, or when an object-counting callback receives an object type it can't count.	Handler can't handle objects of this type.
errAEIllegalIndex (–1719)	When the scripter asks for an index greater than the number of objects or less than 1. Remember that negative indexes are legal, so convert them to a positive index before performing a range check.	Can't get <object-reference>. Invalid index.
errAEImpossibleRange (–1720)	When you process formRange and you can't return a list of objects between the boundary objects — for example, when two rectangles are specified and are in different documents.	Invalid range.
errAEWrongDataType (–1703)	When a descriptor contains an unexpected data type, or when an object-comparison callback doesn't know how to compare one of the data types.	<value> is the wrong data type.
errAETypeError (–10001)	When you receive a Set Data event, and the descriptor isn't of the expected type and can't be coerced into that type.	<value> is the wrong type.
errAEBadKeyForm (–10002)	When an object is requested by a key form that your accessor doesn't support — for example, rectangle by name.	Invalid key form
errAECantSupplyType (–10009)	When you can't return the type of data specified in the **as** parameter of a Get Data event.	Can't supply the requested type for the data.

Some error codes have a very generic error message as a default, but you can supply additional parameters in the reply event so that the error message will be more specific. For example, an errAECoercionFail message usually says, "Can't make some data into the expected type," but if you add kOSAErrorOffendingObject and kOSAErrorExpectedType parameters to the reply event, you'll get a much more informative message, such as "Can't make fill color of rectangle 1 into a string." These parameters can also be added to errAEWrongDataType and errAETypeError replies. For more detail on giving better error messages using this technique, see Developer Notes in the AppleScript Software Development Toolkit. You may want to define additional error codes for your application, and if so you should be sure to also set the error text in the reply event. Take a look at the PutReplyErrorNumber and PutReplyErrorMessage functions in the Sketch source code to see how to do this.

EXERCISING YOUR IMPLEMENTATION

This article has described some things you can do to implement an Apple event object model in your application so that it can handle complex scripts. Take a close look at the code for the Sketch application to see how it uses the object-first method to handle events and scriptable objects. Carefully examine the dictionaries of several applications that are fully scriptable, such as QuarkXPress, the Scriptable Text Editor, or PhotoFlash. Pay attention to how their 'aete' resources are constructed, and read the *develop* columns by Cal Simone ("According to Script") to gain further insight into how to organize both your 'aete' resource and your object model.

Then give your implementation a thorough workout to see if you can spot any problems. Write AppleScript test cases to exercise the most complex AppleScript scripts that you want to support. Use the key forms that return lists, and mix them unmercifully in your test scripts. Exercise every gotcha. If your application stands up to the test, shout "Ship it!"

RECOMMENDED READING

- "Apple Event Objects and You" by Richard Clark, *develop* Issue 10.
- "Speeding Up **whose** Clause Resolution in Your Scriptable Application" by Greg Anderson, *develop* Issue 24.
- "Designing a Scripting Implementation" by Cal Simone, *develop* Issue 21. Also, look for Cal's According to Script columns starting with *develop* Issue 22.
- *Inside Macintosh: Interapplication Communication* by Apple Computer, Inc. (Addison-Wesley, 1993).
- *Apple Event Registry: Standard Suites* (Apple Computer, Inc., 1992).
- *AppleScript Language Guide* by Apple Computer, Inc. (Addison-Wesley, 1993). This book is included in the AppleScript Software Development Toolkit.

Thanks to our technical reviewers Greg Anderson, Andy Bachorski, Greg Friedman, C. K. Haun, and Jon Pugh for reviewing this article.

THE OPENDOC ROAD

OpenDoc Memory Management and the Toolbox

TROY GAUL AND
VINCENT LO

OpenDoc has its own memory management system, but OpenDoc part editors also need to interact with the Macintosh Toolbox, which often does memory management using its own system. This column will point out potential pitfalls resulting from the interaction of these two systems, and suggest strategies to avoid them.

OpenDoc has adopted the well-tested Memory Manager from the MacApp and Bedrock frameworks for its own use. This memory manager was designed to provide fast and efficient memory allocation to the framework, and since OpenDoc's memory requirements are similar to those of a framework, it's natural for OpenDoc to reuse the code that has served the framework so well. The OpenDoc Memory Manager (as we'll call it here) is installed with OpenDoc as a shared library and handles most of the memory allocation and deallocation in an OpenDoc process.

There are several reasons for OpenDoc to have its own memory manager:

- The OpenDoc Memory Manager can improve upon the memory manager of the underlying platform. On Macintosh 7.x systems, for instance, a faster and more space-efficient memory allocation algorithm replaces the one provided by the Macintosh Toolbox. In general, having a separate memory manager for OpenDoc allows the platform implementer to fine-tune OpenDoc performance.
- The OpenDoc Memory Manager provides a cross-platform API for both platform implementers and part developers, covering both nonrelocatable and relocatable blocks. For platforms with no built-in relocatable blocks, platform implementers could provide relocatable blocks through the OpenDoc Memory Manager API without changing the underlying operating system.
- The OpenDoc Memory Manager is packaged as a CFM shared library, so it can easily be replaced if a new version is required for improved performance or feature enhancements. Also, as Apple evolves the Mac OS and its memory manager, OpenDoc will adopt this new technology. Developers using the OpenDoc Memory Manager API will reap the benefits of the new system memory manager without modifying or even recompiling their part editors.

The OpenDoc Memory Manager defines a low-level procedural API, which is described in the "Memory Management" section of Appendix A in the *OpenDoc Cookbook*. An OpenDoc utility library named ODMemory (provided as source code) organizes the low-level API into high-level functions. The major difference between the underlying API and the "wrapper" ODMemory API is that the latter signals an exception when an error condition is encountered. For more on exception handling in OpenDoc, see "OpenDoc Exception Handling."

To ensure optimum usage of memory, part developers should use the OpenDoc Memory Manager API or the ODMemory utility library to satisfy their memory needs. The only exception is when the part editor is interacting with a Toolbox manager that requires memory to be allocated in a certain memory heap. We'll go into more detail about this in a moment.

HOW THE OPENDOC MEMORY MANAGER WORKS

The OpenDoc Memory Manager allocates fixed-sized, nonrelocatable blocks of memory and organizes them into heaps. The memory allocated for these heaps may come from the application heap, temporary memory, or the system heap. These memory blocks (called *allocation segments*) are subdivided into smaller memory blocks as memory requests are made by OpenDoc objects and part editors. When MMAllocate or ODNewPtr is called to allocate a block of memory, the OpenDoc Memory Manager returns a pointer to one of these memory blocks in an allocation segment.

When an OpenDoc process is started up, the OpenDoc Memory Manager allocates a small amount of memory for a heap, which becomes the default heap. Clients of

TROY GAUL (tgaul@apple.com) has been writing OpenDoc parts — er, Live Objects — since starting at Apple last May. Having created the Infinity Windoid WDEF in 1991, he has since appeared in more About boxes than Elvis.•

VINCENT LO is Apple's technical lead for OpenDoc. When he's not dealing with Live Objects, he's frequently spotted at fine dining establishments in the San Francisco Bay Area. One of his dreams is to open up a Chinese restaurant in Italy.•

OPENDOC EXCEPTION HANDLING

Handling exceptions in OpenDoc is a large topic, which will only be touched on here. Readers should refer to the "Exception Handling" section of Appendix A in the *OpenDoc Cookbook* for more details.

In a nutshell, OpenDoc allows developers to choose their own exception mechanisms while providing a convenient utility to enable these exception mechanisms to work with SOMobjects™ for Mac OS (the Apple implementation of the IBM SOM™ technology), which underlies OpenDoc.

SOM propagates exceptions through the environment parameter (commonly known as the ev parameter). It's illegal to throw an exception out of a SOM method. Instead, the exception code should be stuffed into the ev parameter and returned to the caller. The caller should examine the ev parameter to see whether an error has been signaled in the called function. Since this checking needs to be done after *every* SOM method invocation, OpenDoc provides a utility to automatically check the ev parameter. If an error has been signaled, the utility will use the chosen exception mechanism (through the use of macros) to propagate the exception.

The macros for the SOM exception handlers are prefixed with "SOM_": SOM_TRY, SOM_CATCH_ALL, and SOM_ENDTRY. These macros should not be confused with TRY, CATCH_ALL, and ENDTRY. The non-SOM exception handler macros do not propagate the exception automatically unless RERAISE is called explicitly in the catch block, and they can't be used to propagate an exception across a SOM boundary.

the OpenDoc Memory Manager can create extra heaps and make any of these the default heap.

If a new block is requested and no allocation segments have enough free space to satisfy the request, the OpenDoc Memory Manager will trigger the creation of another allocation segment, which has the effect of growing the heap. Similarly, when all blocks in an allocation segment are freed, the segment is freed as well, shrinking the heap.

The OpenDoc Memory Manager and ODMemory utility also provide a way to allocate relocatable blocks. These blocks are not suballocated from the allocation segments; instead, the OpenDoc Memory Manager allocates them directly from the same operating system heap zone that the OpenDoc heap allocates segments from.

MEMORY PARTITION

Typically, several OpenDoc part editors run in the single process associated with a document. There's no way to determine how many part editors are going to be used in a document and how much memory each part editor requires. Therefore, it's impossible to know how big the memory partition of the process should be before opening the OpenDoc document. As described above, the OpenDoc Memory Manager has its own allocation scheme, which is not limited by the application partition, so the memory partition becomes less significant. (Currently the default heap is allocated from temporary memory, but this may change in the future.) The elimination of the need for the end user to understand the concept of application heap and adjust the memory partition is one of the design goals of OpenDoc.

However, users familiar with OpenDoc might remember that the Document Info dialog allows them to change the partition size of the process. You might ask, "If the OpenDoc Memory Manager does what it claims to, why do I need to adjust the memory partition?"

Even though most of the memory allocation is done through the OpenDoc Memory Manager, Toolbox managers do allocate memory in the application heap, and the amount required varies considerably depending on the size of the data manipulated and the operations performed. Changing the memory partition is needed to accommodate these cases.

When a document is created, it's opened into a process of a default size. Users can change the default size for the document by using the Document Info dialog. There's also a desktop utility called Infinity OpenDoc Sizer that's capable of changing either the partition size of a particular document (without first having to open it) or the default partition size used by all documents that don't already have custom partitions. It's available in the Developer Release area of the OpenDoc Web site (http://www.opendoc.apple.com) and accompanies this column on this issue's CD and *develop*'s Web site.

MANAGING TOOLBOX MEMORY ALLOCATIONS

As your code makes calls into the Macintosh Toolbox, you'll find several places where the Toolbox allocates memory for you. Generally, this memory is allocated out of the current heap, which is usually the application

heap. Since the size of an OpenDoc document's application heap is limited (the default heap size leaves only about 100K of space free after OpenDoc itself is loaded), you need to be careful when calling Toolbox routines that allocate in that heap. With some care, you can control your allocations so that your users won't have to increase your document's heap size in order to use your part.

When you're dealing with resources in particular, there are a few techniques you can use to handle memory allocations. Standard resource access routines such as GetResource will generally cause the associated memory to be allocated in the application heap. If you're using a resource for only a short period of time and it's fairly small, you can continue to use these routines to access it and load it into the application heap.

For larger resources, however, this won't work. Luckily, OpenDoc provides the utility library UseRsrcM to help, as described in the "Resource Handling" section of Appendix A in the *OpenDoc Cookbook*. The utility routine ODReadResource allows you to load resources from your part's shared library file into temporary memory. This works by determining the size of the resource, allocating a relocatable block of this size in temporary memory, and using ReadPartialResource to load the resource directly into that block. (Note, however, that resources read in by ODReadResource are detached; you cannot, for instance, call ChangedResource to write out modifications to them. Also, each call to ODReadResource will return a new copy of the resource.) If you need to access large resources from files other than your part — for instance, for a sound-editing part that needs to load an 'snd ' resource from another file — you can use this same technique yourself. A part editor must also ensure that its resource file is in the resource chain before accessing its resources (see "Resource File Access").

There are times when the Toolbox loads resources for you. In these cases, you generally can't get them to be loaded into temporary memory. In several cases, the resources are small or are allocated for only a short time, so there isn't much to worry about. For instance, when accessing resources such as string lists ('STR#'), cursors, and icon families, you can continue to use the normal Toolbox routines. In these cases, the resource is often accessed once and left around in memory. Therefore, you'll want to make sure these resources are marked as purgeable so that they can be deallocated automatically when more space is needed.

There are other times when larger resources (such as pictures) are being accessed and problems can crop up where you might not expect them. For instance, if you

RESOURCE FILE ACCESS

In an OpenDoc environment, parts must share access to system services that applications normally own exclusively. This adds some complications to the programming model you're probably used to.

One such shared service is the resource chain. Since there are potentially many parts all working in the active document, the resource chain must be shared between them. Also, because OpenDoc parts are shared libraries, the resources in your part's file aren't automatically available like those in an application.

The utility library UseRsrcM facilitates making your resource file available and accessing resources from it. To open and initialize access to your shared library's resource file, you call InitLibraryResources from your CFM initialization routine. You also call CloseLibraryResources from your CFM termination routine to close the resource file when you're done with it.

To access a resource from your part's shared library, you must first call BeginUsingLibraryResources. This adds the part's resource file to the resource chain and sets the top of the chain to that file (so that calls such as Get1Resource will retrieve resources from the correct file). After reading or writing the necessary resource, you call EndUsingLibraryResources to remove the file from the resource chain. For C++ users, a stack-based class named CUsingLibraryResources is provided to do this for you automatically.

There are a couple of implications about using this mechanism for handling the resource chain. Because Resource Manager routines are available only inside a Begin…/ End… block, you must make sure your code and the Toolbox aren't trying to manipulate resources at other times. You also have to be careful that LoadResource isn't called on a purged resource from a file that's not in the chain.

Other things, such as icons or Balloon Help in menus (which are loaded while the menu is pulled down and the part isn't in control), can also cause problems. If you understand the relationship between your resource file and the resource chain, however, you can work around these potential pitfalls.

have a large picture that's referenced by a dialog item (like that 24-bit rendered image for the background of your About box) and there isn't sufficient memory available when the dialog is displayed, the picture won't be shown. One solution to this is to create your own heap zone, as discussed later. Another is to create a user item procedure for the picture, handling the memory allocation and spooling in of the picture yourself.

For resources such as menus and definition procedures (WDEFs, CDEFs, and MDEFs), there is little you can safely do. Most of these types of resources, although they persist for a long time, are fairly small, so having them allocated in the application heap isn't terrible.

Another commonly used piece of memory allocated by the Toolbox, but in this case not resource-based, is a region. If you're doing many region operations and the regions aren't being kept around, you can continue to use NewRgn to allocate them in the application heap. However, if you're keeping regions around for long periods of time, there's an ODMemory utility routine, ODNewRgn, that you can call to allocate your regions from the OpenDoc heap.

OTHER TECHNIQUES TO KEEP ALLOCATIONS OUT OF THE APPLICATION HEAP

In some cases, the Toolbox allows you to specify that a memory allocation come from temporary memory rather than the application heap. Probably the most common example of this is a GWorld. Passing the useTempMem flag to NewGWorld or UpdateGWorld will cause it to allocate the PixMap's buffer in temporary memory rather than in the application heap. Use these opportunities when they present themselves.

Also, if you're allowed to specify the location of memory used by the Toolbox, such as for the WindowRecord in calls to NewWindow and the sound channel in SndNewChannel, you should take these opportunities to allocate the memory with ODNewPtr rather than allowing the Toolbox to do the allocation.

Another technique that you might consider for dealing with large or long-term resources is loading them into the system heap. This can be done by setting the "system heap" flag in the resources' attributes. This has the advantage of being easy to do and working even for resources allocated by the Toolbox. The disadvantage is that increasing the system heap's size can be bad for performance if virtual memory is enabled. Since the system heap is always paged into real memory space in System 7, a large system heap means there isn't much space in RAM available for paging in the rest of memory.

Yet another technique that can sometimes be useful is to create your own heap zone. To do this, create a block in temporary memory using ODNewHandle, lock it, and create a heap inside it by calling InitZone. This technique shouldn't be used for just any allocation, but only if other methods don't work and if the allocation is ephemeral. One example we mentioned before where this might be useful is for a dialog box with large picture items. Note that you'll have to make sure the heap is large enough to hold not only the pictures but also the other dialog-related resources, and you'll want to leave some room to spare. Also, you probably wouldn't want to have more than one of these heaps allocated at a time. When locking *any* handle in temporary memory, make sure you unlock it again as soon as possible.

Other parts of the Mac OS, such as QuickTime, will require you to use these and other techniques to deal effectively with memory. (In the case of QuickTime, you can use SetZone to switch to the system heap.) In such cases, it's a good idea to use a tool such as Metrowerks' ZoneRanger to examine memory allocated in the application heap and, if problems are found, look for ways to move allocations elsewhere.

MEMORY MATTERS

In the future, a new system memory manager will allow for heaps that can grow. When this becomes available, many of these contortions will become unnecessary. In the meantime, however, it's important to remember that you're sharing the application heap with other clients and to act accordingly.

RELATED READING

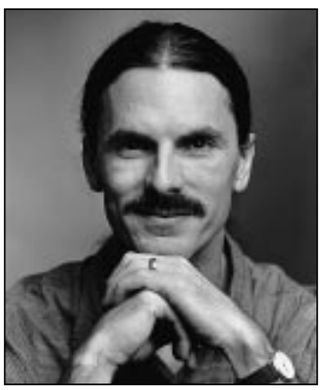
For more information on the memory manager in OpenDoc and the utility libraries mentioned in this column, check out the following references:

- *OpenDoc Programmer's Guide for the Mac OS* by Apple Computer, Inc. (Addison-Wesley, 1995).
- *OpenDoc Cookbook for the Mac OS* by Apple Computer, Inc. (Addison-Wesley, 1995). In particular, see Appendix A, "OpenDoc Utilities."
- The OpenDoc World Wide Web site, located at http:/ / www.opendoc.apple.com.

Thanks to Jens Alfke, Dave Bice, and Steve Smith for reviewing this column.

New QuickDraw 3D Geometries

A number of new QuickDraw 3D geometric primitives can save you time as you create 3D objects — from footballs to the onion domes of the Taj Mahal. Most of these new primitives are very versatile, but this versatility comes at the cost of some complexity. Here you'll find a discussion of their various features and uses, with special attention to the differences among the features and structural characteristics of the polyhedral primitives. Being aware of these differences will help you make the right choices when you're using these primitives in a particular application.



PHILIP J. SCHNEIDER

When QuickDraw 3D version 1.0 made its debut, it came with 12 geometric primitives that you could use to model pretty much anything you wanted. With applied cleverness, you could make arbitrary shapes by combining and manipulating such primitives as polylines, polygons, parametric curves and surfaces, and polyhedra. Because some shapes are so commonly used, recent versions of QuickDraw 3D have added them as high-level primitives, including two new polyhedral primitives. This frees each developer from having to reinvent them and ensures that the new primitives are implemented in such a way as to fit nicely with the existing paradigm in QuickDraw 3D.

We'll start by looking at how the new ellipse primitive was designed. A similar paradigm was used in creating most of the other new high-level primitives. Understanding their design will help you use them effectively. Later, we'll move on to the two new polyhedral primitives — the polyhedron and the trimesh — which you can use to model very complex objects. We'll also take a fresh look at the mesh and trigrid, which have been around for a while, and compare the usefulness of all four polyhedral primitives. Along the way, you'll find some relevant background information about the QuickDraw 3D team's design philosophy.

I'm going to assume that you're already familiar with the capabilities of QuickDraw 3D, including how to use the original 12 geometric primitives. But if you want more

PHILIP J. SCHNEIDER (pjs@apple.com) is still the longest-surviving member of the QuickDraw 3D team (and in answer to an oft-posed question, no, not as in "surviving member of the Donner Party"). One current task is to find a name for his second son, which he and his wife expect in January, that won't eventually lead to a question like "Why did you give my older brother a cool name like Dakota, and then name me Bob?" He's given up trying to teach two-year-old Dakota to change his own diapers and has instead begun teaching him Monty Python's "The Lumberjack Song," which isn't nearly as useful a skill, but is one at which he has a better chance of succeeding. Philip's original interest in geometry began early, when an elementary school teacher warned him that he could "put an eye out" with a protractor.

basic information, see the articles "QuickDraw 3D: A New Dimension for Macintosh Graphics" in *develop* Issue 22 and "The Basics of QuickDraw 3D Geometries" in Issue 23. The book *3D Graphics Programming With QuickDraw 3D* has complete documentation for the QuickDraw 3D programming interfaces for version 1.0. Version 1.5 of QuickDraw 3D, which supports these new primitives, is now available.

To get you started using the new primitives, the code listings shown here accompany this article on this issue's CD and *develop*'s Web site.

Aficionados of QuickDraw 3D use a variety of terms to refer to a geometric primitive. But a geometric primitive by any other name (primitive geometric shape, basic geometric object, geometric primitive object, or geometry) is still a geometric primitive.•

CONICS, QUADRICS, AND QUARTICS

One category of geometric primitives is conics, quadrics, and quartics; this class includes such shapes as ellipses, disks, ellipsoids (the generalization of spheres), cones, cylinders, and tori (doughnuts). Each of these shapes is a recently introduced primitive that's defined with a paradigm similar to the one already used in the box primitive. I'll begin by explaining how the ellipse primitive works because the same basic approach is used for the more complex geometries.

ELLIPSES

My article "NURB Curves: A Guide for the Uninitiated" in *develop* Issue 25 describes how you can make circles and partial circles with NURB curves. Though you can further use NURB curves to make ellipses and elliptical arcs by manipulating the locations of the control points, this isn't necessarily the most convenient way to do it. So QuickDraw 3D now provides an ellipse primitive.

The data structure for the ellipse is as follows:

```
typedef struct TQ3EllipseData {
    TQ3Point3D          origin;
    TQ3Vector3D         majorRadius;
    TQ3Vector3D         minorRadius;
    float               uMin, uMax;
    TQ3AttributeSet     ellipseAttributeSet;
} TQ3EllipseData;
```

Let's assume we have a variable declared like this:

```
TQ3EllipseData      ellipseData;
```

As we go over the ellipse primitive, I'll explain the various fields in the data structure, then fill them in as I tell you how they work. Let's take for our example a special case of an ellipse — a circle of radius 2 that lies in the {*x*, *y*} plane, with an origin at {3, 2, 0} — and show how we'd define it in QuickDraw 3D.

For starters, a circle must have a center. One way QuickDraw 3D could do this is always center the circle at the point {0, 0, 0} and then have us translate the circle to the desired location. However, it seems a bit odd to be able to make, say, a line with arbitrary endpoints, but not be able to make a circle with an arbitrary center. So, as shown in Figure 1, a QuickDraw 3D circle follows the paradigm for primitives and has an explicit center, called the origin in the data structure:

```
Q3Point3D_Set(&ellipseData.origin, 3, 2, 0);
```

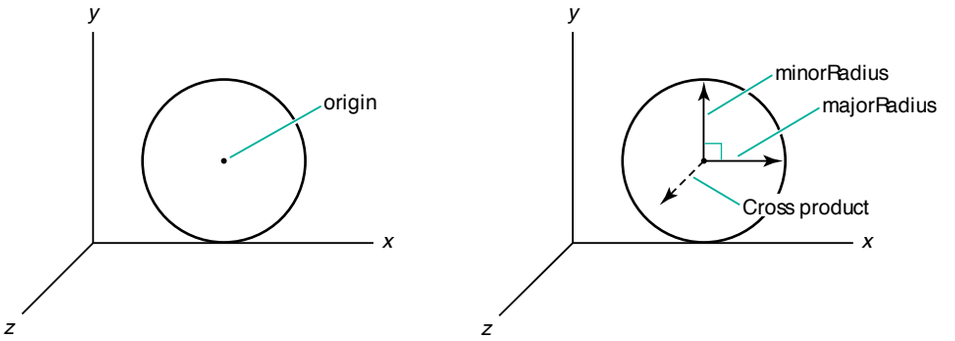


Figure 1. Defining a circle's origin, size, and plane

Of course, circles must have a size. Again, QuickDraw 3D could make all circles a unit size (that is, have a radius of 1) and then require us to scale them appropriately. But, for the same reason that the circle has an explicit center, it has an explicit size.

Given an origin and size, we have to specify the plane in which the circle lies in 3D space. Though it would be possible for QuickDraw 3D to define a circle's plane by default — say, the {*x*, *z*} plane — and require us to rotate the circle into the desired plane, QuickDraw 3D lets us define the radius with a vector whose length is the radius. Then we similarly define a second radius perpendicular to the first radius. The cross product of these two vectors (majorRadius and minorRadius) defines the plane the ellipse lies in:

```
Q3Vector3D_Set(&ellipseData.majorRadius, 2, 0, 0);
Q3Vector3D_Set(&ellipseData.minorRadius, 0, 2, 0);
```

In other words, the plane the circle lies in passes through the origin of the circle, and the cross product of the majorRadius and minorRadius vectors is perpendicular to the plane (see Figure 1).

For a full circle, we need to set uMin to 0 and uMax to 1 (more on this later):

```
ellipseData.uMin = 0;
ellipseData.uMax = 1;
```

As for the final field in the data structure, ellipseAttributeSet, QuickDraw 3D includes this field so that we can, for instance, make screaming yellow ellipses:

```
ellipseData.ellipseAttributeSet = Q3AttributeSet_New();
Q3ColorRGB_Set(&color, 1, 1, 0);
Q3AttributeSet_Add(ellipseData.ellipseAttributeSet,
    kQ3AttributeTypeDiffuseColor, &color);
```

Finally, we create an ellipse object that describes the circle:

```
ellipse = Q3Ellipse_New(&ellipseData);
```

Or we can use the data structure in a Submit call in immediate mode (for rendering, bounding, picking, or writing):

```
Q3Ellipse_Submit(&ellipseData, view);
```

The ellipse comes with the usual array of calls for getting and setting individual definitional properties, as well as the entire definition, just like the other primitives.

Why all of this power just for a circle? This power gives us flexibility. Let's use some of the object-editing calls to make some more interesting shapes. We'll start by making an ellipse out of the circle we've just constructed. If you recall, we originally made the circle with majorRadius and minorRadius equal to 2. So to make an ellipse instead of a mere circle, all we have to do is make majorRadius and minorRadius different lengths. To get the first ellipse you see in Figure 2, we can use this:

```
Q3Vector3D_Set(&vector, 0, 1, 0);
Q3Ellipse_SetMinorRadius(ellipse, &vector);
```

"But wait," you say, "vectors have direction as well as size!" Well, we can get really carried away and make the two defining vectors nonperpendicular to get something like the second ellipse shown in Figure 2:

```
Q3Vector3D_Set(&vector, 1, 2, 0);
Q3Ellipse_SetMinorRadius(ellipse, &vector);
```

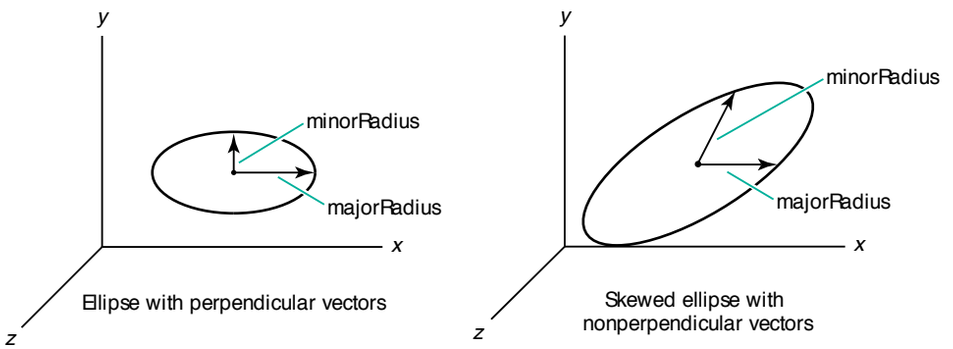


Figure 2. Defining a regular and skewed ellipse

All we have left is how to define partial ellipses. We can do this by taking a parametric approach. Let's say that an ellipse starts at $u = 0$ and goes to $u = 1$. Then we have to define the starting point. Let's make it be the same point that's at the end of the vector defining majorRadius in the first circle in Figure 3. To make a partial ellipse (that is, an elliptical or circular arc), we specify the parametric values of the starting and ending points for the arc:

```
Q3Ellipse_SetParameterLimits(ellipse, 0.05, 0.3);
```

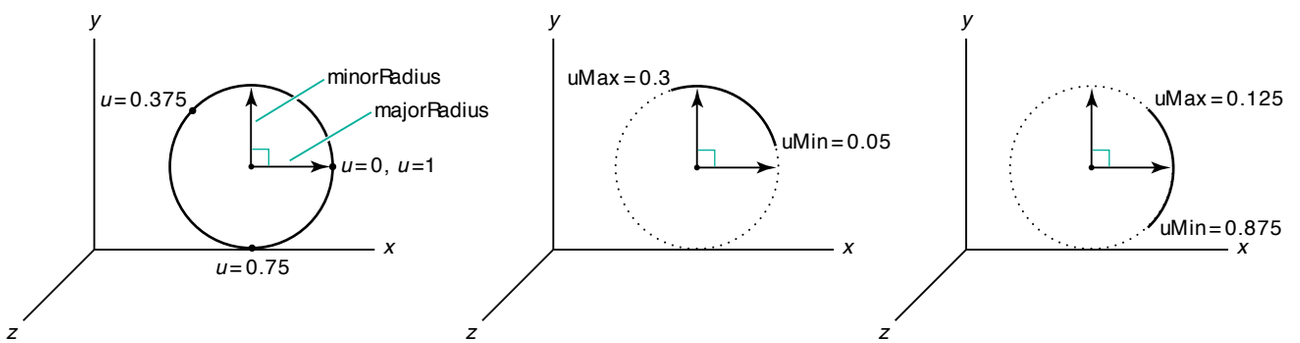


Figure 3. Defining a partial ellipse

This gives us an elliptical (or in this case, circular) arc, as shown in Figure 3. (The dotted line isn't actually rendered — it's just there for diagrammatic reasons.) Though the starting and ending points must be between 0 and 1, inclusive, we can make the starting point have a greater value than the ending point:

```
Q3Ellipse_SetParameterLimits(ellipse, 0.875, 0.125);
```

As you can also see in Figure 3, this allows us to "wrap around" the point $u = 0$.

> **In version 1.5 of QuickDraw 3D,** the feature for defining partials is not enabled, so until it is, you must set the minimum and maximum parameter limits to 0 and 1, respectively.•

About now, you're probably thinking that all this power and flexibility for just a simple ellipse is overkill and that the preceding explanation is overkill, too. Sorry about that, but there is a reason — it turns out that we can take this same approach to defining disks, ellipsoids, cones, cylinders, and tori.

DISKS

If you go back over the past few pages and substitute *disk* for *ellipse*, you pretty much get everything you need to know. The data structure and functionality are analogous, except that disks are filled primitives, like polygons, while ellipses are curvilinear primitives, like polylines. So, partial disks are like pie slices rather than arcs. The only other difference is that since disks are surfaces rather than curves, they have parameters in two directions. Figure 4 illustrates the definition of a disk, including the U and V parameters.

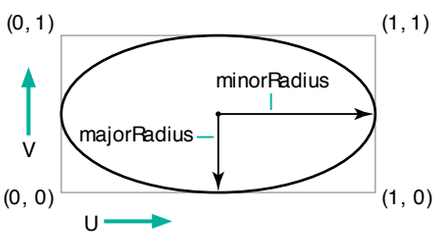


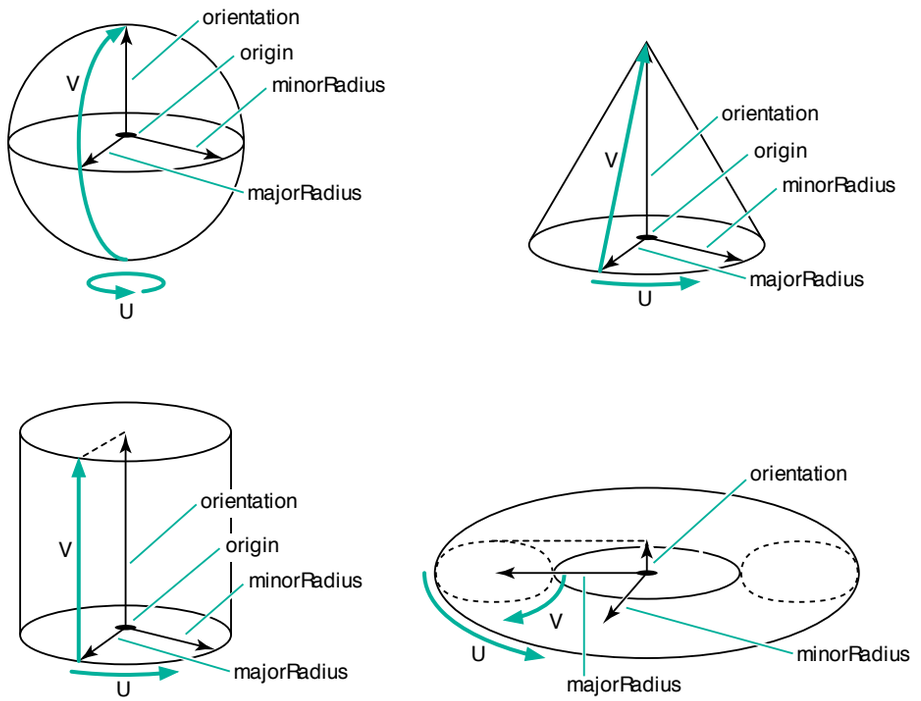
Figure 4. Defining a disk

Note that the UV surface parameterization for the disk is different from the parametric limit values around the perimeter of the disk. The UV surface parameterization was chosen so that an image applied as a texture would appear on the disk or end cap as if it were applied as a decal. The values associated with positions around the perimeter are used for making partial disks, just as we used them to make partial ellipses. The distinct parametric limit values (uMin and uMax) are necessary so that the partial end caps on partial cones and cylinders will properly match. If the surface parameterization for the disk meant that the U direction went around the perimeter, you'd have a nearly impossible time applying decal-like textures.

ELLIPSOIDS, CONES, CYLINDERS, AND TORI

Now, I want you to hold two thoughts in your head at the same time: recall that the box primitive is defined by an origin and three vectors, which define the lengths and orientations of the edges of the box, and then think about the definition of the ellipse. Doing that, you should be able to imagine how we define, say, a sphere — we just add another vector to the definition of the ellipse!

Figure 5 shows how an ellipsoid (a sphere), cone, cylinder, and torus are defined with respect to an origin and three vectors (the labels being fields in the corresponding data structures). Note that the torus requires one more piece of information to allow for elliptical cross sections: the ratio between the length of the orientation vector (which gives the radius of the "tube" of the torus in the orientation direction) and the



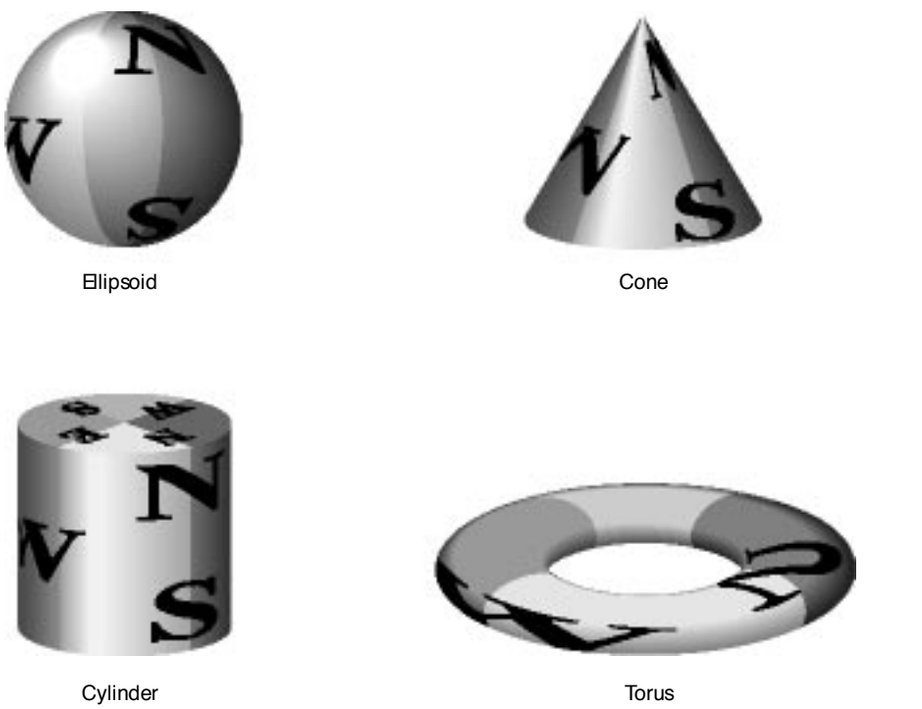


Figure 5. Creating four primitive objects and applying texture

radius of the tube of the torus in the majorRadius direction. With the resulting torus primitive, you can make a circular torus with an elliptical cross section, or an elliptical torus with a circular cross section, or an elliptical torus with an elliptical cross section. (Hmm…perhaps I was drinking too much coffee when I designed the torus.)

You use the U and V parameters to map a texture onto a shape. In Figure 5, the U and V parameters have their origins and orientations relative to the surface in what should be the most intuitive layout. If you apply a texture to the object, the image appears as most people would expect.

- For the ellipsoid, the parametric origin is at the south pole, with V going upward toward the north pole, and U going around the axis in a counterclockwise direction (when viewed from the north pole).
- For the cone and cylinder, the parametric origin is on the bottom edge, at the point where the majorRadius vector ends. V goes up while U goes around in the direction shown by the arrows. The bottom of the cone, and the top and bottom of the cylinder, are parameterized exactly like the disk.
- For the torus, the parametric origin is located at the point on the inner edge where the majorRadius goes through it. V goes around as shown by the arrow, and U goes around the "outside" of the entire torus.

By changing the relative lengths of the majorRadius, minorRadius, and orientation vectors, you can get ellipsoids, cones, cylinders, and tori with elliptical cross sections, similar to how we made a circle into an ellipse earlier.

So to make an ellipsoid that's a sphere, you make the majorRadius, minorRadius, and orientation vectors the same length as well as mutually perpendicular. To make an elliptical cylinder, you can vary the lengths of the three vectors. Even more fun can be had by making the vectors nonperpendicular — this makes skewed or sheared objects. This is easy to see with a cylinder (Figure 6).

Figure 6. Creating an elliptical or sheared cylinder

You can make partial disks, cones, cylinders, and tori in a fashion analogous to what we did with the ellipse (see Figure 7). Since these are surfaces, you can set a minimum and maximum for each direction.

One important thing to notice is that the "wraparound" effect I showed with the ellipse, by making uMin be greater than uMax, is possible with all the other primitives in this category, but the equivalent feature in the V direction is possible only with the torus. For example, the cone wraps around naturally in the U direction because the face itself is one continuous surface in that direction, but the surface doesn't wrap in the V direction.

Figure 7. A partial cylinder and cone

Some of you must be wondering what we can do with the ends of cones and cylinders. Do we want them left open so that the cones look like dunce caps and the cylinders look like tubes? Or do we want them to be closed so that they appear as if they were true solid objects? You may have already wondered about a similar issue when we used the uMax and uMin parameter values to cut away part of the object. Do we make a sphere look like a hollow ball, or like a solid ball that's been cut into?

To take care of these issues, the ellipsoid, cone, cylinder, and torus have an extra field in their data structures that you can use to tell the system which of these end caps to draw:

```
typedef enum TQ3EndCapMasks {
    kQ3EndCapNone           = 0,
    kQ3EndCapMaskTop        = 1 << 0,
    kQ3EndCapMaskBottom     = 1 << 1,
    kQ3EndCapMaskInterior   = 1 << 2
} TQ3EndCapMasks;

typedef unsigned long TQ3EndCap;
```

The *end* cap is a bit of a misnomer, as it refers to the end caps of the cone and cylinder as well as to the "interior end caps" that are the analog to the base and top caps of the cylinder for the portion at the boundary of the cutaway.•

What about attributes? As for all other geometries in QuickDraw 3D, there is an attribute set for the entire geometry, so you can make the entire cone, say, all one color, or apply a texture to the entire object. But you're probably wondering about all these end caps. For example, you might want to have different textures for a solid cylinder's top end cap, face, and interior end caps, as shown in Figure 8. The data structures for the ellipsoid, cone, cylinder, and torus have fields for storing these types of attributes.

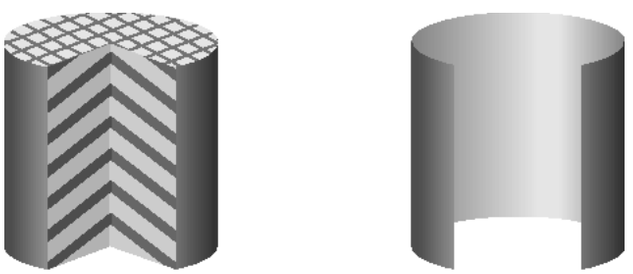


Figure 8. Cylinders with and without end caps or interior end caps

POLYHEDRAL PRIMITIVES

In version 1.5, QuickDraw 3D has four different types of shared-vertex primitives, or polyhedral primitives — the mesh and trigrid, and the newer polyhedron and trimesh. These primitives vary in such characteristics as memory use, rendering speed, and suitability for representing models. To a great extent, their usefulness is governed by how closely each primitive's design follows the original QuickDraw 3D design philosophy. (For some background, see the philosophical aside, "QuickDraw 3D Design Principles.") Of the four polyhedral primitives, the polyhedron and the trigrid best conform to the design standards set out by the QuickDraw 3D team. I'll begin by describing the design characteristics of the polyhedron, followed by discussion of the other three. Then I'll compare the four primitives and discuss their best uses.

POLYHEDRA

One common way of making a polyhedral primitive is to have an array of points, with a list of faces (often triangles) to organize the points. Each face usually consists of a list of indices into the list of vertices, so basically this is a polygon with one level of array-based indirection. If there is more than one face, the vertices can be shared by simply reusing the same array indices in each face. This allows the graphics system to run faster because the same point doesn't have to be transformed or shaded more than once, and it saves quite a lot of storage space. In addition, because two or more faces share only one real vertex, this type of polyhedral primitive can make it easier to program interactive editing.

The design philosophy for the geometry of type kQ3GeometryTypePolyhedron — from now on, let's call it the *polyhedron* — was to implement this idea in a way that was consistent with all the other QuickDraw 3D primitives. The basic entity for polygonal primitives (line, triangle, polygon, and so forth) is TQ3Vertex3D, which is an {*x*, *y*, *z*} location with an attribute set. For consistency with the rest of the geometric primitives, the polyhedron also uses this data structure for its vertices.

The vertices of adjacent triangular faces are shared simply by using the same vertex indices. Also, sets of attributes may be shared like other objects in QuickDraw 3D:

```
vertex->attributeSet = Q3Shared_GetReference(otherVertex->attributeSet);
```

Vertices can contain the same locations, but may or may not share attributes. This can be quite useful, for example, if you have a polyhedron you want to be generally smooth-looking, but it has some edges or corners where you want a discontinuity. For example, consider the cross section of a polyhedral object in Figure 9. Each location is shared, and vertices at positions A, B, D, and E share normals, while the vertices at position C share the location but not the normal. So when smooth-shaded, the object has an edge or corner at position C but appears smooth elsewhere.

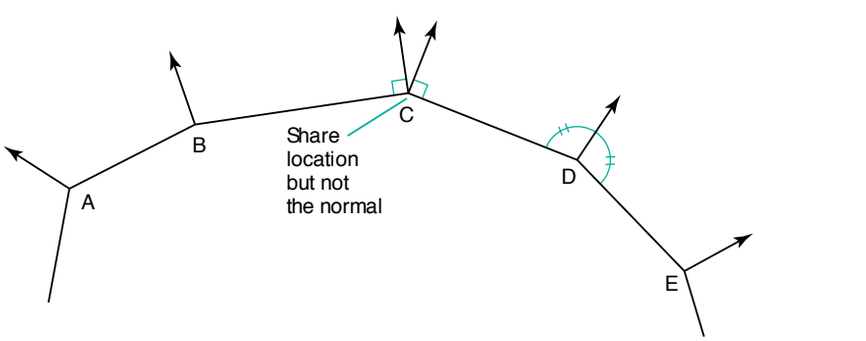


Figure 9. A cross section of a polyhedron with all vertices sharing locations but not attributes

QUICKDRAW 3D DESIGN PRINCIPLES

The founders of the QuickDraw 3D team had as a primary design tenet that retained mode and immediate mode would be coequal in the API in every "renderable" component (geometric primitives, transforms, styles, and attributes). Most other graphics systems support only one mode or, if both, one is given short shrift in the API. The intention for QuickDraw 3D was to allow developers to make decisions based on such things as their programming style and preferences and their particular program needs.

This intention manifests itself in several ways. For example, in most cases the calls that create an object take the address of a public data structure as an argument, and this argument is exactly the same as that for the immediate-mode Submit call:

```
TQ3GeometryObject      polyhedron;
TQ3PolyhedronData      polyhedronData;
...    /* Fill in poly data structure here. */

/* Create a polyhedron object... */
polyhedron = Q3Polyhedron_New(&polyhedronData);

/* ...or use immed. mode with same struct. */
Q3Polyhedron_Submit(&polyhedronData, view);
```

Further, the Get and Set object-editing calls take arguments that correspond to the part of the public data structure being retrieved or modified (or both). Here, you see how to retrieve the sixth vertex from the polyhedron and add a normal to it:

```
TQ3Vertex3D     vertex;
TQ3Vector3D     normal = { 0, 1, 0 };

Q3Polyhedron_GetVertex(poly, 5, &vertex);
Q3AttributeSet_Add(vertex.attributeSet,
    kQ3AttributeTypeNormal, &normal);
Q3Object_Dispose(vertex.attributeSet);
```

It was also intended that the developer be able to mix immediate mode and retained mode graphics freely and use these modes alternately or even simultaneously for one geometric primitive (transform, and so forth). Thus, you can easily use immediate mode for a triangle and then later make it a retained object and place it in a group. If you have a retained object, you can retrieve its data structure and use it in an immediate-mode call. The following code retrieves the representation from a polyhedron object and then uses it to make an immediate-mode rendering call:

```
TQ3PolyhedronData      polyhedronData;

Q3Polyhedron_GetData(polyhedron, &polyhedronData);
Q3Polyhedron_Submit(&polyhedronData, view);
```

This capability is important if an application uses immediate mode but reads in models from 3DMF files (which creates objects). The application can retrieve the data structure from the object with a GetData call and then dispose of the object with Q3Object_Dispose.

There's also a rich set of editing calls for retained objects. While this makes the API rather large, it allows retained mode to have much of the flexibility of immediate mode. In some display-list or object-oriented graphics systems (or systems with both), such editing was often awkward to program and inefficient. The design of QuickDraw 3D, however, makes these operations easy, consistent, and convenient.

Another design principle that pervades QuickDraw 3D is that you define properties by creating attribute set objects and locating them as close as possible (in the data structures) to the item to which they apply. For example, attribute sets for vertices are contained in a data structure along with the location (coordinates):

```
typedef struct TQ3Vertex3D {
    TQ3Point3D          point;
    TQ3AttributeSet     attributeSet;
} TQ3Vertex3D;
```

The consistent use of attribute sets for vertices, faces, and entire geometric primitives has two significant benefits. First, it allows for a reduction in space when the same data is to be applied to a number of faces (or vertices or geometries). For example, a shared texture needs to be stored only once, and each face using it simply has a reference (a pointer) to it. Second, only a single modification to an attribute set is needed to change the properties of all faces (or vertices or geometries) that have a reference to that attribute set.

Not least, a notable design criterion was consistency in naming, ordering, and structuring, because with an API so large it would be easy to get lost. For example, all of the primitives that have explicit vertices (like polyhedra, lines, triangles, polygons, and trigrids) use as the type of their vertices TQ3Vertex3D (with the exception of the trimesh). Also, the editing calls for all the objects are cut from the same cloth.

Another advantage to this approach is that values in an attribute set apply to all vertices sharing that attribute set, so operations on it simultaneously affect all the vertices to which it's attached. Of course, this applies to attributes on faces as well. For example, though you can texture an entire object by attaching the texture to the attribute set for the object, you can more naturally associate a single texture with a group of faces by simply having each face contain a shared reference to the texture-containing attribute set. But for a single texture to span a number of faces, you need to make sure their shared vertices share texture coordinates. You can do this by simply having shared vertices of faces that are spanned by a single texture use the same attribute set, which contains texture coordinates (see Figure 10).

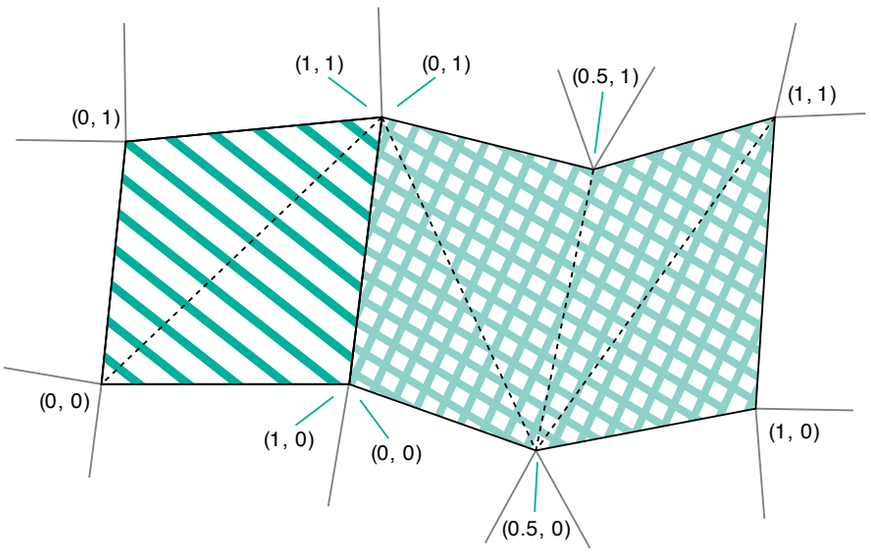


Figure 10. Applying textures that span a number of faces

Rendering the edges. Since the geometric primitives are generally array-based, the polyhedron needs an array of faces — and more information for a face. Besides an attribute set for the face, the three vertices defining a face are in an array (of size 3). The polyhedron also needs an enumerated type that tells us which edges are to be drawn, and which not:

```
typedef enum TQ3PolyhedronEdgeMasks {
    kQ3PolyhedronEdgeNone    = 0,
    kQ3PolyhedronEdge01      = 1 << 0,
    kQ3PolyhedronEdge12      = 1 << 1,
    kQ3PolyhedronEdge20      = 1 << 2,
    kQ3PolyhedronEdgeAll     = kQ3PolyhedronEdge01 |
                               kQ3PolyhedronEdge12 |
                               kQ3PolyhedronEdge20
} TQ3PolyhedronEdgeMasks;

typedef unsigned long TQ3PolyhedronEdge;
```

That way, by OR-ing these flags together, you can select which edges of a particular triangle you want drawn. For example, if you're using a wireframe renderer to draw an object like the one in Figure 11 (or you're using a scan-line or z-buffer type renderer that implements the "edges" fill style), you wouldn't have to show the "internal" edges, just the edges that represent the true border of the face. For face 0

in Figure 11, you could tell the system that you only want to display the edges between vertices 0 and 1, and between vertices 2 and 0, and not draw the edge between vertices 1 and 2. You'd do this by specifying (kQ3PolyhedronEdge01 | kQ3PolyhedronEdge20) as the edge mask.

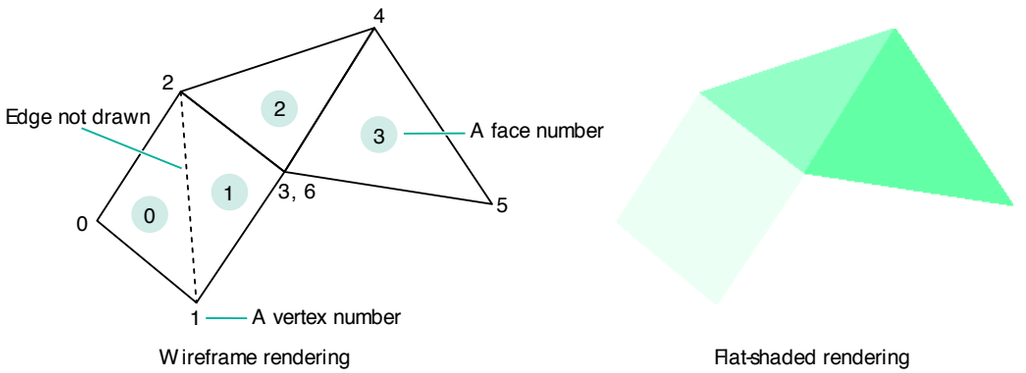


Figure 11. A wireframe and filled polyhedron

All this information is collected in this data structure:

```
typedef struct TQ3PolyhedronTriangleData {
    unsigned long           vertexIndices[3];
    TQ3PolyhedronEdge       edgeFlag;
    TQ3AttributeSet         triangleAttributeSet;
} TQ3PolyhedronTriangleData;
```

Of course, an alternative to using a mask to specify the edges would be to have a list of edges for the entire polyhedron. This can be advantageous in that if the renderer draws the edges (or lines, in the case of a wireframe renderer) from an edge list, the renderer can transform the points just once each and draw each edge just once, resulting in much faster rendering. So if you're willing and able to generate this representation of edges, there's a way to do this. The renderer ignores the edge flags in the face data structure if an array of these edges is present:

```
typedef struct TQ3PolyhedronEdgeData {
    unsigned long       vertexIndices[2];
    unsigned long       triangleIndices[2];
    TQ3AttributeSet     edgeAttributeSet;
} TQ3PolyhedronEdgeData;
```

As Figure 12 shows, the vertexIndices field specifies indices into the vertex array, one for the vertex at each end of the edge. The triangleIndices field specifies indices into the array of faces. You need to provide the indices to the faces that share this edge because in order to perform proper backface removal, the edge is drawn only if at least one of the faces that it's part of is facing forward.

The edgeAttributeSet field allows the application to specify the color and other attributes of the edges independently. If no attribute is set on an edge, the attributes are inherited from the geometry, or if that's not present, then from the view's state. Every edge must have two points, but edges may have one or two faces adjacent to them — those with just one are on a boundary of the object. To represent this in an array-based representation, you use the identifier kQ3ArrayIndexNULL as a face index for the side of the edge that has no face attached to it. Note the relationship

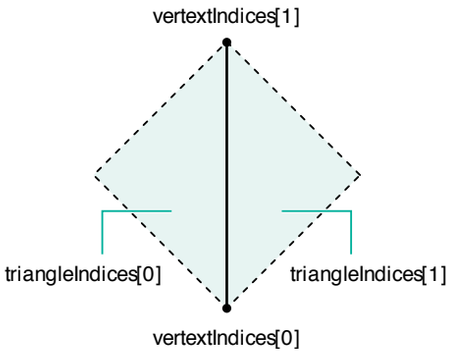


Figure 12. A schematic for filling out the polyhedron edge data structure

between the face indices and vertex indices in Figure 12. Relative to going from the vertex at index 0 to the vertex at index 1, the 0th face is to the left. If at all possible, fill out your data structures to conform to this schematic. For example, an application may want to traverse the edge list and be assured of knowing exactly which face is on which side of each edge.

The polyhedron data structure. Whew! Finally we're ready for the entire data structure:

```
typedef struct TQ3PolyhedronData {
    unsigned long                numVertices;
    TQ3Vertex3D                  *vertices;
    unsigned long                numEdges;
    TQ3PolyhedronEdgeData        *edges;
    unsigned long                numTriangles;
    TQ3PolyhedronTriangleData    *triangles;
    TQ3AttributeSet              polyhedronAttributeSet;
} TQ3PolyhedronData;
```

Creating a polyhedron. In Listing 1, you'll find the code that creates the four-faced polyhedron in Figure 11.

Listing 1. Creating a four-faced polyhedron

```
TQ3ColorRGB             polyhedronColor;
TQ3PolyhedronData       polyhedronData;
TQ3GeometryObject       polyhedron;
TQ3Vector3D             normal;

static TQ3Vertex3D   vertices[7] = {
   { { -1.0,   1.0,   0.0 }, NULL },
   { { -1.0,  -1.0,   0.0 }, NULL },
   { {  0.0,   1.0,   1.0 }, NULL },
   { {  0.0,  -1.0,   1.0 }, NULL },
   { {  2.0,   1.0,   1.0 }, NULL },
   { {  2.0,  -1.0,   0.0 }, NULL },
   { {  0.0,  -1.0,   1.0 }, NULL }
};
```

(continued on next page)

Listing 1. Creating a four-faced polyhedron *(continued)*

```
TQ3PolyhedronTriangleData triangles[4] = {
   {  /* Face 0 */
      { 0, 1, 2 },                                  /* vertexIndices */
      kQ3PolyhedronEdge01 | kQ3PolyhedronEdge20,    /* edgeFlag */
      NULL                                          /* triangleAttributeSet */
   },
   {  /* Face 1 */
      { 1, 3, 2 },
      kQ3PolyhedronEdge01 | kQ3PolyhedronEdge12,
      NULL
   },
   {  /* Face 2 */
      { 2, 3, 4 },
      kQ3PolyhedronEdgeAll,
      NULL
   },
   {  /* Face 3 */
      { 6, 5, 4 },
      kQ3PolyhedronEdgeAll,
      NULL
   }
};

/* Set up vertices, edges, and triangular faces. */
polyhedronData.numVertices    = 7;
polyhedronData.vertices       = vertices;
polyhedronData.numEdges       = 0;
polyhedronData.edges          = NULL;
polyhedronData.numTriangles   = 4;
polyhedronData.triangles      = triangles;

/* Inherit the attribute set from the current state. */
polyhedronData.polyhedronAttributeSet = NULL;

/* Put a normal on the first vertex. */
Q3Vector3D_Set(&normal, -1, 0, 1);
Q3Vector3D_Normalize(&normal, &normal);
vertices[0].attributeSet = Q3AttributeSet_New();
Q3AttributeSet_Add(vertices[0].attributeSet, kQ3AttributeTypeNormal,
   &normal);

/* Same normal on the second. */
vertices[1].attributeSet =
   Q3Shared_GetReference(vertices[0].attributeSet);

/* Different normal on the third. */
Q3Vector3D_Set(&normal, -0.5, 0.0, 1.0);
Q3Vector3D_Normalize(&normal, &normal);
vertices[2].attributeSet = Q3AttributeSet_New();
Q3AttributeSet_Add(vertices[2].attributeSet, kQ3AttributeTypeNormal,
   &normal);
```

(continued on next page)

Listing 1. Creating a four-faced polyhedron *(continued)*

```
/* Same normal on the fourth. */
vertices[3].attributeSet =
    Q3Shared_GetReference(vertices[2].attributeSet);

/* Put a color on the third triangle. */
triangles[3].triangleAttributeSet = Q3AttributeSet_New();
Q3ColorRGB_Set(&polyhedronColor, 0, 0, 1);
Q3AttributeSet_Add(triangles[3].triangleAttributeSet,
    kQ3AttributeTypeDiffuseColor, &polyhedronColor);

/* Create the polyhedron object. */
polyhedron = Q3Polyhedron_New(&polyhedronData);

...    /* Dispose of attributes created and referenced. */
```

Listing 2 shows the code that you'd use to specify the edges of the polyhedron in Figure 11, but this time with the optional edge list. You would add this code to the code in Listing 1, except that if you're using the edge list, you should set the edge flags in the triangle data to some legitimate value (like kQ3EdgeFlagAll), which will be ignored.

Listing 2. Using an edge list to specify the edges of a polyhedron

```
polyhedronData.numEdges = 8;
polyhedronData.edges = malloc(8 * sizeof(TQ3PolyhedronEdgeData));

polyhedronData.edges[0].vertexIndices[0]     = 0;
polyhedronData.edges[0].vertexIndices[1]     = 1;
polyhedronData.edges[0].triangleIndices[0]   = 0;
polyhedronData.edges[0].triangleIndices[1]   = kQ3ArrayIndexNULL;
polyhedronData.edges[0].edgeAttributeSet     = NULL;

polyhedronData.edges[1].vertexIndices[0]     = 2;
polyhedronData.edges[1].vertexIndices[1]     = 0;
polyhedronData.edges[1].triangleIndices[0]   = 0;
polyhedronData.edges[1].triangleIndices[1]   = kQ3ArrayIndexNULL;
polyhedronData.edges[1].edgeAttributeSet     = NULL;

polyhedronData.edges[2].vertexIndices[0]     = 1;
polyhedronData.edges[2].vertexIndices[1]     = 3;
polyhedronData.edges[2].triangleIndices[0]   = 1;
polyhedronData.edges[2].triangleIndices[1]   = kQ3ArrayIndexNULL;
polyhedronData.edges[2].edgeAttributeSet     = NULL;

polyhedronData.edges[3].vertexIndices[0]     = 3;
polyhedronData.edges[3].vertexIndices[1]     = 2;
polyhedronData.edges[3].triangleIndices[0]   = 1;
polyhedronData.edges[3].triangleIndices[1]   = 2;
polyhedronData.edges[3].edgeAttributeSet     = NULL;

...    /* Specify the rest of the edges. */
```

Using the polyhedron to your best advantage. Before we leave the polyhedron, let's take a look at some of the characteristics that should make it the most widely used polyhedral primitive.

Geometric editing operations, which change the positions of existing vertices, are easy and convenient. In immediate mode, you simply alter the point's position in the array in the data structure, and rerender. For retained mode, you'll find a number of function calls that allow you to change vertex locations, as well as the usual assortment of Get and Set calls for attributes, faces, face attributes, and so forth.

Topological editing operations change the relationships between vertices, faces, edges, and the entire object. Though you can do these operations, the addition or deletion of vertices, faces, or edges may require reallocation of one or more of the arrays. Because the polyhedron has a public data structure, these operations are possible in both immediate mode and retained mode. So long as such operations aren't the primary ones required for using the polyhedron, it's not a problem; however, in the case where they are, you should use the mesh primitive.

The polyhedron uses memory and disk space in a maximally efficient manner because shared locations and attributes are each stored only once and only those parts that logically require attributes need to have them. This results in generally excellent I/O characteristics (though, as is true of all geometric primitives, the addition of textures requires a great deal of space and can increase I/O time significantly).

Finally, good to very good rendering speed is possible with the polyhedron, owing to the shared nature of the vertices. In short, you can easily use the polyhedron to represent almost any polyhedral object.

TRIMESHES

The trimesh primitive is similar to the polyhedron in that it has a list of points and a list of triangular faces that contain indices into the list of points. Similarly, it also has an optional edge list. However, beyond these general characteristics, the two primitives differ greatly. Indeed, the trimesh primitive differs in style from every other geometric primitive in QuickDraw 3D, and this significantly affects its applicability.

Three features characterize the design of the trimesh data structures:

- All the data is in explicit arrays — the locations of the vertices, vertex attributes, triangle attributes, and edge attributes.
- Unlike with all other QuickDraw 3D geometries, attributes are not kept in objects of type TQ3AttributeSet; rather, they're kept as (arrays of) explicit data structures. The exception to this is that the trimesh has a standard attribute set for its entire geometry, just like all the other primitives.
- Again, unlike with all other QuickDraw 3D geometries, you must have the exact same types of attributes on all vertices, faces, or edges, with the exception of custom attributes.

The third of these features, which I'll call the *uniform-attributes requirement*, makes it necessary for you to put, say, a color on every face if you want to put a color on just one of the faces (and it's similar for vertices and edges). For some types of models, this may not be a problem, in which case the trimesh is a good choice. In addition, preexisting applications that are to be ported to QuickDraw 3D and already use uniform attributes (particularly in immediate mode) may find this the easiest polyhedral primitive to use, as the "translation" is more direct. (This use was one of the motivations for creating the trimesh primitive.) In such cases, the trimesh may be faster and more compact.

Diverging from the design philosophy. The triangular face of a trimesh is simply a three-element array of indices into a location (TQ3Point3D) array, and an edge consists of indices into the location array and triangular face array:

```
typedef struct TQ3TriMeshTriangleData {
    unsigned long       pointIndices[3];
} TQ3TriMeshTriangleData;

typedef struct TQ3TriMeshEdgeData {
    unsigned long       pointIndices[2];
    unsigned long       triangleIndices[2];
} TQ3TriMeshEdgeData;
```

Note that this differs from the polyhedron, and most of the rest of the QuickDraw 3D primitives, in that the attributes associated with a part of the geometry are not closely attached to the geometric part. Instead, the normal — say, for vertex number 17 — is contained in the 17th element of an array of vertex normals, and it's the same for face and edge attributes. Of course, because you might have more than one type of attribute on a vertex, face, or edge, you might have an array of arrays of attributes. To keep things organized, the trimesh has a data structure that contains an identifier for the type of the attribute and a pointer to the array of values; so you actually will have an array of structures of the following type that contains arrays of data defining the attributes:

```
typedef struct TQ3TriMeshAttributeData {
    TQ3AttributeType    attributeType;
    void                *data;
    char                *attributeUseArray;
} TQ3TriMeshAttributeData;
```

For example, if a trimesh has 17 vertices with normals on them, you would create a data structure of this type, set attributeType to kQ3AttributeTypeNormal, allocate a 17-element array of TQ3Vector3D, and then fill it in appropriately. For all but custom attributes, the attributeUseArray pointer must be set to NULL. In the case of custom attributes, you can choose whether or not a particular vertex has that attribute by (in our example) allocating a 17-element array of 0/1 entries and setting to 1 the *n*th element if the *n*th vertex has a custom attribute on it (and 0 otherwise). You would use the same approach for vertex, face, and edge attributes.

The trimesh data structure. The data structure for the trimesh consists of the attribute set for the entire geometry, plus pairs of (count, array) fields for points, edges, and faces, and the attributes that may be associated with each:

```
typedef struct TQ3TriMeshData {
    TQ3AttributeSet             triMeshAttributeSet;
    unsigned long               numTriangles;
    TQ3TriMeshTriangleData      *triangles;
    unsigned long               numTriangleAttributeTypes;
    TQ3TriMeshAttributeData     *triangleAttributeTypes;
    unsigned long               numEdges;
    TQ3TriMeshEdgeData          *edges;
    unsigned long               numEdgeAttributeTypes;
    TQ3TriMeshAttributeData     *edgeAttributeTypes;
    unsigned long               numPoints;
    TQ3Point3D                  *points;
    unsigned long               numVertexAttributeTypes;
    TQ3TriMeshAttributeData     *vertexAttributeTypes;
    TQ3BoundingBox              bBox;
} TQ3TriMeshData;
```

Trimesh characteristics. The uniform-attributes requirement and the use of arrays of explicit data for attributes — as opposed to the attribute sets used throughout the rest of the system — may be advantageous for some models and applications and make the trimesh relatively easy to use. The simplicity of this approach, however, makes it very hard to use this primitive to represent arbitrary, nonuniform polyhedra. (You'll learn more about this at the end of this article when I compare the characteristics of the four polyhedral primitives.)

Geometric editing operations on the trimesh are similar to those on the polyhedron in immediate mode: you simply alter the point's position in the array in the data structure and rerender. There are no retained-mode part-editing API calls for the trimesh, as is befitting its design emphasis on immediate mode.

Topological editing in immediate mode is also similar to that on the polyhedron. However, unlike the polyhedron, there are no retained-mode part-editing calls, so editing an object topologically is not possible.

The uniform-attributes requirement for this primitive results in generally good I/O characteristics. However, the redundant-data problem that's inherent in this requirement may cause poor I/O speeds due to the repeated transfer of multiple copies of the same data (for example, the same color on every face). Rendering speed for the trimesh is generally good to very good.

MESHES

Here I'll expand on the information about the mesh primitive that's in *3D Graphics Programming With QuickDraw 3D*, focusing on its use.

Like the polyhedron and trimesh, the mesh is intended for representing polyhedra. However, it was designed for a very specific type of use and has characteristics that make it quite different from the polyhedron and trimesh (and all the other QuickDraw 3D primitives).

The mesh is intended for interactive topological creation and editing, so the architecture and API were designed to allow for iterative construction and topological modification. By *iterative construction* I mean that you can easily construct a mesh by building it up face-by-face, rather than using an all-at-once method of filling in a data structure and constructing the entire geometric object from that data structure (which you do with all the other geometric primitives). By *topological modification* I mean that you can easily add and delete vertices, faces, edges, and components. A particularly notable feature of this primitive is that it has no explicit (public) data structure, and so it has no immediate-mode capability, as do all the other geometric primitives.

The mesh is specifically *not* intended to be used for representation of large-scale polyhedral models that have a lot of vertices and faces. If you use it this way, you get extremely poor I/O behavior, enormous memory usage, and less-than-ideal rendering speed. In particular, individuals or companies creating 3DMF files should immediately cease generating large models in the mesh format and instead use the polyhedron. Modeling, animation, and design applications should also cease using the mesh and begin using the polyhedron for most model creation and storage.

The reason for this is that meshes consist of one or more components, which consist of one or more faces, which consist of one or more contours, which consist of a number of vertices. To enable the powerful topological editing and traversal functions, each of these entities must contain pointers not only to their constituent parts, but also to the entity of which they are parts. So a face must contain a reference to the

component of which it is a part, and references to the contours that define the face itself. All of this connectivity information can significantly dwarf the actual geometrical information (the vertex locations), so a nontrivial model can take up an unexpectedly large amount of space in memory. The connectivity information (the pointers between parts) can take up from 50% to 75% of the space. Further, when reading a mesh, QuickDraw 3D must reconstruct all the connectivity information, which is computationally expensive. Writing is relatively slow as well, primarily because of the overhead incurred by the richly linked architecture. In addition, models distributed in the mesh format do a tremendous disservice to subsequent users of these models who might want to extract the data structure for immediate-mode use. This won't be possible because the mesh has no public data structure.

These somewhat negative characteristics don't mean this isn't a useful primitive. For the purposes for which it was designed, it's clearly superior to any other available QuickDraw 3D geometric primitive. For example, if you have an application that uses a 3D sampling peripheral (for instance, a Polhemus device) for digitizing physical objects, the mesh would be ideal. You can easily use the mesh in such situations to construct the digitized model face-by-face, merge or split faces, add or delete vertices, and so forth. Doing this sort of thing with an array-based data structure would be awkward to program and inefficient becuase of the repeated reallocation you'd be forced to do.

To give you an idea of the richness of the API and the powerful nature of this primitive, you can expect to find routines to create and destroy parts of meshes, retrieve the number of parts (and subparts of parts), get and set parts, and iterate over parts (and subparts). And because the iterators are so essential to the editing API, you'll find a large set of convenient macros for common iterative operations.

Mesh characteristics. The mesh API richly supports both geometric and topological editing operations, but only for retained mode because the mesh has no immediate-mode public data structure — an inconsistency with the design goals of the QuickDraw 3D API. (You should use the polyhedron primitive if immediate mode is desired.)

In general, the rendering speed of meshes is relatively slow. In the case of the polyhedron and trimesh, faster rendering is facilitated by the use of arrays of points, which are presented to renderers in the form of a public data structure. The mesh, having neither an array-based representation nor a public version of the same, must be either traversed for rendering or decomposed into some other primitives that are more amenable to faster rendering. However, traversing usually results in retransformation and reshading of shared vertices (which tends to be extremely slow), while decomposition may involve tremendous use of space as well as complex and slow bookkeeping code.

Faces of meshes (unlike those in the polyhedron and trimesh) may have more than three vertices, may be concave (though not self-intersecting), and may have holes by defining a face with more than one contour (list of vertices).

Using the mesh. Listing 3 creates a mesh that's geometrically equivalent to the polyhedron created in Listing 1.

TRIGRIDS

Like the mesh, the trigrid has been around since the first release of QuickDraw 3D and is quite simple, so I won't go into a long discussion here. The basic idea is that you have a list of items of type TQ3Vertex3D, each representing a series of rows of vertices in a topologically rectangular grid, as illustrated in Figure 13.

Listing 3. Creating a mesh

```
static TQ3Vertex3D  vertices[7] = {
   { { -1.0,  1.0,  0.0 }, NULL },
   { { -1.0, -1.0,  0.0 }, NULL },
   { {  0.0,  1.0,  1.0 }, NULL },
   { {  0.0, -1.0,  1.0 }, NULL },
   { {  2.0,  1.0,  1.0 }, NULL },
   { {  2.0, -1.0,  0.0 }, NULL },
   { {  0.0, -1.0,  1.0 }, NULL },
};
TQ3MeshVertex        meshVertices[7], tmp[4];
TQ3GeometryObject    mesh;
TQ3MeshFace          face01, face2, face3;
TQ3AttributeSet      faceAttributes;
unsigned long        i;
TQ3ColorRGB          color;
TQ3Vector3D          normal;

/* Add normals to some of the vertices. */
vertices[0].attributeSet = Q3AttributeSet_New();
Q3Vector3D_Set(&normal, -1, 0, 1);
Q3Vector3D_Normalize(&normal, &normal);
Q3AttributeSet_Add(vertices[0].attributeSet, kQ3AttributeTypeNormal,
   &normal);

vertices[1].attributeSet =
   Q3Shared_GetReference(vertices[0].attributeSet);

vertices[2].attributeSet = Q3AttributeSet_New();
Q3Vector3D_Set(&normal, -0.5, 0.0, 1.0);
Q3Vector3D_Normalize(&normal, &normal);
Q3AttributeSet_Add(vertices[2].attributeSet, kQ3AttributeTypeNormal,
   &normal);

vertices[3].attributeSet =
   Q3Shared_GetReference(vertices[2].attributeSet);

/* Create the mesh. */
mesh = Q3Mesh_New();

/* Create the mesh vertices. */
for (i = 0; i < 7; i++) {
   meshVertices[i] = Q3Mesh_VertexNew(mesh, &vertices[i]);
}

/* Create a quad equal to the first two triangles in the polyhedron. */
tmp[0] = meshVertices[0];
tmp[1] = meshVertices[1];
tmp[2] = meshVertices[3];
tmp[3] = meshVertices[2];
face01 = Q3Mesh_FaceNew(mesh, 4, tmp, NULL);
```

(continued on next page)

Listing 3. Creating a mesh *(continued)*

```
/* Create other faces. */
tmp[0] = meshVertices[2];
tmp[1] = meshVertices[3];
tmp[2] = meshVertices[4];
face2 = Q3Mesh_FaceNew(mesh, 3, tmp, NULL);

tmp[0] = meshVertices[6];
tmp[1] = meshVertices[5];
tmp[2] = meshVertices[4];
face3 = Q3Mesh_FaceNew(mesh, 3, tmp, NULL);

/* Add an attribute set to the last face. */
faceAttributes = Q3AttributeSet_New();
Q3ColorRGB_Set(&color, 0, 0, 1);
Q3AttributeSet_Add(faceAttributes, kQ3AttributeTypeDiffuseColor,
    &color);
Q3Mesh_SetFaceAttributeSet(mesh, face, faceAttributes);
```

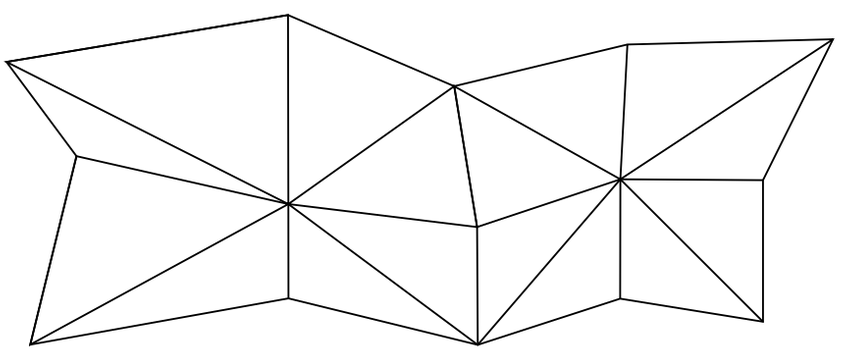


Figure 13. A trigrid

The numbers of rows and columns are part of the data structure, and there is an optional array of type TQ3AttributeSet for face attributes.

This primitive has a fixed topology, defined by the numbers of rows and columns. Thus, the space in memory and in files is very efficiently used. I/O is relatively fast because of the simplicity and efficiency of the primitive. Rendering can also be fast because of the shared nature of the points and the fixed topology. However, the fixed topology and the fact that shared locations must share attributes restrict the generality and flexibility of this primitive.

COMPARING THE POLYHEDRAL PRIMITIVES

It should be clear that the four polyhedral primitives in QuickDraw 3D can be used to represent the same sorts of shapes. It should also be clear that there are some important differences in their generality, flexibility, style of programming, performance, and compliance with the overall design goal of treating retained and immediate mode programming as equivalent. To help you determine the usefulness of these polyhedral primitives, Table 1 compares a number of their important characteristics.

Table 1. Comparison of polyhedral primitives

Characteristic	Polyhedron	Trimesh	Mesh	Trigrid
Memory usage	Very good	Fair to very good	Poor	Very good
File space usage	Very good	Fair to very good	Very good	Very good
Rendering speed	Good to very good	Good to very good	Fair to good	Good to very good
Geometric object editing	Very good	Impossible (no API calls)	Very good	Very good
Topological object editing	Poor	Impossible (no API calls)	Very good	Impossible (fixed topology)
Geometric data structure editing	Very good	Very good	Impossible (no data structure)	Very good
Topological data structure editing	Fair	Fair	Impossible (no data structure)	Impossible (fixed topology)
I/O speed	Good to very good	Fair to very good	Fair	Good to very good
Flexibility/generality	Good	Poor	Very good	Poor (fixed topology)
Suitability for general model representation and distribution	Very good	Fair	Fair	Poor

An upcoming release of QuickDraw 3D will have the capability of generating one geometric primitive from another, but with a different type — for example, getting a group of triangles that corresponds to a cone, or a mesh that's equivalent to a trimesh, or a polyhedron that's equivalent to a mesh.•

Most of the characteristics in Table 1 were covered in greater detail in the sections that described each primitive. So let's look at the last characteristic — suitability for general model representation and distribution — to help you determine when you would use one of these primitives. We'll look at the primitives one by one.

The polyhedron primitive. This polyhedral primitive is the primitive of choice for the vast majority of programming situations and for the creation and distribution of model files if editing of models is desired. Companies and individuals whose businesses involve creation of, conversion to, distribution of, or sale of polyhedral models should produce them in polyhedron format, rather than mesh or trimesh. User-level applications such as modelers and animation tools should generally use the polyhedron as well. Creators of plug-in renderers are required to support certain basic primitives (triangles, points, lines, and markers) and are also *very* strongly urged to support the polyhedron.

Let's quickly recount some of the pluses for the polyhedron: it can easily represent arbitrarily shaped polyhedral models in a space-efficient fashion, it's amenable to fast rendering, it's highly consistent with the rest of the API, and attributes may be attached in whatever combination is appropriate for the model. The polyhedron has advantages over the mesh because of the mesh's profligate use of space and lack of immediate mode.

The mesh primitive. You should use the mesh primitive for interactive construction and topological editing. The rich set of geometric and topological object editing calls, the ability to make nontriangular faces directly, the allowance of concave faces and faces with holes, and the consistent use of attribute sets make this primitive ideal for those purposes. In addition, the 3DMF representation of a mesh is quite space

efficient. However, because the mesh lacks an immediate mode, it requires a large amount of memory and is generally "overkill" in terms of representation for other uses.

The trigrid primitive. Because of its fixed rectangular topology, the trigrid is a good choice for objects that are topologically rectangular — for example, surfaces of revolution, swept surfaces, and terrain models — and as an output primitive for applications that want to decompose their own parametric or implicit surfaces. If the situation matches one of these criteria and space is a serious issue, the trigrid is an especially good choice because it's more space efficient than the other primitives discussed here and it's very consistent with the rest of the QuickDraw 3D API.

The trimesh primitive. I promised earlier that I'd discuss the implications that the uniform-attributes requirement has on the suitability of this primitive for representing general polyhedral objects. Real objects have regions that are smoothly curved and regions that are intentionally flat or faceted, and often have sharp edges, corners, and creases. The vertices in the curved regions need normals that approximate the surface normal at that vertex, but vertices at corners or along edges or that are part of a flat region need none. On a polyhedron, mesh, or trigrid, you need only take up storage (for the normal) on those vertices that actually require a normal, but on a trimesh you would be required to place vertex normals on all the vertices, resulting in a tremendous use of space.

This same problem can be seen for face attributes. Real objects often have regions that differ in color, transparency, or surface texture. For example, a soccer ball has black and white faces, and a wine bottle may have a label on the front, a different one on the back, and yet another around the neck. The other polyhedral primitives would, in the case of the soccer ball, simply create two attribute sets (one for each color) and attach a reference to the appropriate attribute set to each face, thus sharing the color information. In a trimesh, you would be required to create an array of colors, thus using quite a lot of space to represent the same data over and over. If you wanted to highlight one face, you couldn't simply attach a highlight switch attribute to that face (set to "on") — you'd need to attach it to the rest as well (set to "off"). As for the wine bottle, you would want to attach the label textures to the appropriate faces on the bottle, which would require attaching texture parameters to the vertices of the faces to which you attached the label texture. With a trimesh, this extremely useful and powerful approach is simply not possible.

In using the trimesh for large polyhedral models, these problems can result in a rather startling explosion of space, both on disk and in memory. Consider a 10,000-face model whose faces are either red or green. The other polyhedral primitives would use references to just two color attribute sets while the trimesh would use up 10,000 x 12 bytes = 120,000 bytes. Further, if the red faces were to be transparent, we would have to use up yet another 120,000 bytes. Highlighting just one face would require another 40,000 bytes. This same sort of data explosion can occur with vertex attributes as well. Note that these problems do not affect the other polyhedral primitives.

Thus, developers should carefully weigh the potentially negative consequences of the trimesh's characteristics when considering its use in applications. Its lack of object-editing calls renders it almost useless for an object-oriented approach, and this inconsistency with the rest of the QuickDraw 3D library may make its inclusion in a program awkward. In addition, because the trimesh doesn't use attribute sets (which are the foundation of the rest of the geometric primitives) for vertices, faces, and edges, it requires special-case handling in the application.

In spite of these features that limit the suitability of the trimesh for general-purpose polyhedral representation, the uniform-attributes requirement makes it ideal for

models in which each vertex or face naturally has the same type of attributes as the other vertices (or faces), but with different values. For example, if your application uses Coons patches, it could subdivide the patch into a trimesh with normals on each vertex. Games often are written with objects such as walls, or even some stylized characters, that typically have just one texture for the entire thing and either no vertex attributes or, more often, normals on every vertex. Multimedia, some demo programs, and other "display-only" applications in which the user typically is unable to modify objects may find the trimesh useful, at least for those primitives that don't suffer from the size problems described earlier.

DRAWING TO A CLOSE

Well, I have to say that I would have liked to have waxed eloquent a bit longer regarding these "primitive creations" for QuickDraw 3D. But for the most part, you have the long and short of it: some new high-level primitives to save you time and two new polyhedral primitives. Use them well and wisely — and have fun doing it!

RELATED READING

- "QuickDraw 3D: A New Dimension for Macintosh Graphics" by Pablo Fernicola and Nick Thompson, *develop* Issue 22.
- "The Basics of QuickDraw 3D Geometries" by Nick Thompson and Pablo Fernicola, *develop* Issue 23.
- "NURB Curves: A Guide for the Uninitiated" by Philip J. Schneider, *develop* Issue 25.
- "Adding Custom Data to QuickDraw 3D Objects" by Nick Thompson, Pablo Fernicola, and Kent Davidson, *develop* Issue 26.
- *3D Graphics Programming With QuickDraw 3D* by Apple Computer, Inc. (Addison-Wesley, 1995).

Thanks to our technical reviewers Rick Evans, Pablo Fernicola, Jim Mildrew, Klaus Strelau, and Nick Thompson.•

PRINT HINTS

Safe Travel Through the Printing Jungle

DAVE POLASCHEK

Implementing printing in a Macintosh application should be pretty straightforward, right? There are currently 18 high-level printing calls (listed on pages 9-92 and 9-93 of *Inside Macintosh: Imaging With QuickDraw*), which is only three more than were listed in *Inside Macintosh* Volume II. Calling them in the right order gives you a printing port that you can treat just like a graphics port — and every Macintosh application knows (or at least ought to know) how to draw into a graphics port.

But in spite of this apparent simplicity, there are an astounding number of Macintosh applications that have problems printing. (Even products from Apple make the list once in a while.) I think one of the reasons for this is that, while basic QuickDraw printing is simple, printing is something that can be — and has been — made more complex by various "extensions" to the original printing architecture. These extensions offer greater control of the printing process, allowing you to take advantage of special features available on some printers and to draw in more sophisticated ways than QuickDraw allows. But they also introduce complexities that can get you in trouble if you're not careful. In this column I'll give a few examples of places where control comes only at the price of complexity, and therefore places where you need to tread very carefully, if at all.

PICTURE COMMENTS

Picture comments are, on the face of it, wonderful things. They let you embed commands in your output that can take advantage of particular printer features if they're available, and they're automatically ignored by printer drivers that don't support them. But there's a flip side: for every picture comment you use, you have to provide an alternative for those printers that don't support it. There are also a number of picture comments that should be avoided, as listed on page B-40 of *Inside Macintosh: Imaging With QuickDraw*. As with any complex and powerful tool, the potential for getting things wrong with picture comments is ever-present.

The SetLineWidth picture comment is a perfect example: not only is it supported by only a few printer drivers, but it's implemented slightly differently in each of them. On some printers the value you pass for the line width is used to modify the current line width (for instance, passing 1/2 will halve the current line width), and on others it's used as an "absolute" value (passing 1/2 will set the line width to 1/2 point, regardless of the previous width). To obtain the desired results, you have to write your code very carefully, and even then the SetLineWidth picture comment may not work on the printer driver that the user happens to be using — and there's no QuickDraw alternative. The territory here is treacherous. Unless you really need fractional line widths, it may be better to take the nice safe QuickDraw path.

PRGENERAL

The PrGeneral call added complexity to the Printing Manager — and even more complexity could be added by driver developers, often without accompanying documentation. After all, since supporting the various PrGeneral opcodes (in fact, supporting PrGeneral itself) is optional, printer drivers can define their own new opcodes and nobody need be the wiser — nobody, that is, except for the one developer who needs the new opcode and the functionality it provides. Things get even more confusing when the same added functionality is available via a different mechanism in a different printer driver, so the application has to start using special-case code for each printer driver it knows about. If you find yourself writing special-case code for particular printer drivers, stop! Back up and look for another solution.

One commonly used PrGeneral capability, provided by the getRslOp and setRslOp opcodes, is finding the resolution(s) supported by the printer you're using and setting the resolution you want to print with. There's clearly a need for this sort of capability. An application that shows graphs of curves or of raw data gathered from some source wants the graphs to look good. Plotting individual pixels at 72 dpi doesn't make for smooth-looking curves, so an application might be

DAVE POLASCHEK (davep@best.com), formerly of Apple's Developer Technical Support group, got so confused by the lack of weather in California that he moved back to Minnesota. This probably won't seem like such a smart move when Celsius and Fahrenheit show the same temperature and Dave starts singing that verse from Jimmy Buffett's "Boat Drinks" that goes, "This morning, I shot six holes in my freezer. I think I've got cabin fever. Somebody sound the alarm."

justified in asking to print at the highest resolution the printer is capable of. But is PrGeneral the right approach?

A potential problem with using PrGeneral to get and set resolutions is that you're depending on the printer driver to keep up with the times. The LaserWriter driver, for example, is used for printers from the original LaserWriter all the way up to high-end typesetters. The driver reports that the maximum physical resolution of the printer is 300 dpi, even if you're printing to a typesetter that's capable of 1270 or even 2540 dpi. The reason for this is that reporting a higher resolution could cause applications that create bitmaps at the printer's resolution to run into QuickDraw's limitations, such as the limit on rowBytes and the 32K maximum region size. This is something that we plan to address in future versions of LaserWriter 8, but currently an application that wants to know a PostScript™ printer's real maximum resolution has to either parse the PostScript Printer Description file (PPD) associated with it or query the printer directly, both of which are functions that the driver should have to worry about, not the application.

In this case, there's an alternative to PrGeneral: If you're going to be generating your data in a GWorld, just make sure the GWorld's resolution is whatever you need for best results. Then take that same GWorld and use CopyBits to copy the PixMap in it to the printer. If you provide appropriate source and destination rectangles, the implementation of CopyBits in the printer driver will scale the PixMap, and you'll be taking advantage of the resolution of the printer without having to worry about new coordinate systems.

Determining just what resolution you need is, however, still a tricky issue. For example, if you're printing a color image to a LaserWriter that can print only black-and-white images and only at 300 dpi, the color image you're displaying onscreen already has more detail than the printer can reproduce, so you don't need to worry about sending a higher-resolution image at all. The way to tell for sure if you have enough data is that your pixel density (in dpi) should be between one and two times the "screen frequency" (in lpi) for the printer. The default screen frequency for PostScript printers is listed in the PPD file for the printer, and in the future we'll be providing access to the PPD file parsing code that's contained in LaserWriter 8's PrintingLib, but for now you may just want to ask the user rather than parse it out yourself.

If you're generating line art or other data that needs to have "hard edges" in a GWorld that's going to be sent to the printer, you've got a different problem: unless you specify the data at the printer's resolution (or higher), it will need to be scaled up to the printer's resolution, producing large, blocky pixels. Your users will think you're a bozo, unless of course your product is *supposed* to make large, blocky pixels. The right solution is to avoid sending data that needs to have hard edges as bitmapped images, if at all possible. This is the sort of data that really should be maintained as objects. If you want to draw the letter *A*, for instance, ask QuickDraw to draw it to the printer for you if possible, rather than image it into a bitmap first. If you really need to generate bitmaps of hard-edged data, be aware that you'd better have your machete sharpened and ready, since you're heading into the brush. On the other hand, this may be a great opportunity to generate your own PostScript code.

POSTSCRIPT CODE

Generating your own PostScript code is another powerful technique that can get your application into trouble. In many cases, it's the right answer to a thorny dilemma; for instance, if you need drawing primitives that QuickDraw doesn't supply, this may be the only way to get them. After all, cubic Bézier curves are neat and powerful. The problem arises when the application developer either doesn't understand how to write compatible PostScript code or takes shortcuts in the PostScript code.

An excellent example is an old version of a certain very popular graphics program that saved its pictures with an EPS version of the graphic embedded in the PICT data. Unfortunately, the PostScript code in the EPS version depended on the **md** dictionary, a private dictionary used by the LaserWriter driver. After warning developers for years that the **md** dictionary was private, Apple Engineering felt justified in changing it. When the new version of the LaserWriter driver shipped, suddenly many graphics quit printing. The problem was made even worse by the fact that many of these graphics had been shipped as clip art, and they still occasionally pop up to bedevil us today.

The solution isn't to avoid PostScript code entirely. Just make sure that if you do generate it, the code is compatible and portable. Obviously, it shouldn't use any of the LaserWriter driver's private PostScript operators. If you make graphics with PostScript code embedded in them, be sure that the PostScript code they contain conforms to the EPS specification that's described by Adobe™ in the *PostScript Language Reference Manual*, Second Edition, Appendix H. Also, be sure to send the PostScript code with the PostScriptBegin, PostScriptHandle, and PostScriptEnd picture comments, as described beginning on page B-38 of *Inside Macintosh: Imaging With QuickDraw*.

Another thing application developers have tried over the years (with mixed success) is to do their own PostScript font management by talking directly to the printer. This is something applications really need to avoid. The LaserWriter driver knows how to handle the PostScript fonts needed to print a page (or series of pages). Applications that attempt to manage the fonts themselves are more likely to get poor font management for their efforts, since the LaserWriter driver (or the LaserWriter GX driver) will have a much harder time recognizing which fonts are needed on which page. When this happens, the drivers err on the side of safety: if there's any doubt about when a font is used, it will be included for the whole job, which is usually exactly what the developer was trying to avoid. Let the driver handle font management.

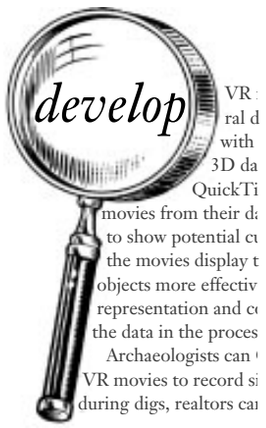
HAVE A SAFE TRIP

I've given a few of the more common examples of how printing has grown in complexity over the years, and how application developers can sometimes get in trouble by trying to take advantage of it. Printing doesn't need to be much harder than drawing to the screen if you stick to the rules, and even when you want to take advantage of particular printer capabilities, you can usually do so in safe, compatible ways. As tempting as it sometimes is to wander off the known path and plunge headlong into the uncharted jungle of possibilities, doing so usually just results in trouble — for you and for your users. In printing, as in all programming, remember: keep it simple.

RECOMMENDED READING

- Technote PR10, "A Printing Loop That Cares."
- *Writing Solid Code* by Steve Maguire (Microsoft Press, 1993).
- *The History of the English-Speaking Peoples* (4 volumes), by Sir Winston S. Churchill (Dodd, Mead, & Co., 1958 & 1959).

Thanks to Rich Blanchard, Paul Danbold, Dan Lipton, and Steve Simon for reviewing this column.



Looking to complete the set?

If you're looking for a complete *develop* collection, full-color, bound copies are available for $13 per issue, including shipping and handling. (Back issues are also on the *develop Bookmark* CD and the *Developer CD Series* Reference Library edition, as well as on the Internet.) For more information about how to order printed back issues (and where to find them online), see the inside front cover of this issue. *Supplies are limited. Please allow 4 to 6 weeks for delivery.*

Issue 1 Color; Palette Manager; Offscreen Worlds; PostScript; System 7; Debugging Declaration ROMs

Issue 2 C++ (Objects; Style Guide); Object Pascal; Memory Manager; MacApp; Object-Based Design

Issue 3 ISO 9660 and High Sierra; Accessing CD Audio Tracks; Comm Toolbox; 8•24 GC Card; PrGeneral

Issue 4 Device Driver in C++; Polymorphism in C++; A/ROSE; PostScript; Apple IIGS Printer Driver

Issue 5 (Volume 2, Issue 1) Asynchronous Background Networking; Palette Manager; Macintosh Common Lisp

Issue 6 Threads; CopyBits; MacTCP Cookbook

Issue 7 QuickTime 1.0; TrueType; Threads and Futures; C++ Objects in a World of Exceptions

Issue 8 Curves in QuickDraw; Date and Time Entry in MacApp; Debugging; Hybrid Applications for A/UX

Issue 9 Color on 1-Bit Devices; TextBox You've Always Wanted; Sound; Terminal Manager; Debugging Drivers

Issue 10 Apple Event Objects; Enhancements for the LaserWriter Font Utility; GWorlds; The Optimal Palette

Issue 11 Asynchronous Sound; Multibuffering Sounds; Exceptions; NetWork: Distributed Computing

Issue 12 Components; Time Bases; Apple Event Coding Through Objects; Globals in Standalone Code

Issue 13 Asynchronous Routines; QuickTime and Components; Debugging; Color Printing; DeviceLoop

Issue 14 Localizable Applications; 3-D Rotation; QuickTime (Video Digitizing; Making Better Movies)

Issue 15 QuickDraw GX; Component Registration; Floating Windows; Working in the Third Dimension

Issue 16 Making the Leap to PowerPC; PowerTalk; Drag and Drop From the Finder; Color Matching With QuickDraw GX; International Number Formatting

Issue 17 Newton Proto Templates; PowerPC (Standalone Code; Debugging); Thread Manager; Window Zooming

Issue 18 Apple Guide; Open Scripting Architecture; Graphics Speed on the Power Macintosh; Displaying Hierarchical Lists; Preferences Files

Issue 19 OpenDoc Part Handlers; PowerPC Memory Usage; Designing for the Power Macintosh; QuickDraw GX (Printing; Bitmaps); Inheritance in Scripts

Issue 20 AOCE; Make Your Own Sound Components; Scripting the Finder; NetWare on PowerPC

Issue 21 OpenDoc Graphics; Designing a Scripting Implementation; Dylan; Object-Oriented Hierarchical Lists

Issue 22 QuickDraw 3D; Copland; PCI Device Drivers; Custom Color Search Procedures; The OpenDoc User Experience; Futures

Issue 23 QuickTime Music Architecture; QuickDraw 3D Geometries; Internet Config; Multipane Dialogs; Document Synchronization; ColorSync 2.0

Issue 24 Speeding Up **whose** Clause Resolution; OpenDoc Storage; Sound; Alert Guidelines; Printing Faster With Data Compression; The New Device Drivers

Issue 25 QuickTime VR Movies From QuickDraw 3D; Flicker-Free Drawing With QuickDraw GX; NURB Curves; C++ Exceptions in C; Localized Strings for Newton

Issue 26 Mac OS 8; QuickTime Conferencing; OpenDoc and SOM Dynamic Inheritance; Adding Custom Data to QuickDraw 3D Objects; 64-Bit Integer Math on 680x0

Issue 27 Speech Recognition Manager; OpenDoc Part Kinds; Apple Guide 2.1 With OpenDoc; Mac OS 8 Assistants; Game Controls for QuickDraw 3D

QuickDraw GX Line Layout: Bending the Rules

Many high-end drawing applications provide the user with a way to draw text along an arbitrary path. Some of them even provide the means to edit this text in place. The methods discussed here will illustrate how to do both in a QuickDraw GX–based application. In fact, I'll show you how QuickDraw GX enables your application to provide better line layout capabilities than those currently available in mainstream graphics arts applications.

DANIEL I. LIPTON

One of the things that makes QuickDraw GX interesting is its amazing typography — whatever else has been said about GX, its type capabilities are universally acknowledged as the best. Besides aesthetics, a distinguishing attribute of QuickDraw GX typography is not what is graphically possible, but the ease with which visual content can be created. A user can create documents that are kerned or tracked or have swashes, ligatures, and so on without QuickDraw GX, but achieving these effects requires a great deal of manual labor, with lots of Shift-Option-Command key combinations and switching to special fonts — assuming the developer has made the effort to provide such a level of functionality. With QuickDraw GX, these effects are features designed into the font itself, so all the user needs to do is type and the text comes out beautifully. For different effects, the user just picks font features from a dialog or palette. The time saved by the user can be 90% over a non-GX interface! It simply has to be experienced to be appreciated.

All of this great typography is easy for the user to use because it's easy for the developer to implement. How many pre-GX drawing and illustration packages support the same level of typographic quality as their pre-GX publishing package counterparts? Not many, and of those that do, how much more memory do those applications require than they would without the typography code? Thanks to QuickDraw GX, we're beginning to see drawing and illustration programs that incorporate high-end *typographic* features and page layout programs that incorporate high-end *graphics* features — all with lower overall memory requirements (when compared with traditional non-GX applications) because the code is in the system and is shared between applications.

Let me get to the real topic of this article. A favorite feature in illustration and page layout programs is to enable the user to lay a line of text along an arbitrary path, be it straight or curved. My goal here is to provide the developer with a way to do this,

DANIEL I. LIPTON (lipton@apple.com) Although he's a member of the QuickDraw GX team, Daniel has been secretly developing a next-generation PostScript printer (based on the latest in artificial intelligence technology) that will achieve sentience on August 29, 1997. When not sleeping, Daniel can be found lost in San Francisco traffic or buying critters at his local aquarium shop.•

using QuickDraw GX, that's familiar to those who have been using GX and enticing to those who haven't tried it yet. If you've written code to handle text on a path in a non-GX application, consider the amount of code needed to do the editing on the screen and the amount of code you had to write to make it print on a PostScript printer. Then compare it to what's presented here — I think the QuickDraw GX advantage will be clear!

I'll assume you have some knowledge of typography and a working knowledge of QuickDraw GX throughout my discussion. If you need to brush up on GX, there are the *Inside Macintosh: QuickDraw GX* books; the *Programmer's Overview* and *Typography* volumes are most relevant here.

Accompanying this article on this issue's CD and *develop*'s Web site is a library of code called CurveLayout which very closely parallels the QuickDraw GX line layout mechanism but with support for *curved* layouts. The functions of this library will be described later.

SIMPLE INTERFACE TO DRAW TEXT ON A PATH

It turns out that QuickDraw GX provides an extremely simple method for drawing text along a curve, since it lets us make any shape a dash outline for any other shape — to *dash a shape* means to draw another shape in a repeating pattern along the perimeter of the first shape (see Figure 1). The PostScript language, by comparison, allow only simple dashing.

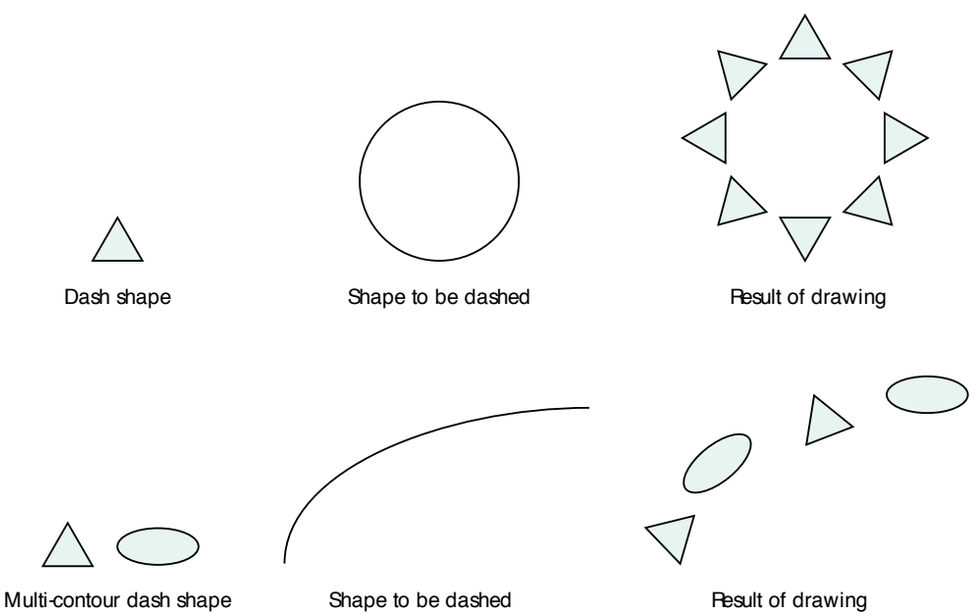


Figure 1. Dashing a shape

Since QuickDraw GX text is a shape, you can draw text along a path by using dashing, as shown in Listing 1.

Listing 1 demonstrates just how simple it can be to put text on a path in QuickDraw GX. This method would also work with layout shapes in addition to text shapes (although there's a bug in GX 1.1.3 and earlier that may cause a crash if you do try to dash with a layout). While dashing may be a simple interface for drawing text along a curve, it isn't an efficient solution for editing because of the overhead incurred by

Listing 1. Drawing text on a curve using dashing

```
void PutTextOnCurve(gxShape myPath)
{
    gxDashRecord    aDash;
    gxShape         textShape = GXNewText(5, "Hello", nil);

    // Call primitive shape to convert text to glyph.
    GXPrimitiveShape(textShape);      // All dashes must be primitive.
    aDash.dash = textShape;
    aDash.attributes = gxBreakDash;  // Dash each letter separately.
    aDash.advance = 0;               // 0 advance means single repeat.

    GXSetShapeDash(myPath, &aDash);
    GXDisposeShape(textShape);              // Dash is now sole owner of it.
    GXSetShapeFill(myPath, gxFrameFill); // Dash only for framed shapes.
}  // PutTextOnCurve
```

constantly rebuilding the dash every time the text changes. Additionally, the dashing solution puts you at the mercy of the algorithms used for dashing — they weren't specifically designed for text or layout manipulation.

BEHIND CURVELAYOUT

The CurveLayout library discussed here provides the illusion of a new shape type for QuickDraw GX which I affectionately call the "curve layout." I wanted to continue the object-oriented philosophy of GX by providing an API for drawing and editing text along a curved path. The idea is to provide an efficient mechanism for adding curve layouts to your application without forcing you to learn too much new stuff. While the article will go into great detail concerning the algorithms used, you need not understand them to incorporate curve layouts into your application using the CurveLayout library.

THE GLYPH SHAPE

Before we go too much further, it's worth discussing some things about our friend the glyph shape. There are three different kinds of shapes for drawing text: the text shape, the glyph shape, and the layout shape.

Text shapes are the most familiar to those of us who have used QuickDraw or PostScript. When a text shape is drawn, its appearance is the same as the result of the DrawString call in QuickDraw or the **show** operator in PostScript: simply the text itself.

The relationship between glyph and layout shapes can be compared to programming. The layout shape is like a high-level language such as LISP — a very high-level, powerful way of drawing text that produces beautiful results, but it makes specific manipulations difficult. The glyph shape is more like assembly language — you can control every aspect of how the text draws, but even simple things require a good deal of programming effort. Nevertheless, it's the direct control possible with the glyph shape that enabled me to write CurveLayout.

A glyph shape allows the specification of individual positions and tangent vectors (and even typographic style) of every typographic element drawn in a shape.

Figure 2 illustrates the power of the glyph shape for our purposes. At left is a typical straight piece of text. The arrows represent the tangent vectors stored in the glyph shape and the dots at the ends of the arrow lines show the positions. (Note that both here and in Figure 3, the tangent vectors are drawn as *normals* — rotated 90 degrees counterclockwise from their actual positions — since otherwise they would run together and make a mess of the figures.) Beside that is the same text with different tangents and positions showing how these attributes might be manipulated to draw text along a path. The tangent vector specifies the scale and rotation of the glyph and is stored in the glyph shape as a point. The *x* and *y* values of this point are used to construct the mapping shown at the right in the figure. This mapping is applied to the glyph to reposition it as desired.

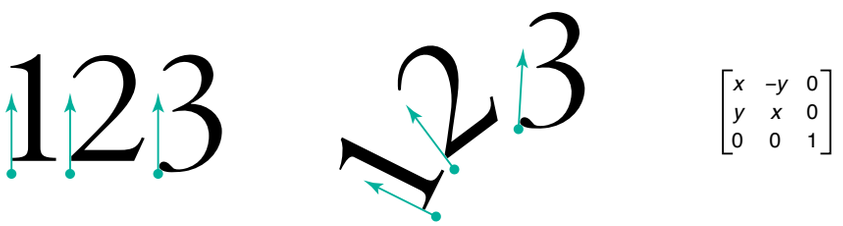


Figure 2. Glyph shape tangents, positions, and mapping matrix

But wait — while glyph shapes give us the control we need to flow our glyphs along a path (by setting positions and tangents for each glyph in the shape), they're still like assembly language, and we don't want to position every glyph ourselves if we don't need to. The good news is that QuickDraw GX provides a "compiler" to convert layout shapes to glyph shapes, so we can use the high-level language (layout shape) instead. Our compiler is the routine GXPrimitiveShape, which allows us to deal mostly in the beautiful world of layouts and then convert them into glyph shapes that can have their positions and tangents modified to flow along a path. Layout shapes also provide us with the high-level abilities required for interactive editing (more on this later).

Now we've seen how we can convert a line layout shape into a glyph shape, and how the glyph shape can have its positions and tangents modified so that the glyphs draw along a path, but how do we *generate* those positions and tangents?

POINTS ALONG A PATH

QuickDraw GX can evaluate points along paths. GXShapeLengthToPoint accepts a distance and a polygon or path and returns the point along that polygon or path that's the specified distance along its perimeter. In addition to the point, the tangent vector at that point (slope of the path or polygon) is returned, which is *exactly* what we need! With this information, we can lay glyph shapes along a path by simply modifying the tangents and positions appropriately.

The *x* coordinate of each glyph's position represents the current linear offset of the glyph in the layout (as if it were along a straight line). A glyph's position can be determined with the GXGetGlyphMetrics function and can be used as the length along the path for GXShapeLengthToPoint. The returned point and tangent can then be inserted into the glyph shape. The results of this technique are *almost* what we want, but consider the example shown in Figure 3.

In Figure 3, the arrows show the positions and tangents of glyphs that have been rotated around their positions, which are typically on the bottom left (at least for Roman fonts) rather than around their centers. Because of this, the text doesn't look

Figure 3. Glyphs positioned on a curve using *x* position as length along curve

quite right — some of the wider glyphs even seem to leap off the curve! We can do better than this. We can use the horizontal centers of the glyphs (that is, the center of the glyph along its baseline) as the input rather than their positions. Unfortunately, the point we put back in the glyph shape must be for the position of the glyph, not its center, so we have a little work to do translating back and forth.

Given a glyph shape, the function GXGetGlyphMetrics gave us the position information that we needed to compute the points and tangents for our first attempt. This function can also be used to obtain the bounding box of a glyph, and from the bounding box we can determine the horizontal center points of each glyph.

In Figure 4, we see the glyphs for the word "Pig." The dots represent the glyph positions obtained from GXGetGlyphMetrics and the starbursts represent the horizontal center of each glyph along the *x* axis, determined by the bounding boxes, which I've bisected for clarity. (Note that the position of a glyph doesn't necessarily fall on the left edge of the glyph's bounding box.)

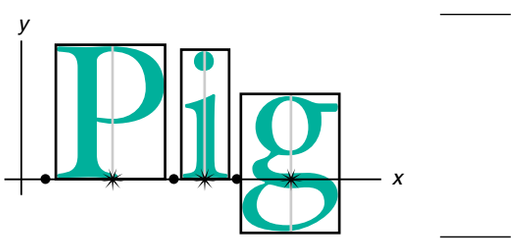


Figure 4. Glyph shape positions and horizontal centers

Listing 2 shows the loop for repositioning glyphs from CurveLayout. It illustrates how to compute the new glyph position given the location and tangent returned from GXShapeLengthToPoint, using the horizontal center to adjust the input length rather than merely using the glyph's *x* position. Before looping through the glyphs, we get some necessary information about the glyph shapes: the tangents (the glyph shape may have rotated glyphs to begin with, and we'd like to preserve them), the positions, and the bounding boxes. Then, for each glyph we do the following:

1. Compute the horizontal center of the glyph's bounding box.
2. Compute a vector that describes the glyph's position relative to the horizontal center of the bounding box (we'll use this in step 4).

3. Find out the position and tangent on the curve using the horizontal center of the glyph as the length along the curve.
4. Compute the new glyph position. Since earlier we described the glyph position as a vector relative to the horizontal bounding box center, and the point returned from GXShapeLengthToPoint is on the curve at the place where we want the horizontal bounding box *center*; we can compute the new glyph *position* as the vector we saved translated to the new position and rotated by the angle described by the tangent.
5. Compute the new tangent. This is the composition of the glyph's original tangent vector and the one from the path.

Listing 2. The CurveLayout glyph loop

```
// Get the positions, tangents, and bounding boxes of all glyphs.
GXGetGlyphs(theGlyphs, nil, nil, nil, nil, tangents, nil, nil, nil);
GXGetGlyphMetrics(theGlyphs, positions, glyphBoxes, nil);

// For each glyph, move its position to the correct place on the curve.
for (idx = 0; idx < glyphCount; ++idx) {
   // Compute glyph's horizontal center.
   pointToMapOnCurve.x = glyphBoxes[idx].left +
      (glyphBoxes[idx].right - glyphBoxes[idx].left) / 2;
   pointToMapOnCurve.y = 0;

   // Compute new glyph position relative to horizontal center.
   relativePosition.x = positions[idx].x - pointToMapOnCurve.x;
   relativePosition.y = positions[idx].y - pointToMapOnCurve.y;

   // Find the new location and tangent for horizontal center.
   GXShapeLengthToPoint(thePath, 0, pointToMapOnCurve.x, &newPoint,
      &newTangent);

   // New position will be the glyph's position relative to the
   // horizontal center rotated by the tangent. First rotate.
   ResetMapping(&aMapping);
   aMapping.map[0][0] = newTangent.x;
   aMapping.map[1][0] = -newTangent.y;
   aMapping.map[0][1] = newTangent.y;
   aMapping.map[1][1] = newTangent.x;
   MapPoints(&aMapping, 1, &relativePosition);
   // Now position this relative to the new point.
   positions[idx].x = newPoint.x + relativePosition.x;
   positions[idx].y = newPoint.y + relativePosition.y;

   // Concatenate the new tangent with the old.
   oldTangent.x = tangents[idx].x;
   oldTangent.y = tangents[idx].y;
   tangents[idx].x = FixedMultiply(oldTangent.x, newTangent.x) -
      FixedMultiply(oldTangent.y, newTangent.y);
   tangents[idx].y = FixedMultiply(oldTangent.x, newTangent.y) +
      FixedMultiply(oldTangent.y, newTangent.x);
}  // end for
```

The results of applying this algorithm are seen in Figure 5, where the round end of the line denotes the horizontal center of the bounding box and the square end of the line denotes the glyph position. On the left is a glyph with a line illustrating the relative location of the glyph's position and its horizontal bounding box center before wrapping. On the right is that same glyph positioned on a curve using our algorithm. You can see how the glyph here is positioned more naturally on the curve than was the case in Figure 3.

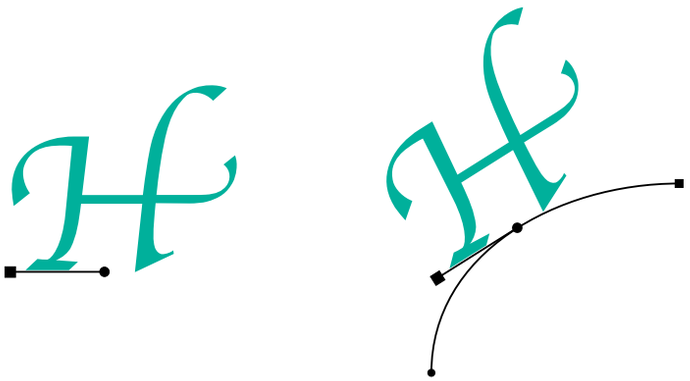


Figure 5. CurveLayout repositioning of a glyph

I've hand-drawn all the pictures so far using a QuickDraw GX–savvy draw program (LightningDraw GX) to illustrate the development of CurveLayout. Figure 6 is a picture generated with CurveLayout itself (see the test application that accompanies this article). The baseline curve is shown for clarity. Notice that the "te" ligature — a particularly wide glyph — in the word "pasteboard" is naturally tangent to the curve. Also, since this is QuickDraw GX, we can apply a number of transforms to the wrapped layout, such as the one shown in Figure 7.

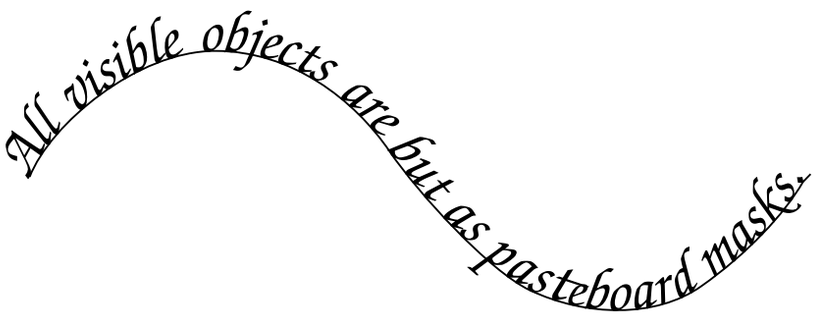


Figure 6. Output of the CurveLayout routine

DRAWING ISN'T ENOUGH — WE WANT TO EDIT

QuickDraw GX's ability to draw high-quality typographic content is only half of what's interesting about its line layout capabilities — GX also provides a number of routines to facilitate writing code to do *editing* of these beautiful lines of text, as follows:

- GXHitTestLayout for hit testing, to find out which character a given location is nearest. It's typically used for converting mouse clicks during editing. A point is converted to a character index.

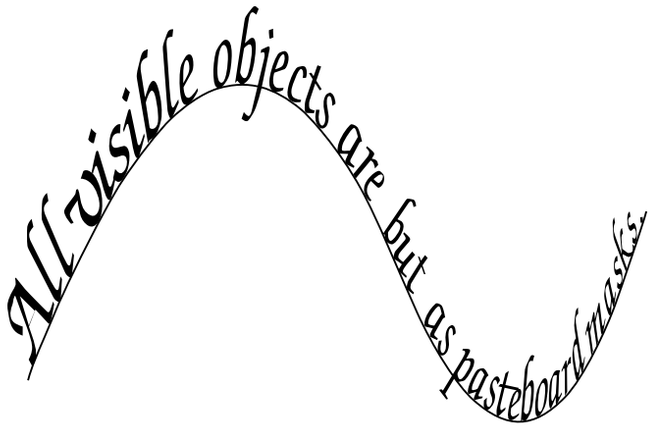


Figure 7. CurveLayout output with perspective transform applied

- GXGetLayoutCaret for computing the caret shape for a position between two characters. This is used for drawing the insertion point caret. A character index is converted into the caret shape.
- GXGetLayoutHighlight to compute the highlight shape for a run of characters, used for selecting text. A starting index and ending index are converted into a shape describing the highlight for the characters in the run.

CurveLayout wouldn't be complete if we couldn't provide these abilities. It's important for the user to be able to edit a layout on a curve in true WYSIWYG fashion. These three functions implement much of what's needed for a basic line editor. Editing whole blocks of text is left as an exercise for the programmer and is not a subject for this article!

HIT TESTING

The first part of editing involves being able to convert a mouse click into a character index. Since the function GXHitTestLayout does this for a normal line layout, we'd like to use as much of it as possible. The input to that function is a point relative to the layout. The important value of the point is the horizontal component. Provided that the position is within the layout, the horizontal component determines which character is nearest that value.

So, given a point on the curve, we want to undo the curvature to determine a value to pass to GXHitTestLayout. If GXShapeLengthToPoint converts a distance into a location, we need an inverse function that converts a location back to a distance. Unfortunately, QuickDraw GX has no such function. Additionally, the location of the mouse click may not be exactly *on* the shape.

Given a point, we can find the closest point on a curve. Knowing a point on a curve, we can determine the distance that point falls along the curve by doing a binary search of GXShapeLengthToPoint:

1. Find the length of the shape (GXGetShapeLength).
2. Do a binary search for the point on the curve closest to the hit point by calling GXShapeLengthToPoint, passing in lengths between 0 and the total length of the curve, and systematically reducing the distance to the point you seek until it's as close as you choose.

The length determined by this method can then be passed into GXHitTestLayout.

It's important to note that the algorithm above is simplified for the sake of brevity: it works on a single quadratic curve or line segment. The code included in the CurveLayout library implements the full functionality using the ShapeWalker library described in my Graphical Truffles column in *develop* Issue 27. The ShapeWalker library is used to determine each individual line or curve segment of a shape; the distance from a hit point to each segment is determined and the smallest of them is chosen (code comments illustrate the method more completely).

COMPUTING THE CARET IN CURVELAYOUT

To compute the caret of a line layout, we call GXGetLayoutCaret. This returns a shape describing the insertion position. If we take the resulting shape and use it to dash the curve (with only one repetition), the result is the curve layout caret. It's that simple (see Listing 3).

Listing 3. Computing the curve layout caret

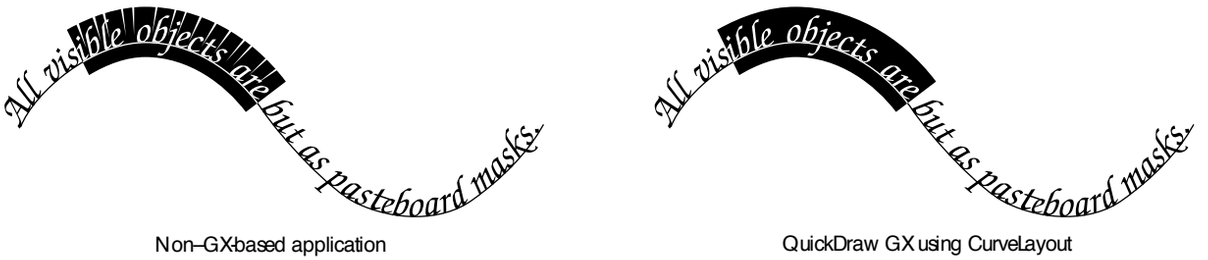
```
theFill = GXGetShapeFill(thePath);
realCaret = GXGetLayoutCaret(layout, offset, highlightType, caretType,
    nil);

// Copy into caret passed in (if it's nil, the right thing happens).
result = GXCopyToShape(caret, thePath);

GXSetShapePen(result, ff(1))     // We don't want dashes to scale.
theDash.attributes = gxBreakDash;
theDash.dash = realCaret;
theDash.advance = 0              // We want only one repeat.
theDash.phase = ff(0);
theDash.scale = ff(1);
GXSetShapeDash(result, &theDash);

// Make sure we have a framed shape for the path.
if ((theFill != gxFrameFill) && (theFill != gxClosedFrameFill))
    GXSetShapeFill(result, gxClosedFrameFill);

// Now apply the dash to get the caret on the path.
GXPrimitiveShape(result);
GXSetShapeDash(result, nil);
GXDisposeShape(realCaret);
```

USING DASHING TO HIGHLIGHT IN CURVELAYOUT

We can also use QuickDraw GX's dashing ability to highlight the individual characters in a curve layout. If we call GXGetLayoutHighlight for each individual character in a specified range, each resulting shape can be used to build up a dash shape. The dash shape built will have one contour per character, each contour representing the highlight of that character. If we then dash the curve with these contours, we get the desired effect.

Well, we *almost* get the desired effect — while the result does highlight the curved layout's text, the highlight has gaps. Achieving a contiguous highlight is a more difficult problem. In fact, popular PostScript-based drawing applications highlight text on a path with noncontiguous regions. Figure 8 compares noncontiguous highlighting similar to that of many popular non-GX-based applications with the

contiguous highlighting obtainable by QuickDraw GX using CurveLayout. The library supports both methods.



Non–GX-based application QuickDraw GX using CurveLayout

Figure 8. Two ways to highlight

The code in Listing 4 achieves the noncontiguous highlight of a curve layout. But we can do better: the CurveLayout library implements contiguous highlighting using QuickDraw GX's ability to do hairline dashing or bend dashing. Rather than going on and on, I'll let the code in the library speak for itself.

Listing 4. Creating the highlight with dashing

```
oneCharHighlight = nil;
newHighlight = GXNewShape(gxEmptyType);

for (idx = startOffset; idx < endOffset; ++idx) {
   oneCharHighlight = GXGetLayoutHighlight(theLayout, idx, idx + 1,
      highlightType, oneCharHighlight);
   if (oneCharHighlight != nil) {
      // Add highlight for individual character to highlight shape.
      GXSetShapeParts(newHighlight, 0, 0, oneCharHighlight,
         gxBreakLeftEdit);
   } // end if
} // end for

GXDisposeShape(oneCharHighlight);
theShape = GXCopyToShape(highlight, thePath);
GXSetShapePen(theShape, ff(1));     // We don't want dashes to scale.
GXSetShapeFill(theShape, gxFrameFill);
theDash.attributes = gxBreakDash;
theDash.dash = newHighlight;
theDash.advance = 0;                // We want only one repeat.
theDash.phase = ff(0);
theDash.scale = ff(1);
GXSetShapeDash(theShape, &theDash);
GXDisposeShape(newHighlight);
GXPrimitiveShape(theShape);
GXSetShapeDash(theShape, nil);

// Change the fill of the result to winding. Since when bent around,
// contours can overlap, we want the overlaps to fill, too. With
// even-odd, they wouldn't be filled and the highlight would look funny.
GXSetShapeFill(theShape, gxWindingFill);
```

CURVE MEASUREMENT

The current version of QuickDraw GX uses a numerical approximation for certain functions that are relevant here. The particular functions relate to the measurement of quadratic Bézier curve segments and are used by GXShapeLengthToPoint, by GXGetShapeLength, and for dashing as invoked by either GXDrawShape or GXPrimitiveShape. The implementation of these was done as a numerical approximation to ensure sufficient performance on 680x0 machines, where the use of a closed mathematical equation would have been too slow. The bad news is that while this method is accurate enough for the original use intended (primarily dashing), it's not accurate enough to produce good-looking text along a curve.

As you've seen in this article, the functions mentioned above are heavily used in the CurveLayout library. The inaccuracies will manifest themselves as characters being spaced a little too far apart. To ensure high-quality text layout, a library has been included with this article that reimplements the low-level quadratic segment measurement functions using a floating-point closed equation. On top of that code, I've built my own versions of GXShapeLengthToPoint, GXGetShapeLength, and GXPrimitiveShape, called AccurateShapeLengthToPoint, AccurateGetShapeLength, and AccuratePrimitiveShape. Within AccuratePrimitiveShape is an implementation of all cases of dashing that the CurveLayout library requires. All other cases of PrimitiveShape are dispatched to GXPrimitiveShape; hence this call isn't a complete replacement for GXPrimitiveShape. The CurveLayout.c file contains a compile-time conditional that can be defined to use the library code instead of the built-in QuickDraw GX code.

The low-level math used by my library was written by Joseph Maurer. He wrote measurement routines that operate on individual quadratic curve segments. I built the shape measurement and dashing functions by using the ShapeWalker library described in my Graphical Truffles column in *develop* Issue 27 to extract individual segments from QuickDraw GX shapes.

USING THE CURVELAYOUT LIBRARY

CurveLayout is designed to look very much like the line layout functions you're used to — we want a curve layout to look as if it really were a new QuickDraw GX shape type. All of the editing routines take the same parameters as the corresponding line layout routines, so I won't spend too much time describing those. See *Inside Macintosh: QuickDraw GX Typography* for more information.

The most important functions in the CurveLayout library are as follows:

```
gxShape ClNewCurveLayout(gxShape theText, gxShape theShape);
void ClDisposeCurveLayout(gxShape curveLayout);
```

The ClNewCurveLayout function creates the curve layout shape. The input is a normal line layout shape and a curve. Actually, the second parameter can be any kind of shape: a line, a polygon, or a path. The resulting curve layout shape is the one that will be passed to the other library functions for editing. It's also the shape you'd draw and to which you can attach transforms for achieving rotation or other effects. The ClDisposeCurveLayout function destroys the shape.

Modification of the text in the layout (such as for editing) should be done on the layout shape passed in. The path can be changed as well. If either of the two input shapes is modified, you should call the ClChangedCurveLayout function before redrawing. For example, this function should be called for each character inserted or deleted from the original line layout shape.

The code accompanying this article also includes an application that demonstrates using the CurveLayout library. While I'll certify the actual CurveLayout library as ready for prime time, the test application is just a test application — not a model of coding perfection! Also, as an extra bonus I've included the CurveLayoutGX control

panel. This control panel adds the basic curve layout feature to all of your *existing* drawing applications by defining a key combination to activate it. (This is a bit of a hack; use it at your own risk.)

All of the functions of the library are documented (as is the CurveLayout code itself) in the header file, CurveLayout.h, so I won't go further into the API. See "Curve Measurement" for a few other details about the implementation.

LAYING IT ALL ON THE LINE

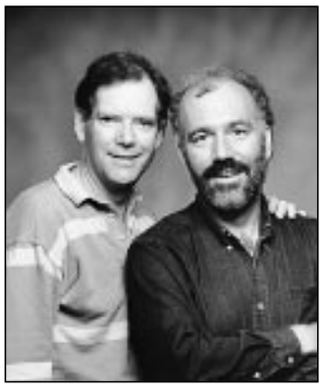
The library provided with this article should enable anybody writing a QuickDraw GX application to add high-quality typography drawn along an arbitrary curve. If you've already written code to edit QuickDraw GX line layouts, you'll find the API in this library very familiar. For those of you who haven't thought about writing a GX-based application, perhaps the simplicity of using CurveLayout as well as the quality demonstrated by the output of this code will make you give GX another look. Without QuickDraw GX, an application would have to include code similar to what's in the CurveLayout library in *addition* to the code the application would need to do ordinary line layout, not to mention all of the additional code that would be required in the application to print something like a curve layout shape.

CurveLayout provides developers with a way to provide distinctly superior user value in their applications, while reducing code size and complexity at the same time — what could be easier?

RELATED READING

- "Graphical Truffles: A Library for Traversing Paths" by Daniel Lipton, *develop* Issue 27.
- *Inside Macintosh: QuickDraw GX Programmer's Overview* and *Inside Macintosh: QuickDraw GX Typography* by Apple Computer, Inc. (Addison-Wesley, 1994).

Thanks to our technical reviewers Alex Beaman, Brian Chrisman, Dave Hersey, and Ingrid Kelly. Special thanks to Joseph Maurer for the low-level curve measurement routines that made accurate text layout on curves possible.



BE OUR GUEST

Source Code Control for the Rest of Us

D. JOHN ANDERSON AND ALAN B. HARPER

Imagine you're working on a program than runs on both Macintosh and Windows computers. Suppose you aren't the only programmer working on the project, and you occasionally want to work from home. Seems like a pretty common situation.

Soon everyone begins to make their own changes to the program and you realize that you need some kind of source code control. When people think of source code control, they often picture a safe database that keeps a history of every change to the project. Most source control systems do a reasonable job of that. But what surprisingly few people worry about is the much greater need to merge each programmer's changes into the common code base. For some reason this is where many popular source code control systems fall flat; here we'll tell you about one that doesn't.

Many systems require that you check out a file before modifying it. This is so that two people don't modify the same file at the same time. Unfortunately, this makes global changes difficult. Something as simple as changing a routine name from doQMFX to PrintFile doesn't get done because you need to check out every file and someone else already has them checked out; it's just not worth the hassle. In the early stages of development, when everyone is changing many files at the same time, the check-in/check-out model can slow you down even more. Some systems let a file be checked out by several people, who must then merge their changes into a single file. Even with the best tools, this process of merging often requires you to look at every individual change, presumably so that you can verify each one. But you typically end up incorporating every change anyway. Why not save a lot of time and headaches by automating this process completely?

There are other annoying problems with some source code control systems, such as limited cross-platform support and difficult or impossible access to the database from home. And with some systems that use a database, you can't do simple things like **grep** through your files. These problems can really get in the way of getting your work done.

If after reading this far you say, "Yeah, I hate source code control. I wish I just had a bunch of files and didn't have to think about other people stepping on my files," then read on. If instead you say, "I don't get what he's talking about. I don't mind checking files out and back in and merging them by hand," then skip to the next article.

OUR SOLUTION

Still with us? Great. To avoid these problems, we at Eclectus have developed three tools based on the GNU **diff** utility: Merge, Difference, and Undifference. They accompany this column on this issue's CD and *develop*'s Web site.

Here's an overview of how our scheme works:

1. We put all the files in the project in one directory tree — named, say, HotApp-0. We mark all the files read-only and put them on a shared server.
2. Each programmer makes a copy of HotApp-0 on his or her local machine. They do all their work in their own local copy, changing any file they want.
3. When everyone agrees that their changes are ready to merge into a new version, we use Merge, which automatically incorporates the changes into a new source tree that we name HotApp-1. When two people modify the same line, Merge reports a conflict that we must look at and resolve. Surprisingly, conflicts are rare. After fixing the conflicts by hand, we make any changes necessary to compile the application; then we mark each file read-only and put HotApp-1 on the shared server.

D. JOHN ANDERSON (jander@c2.org) and Alan are Eclectus Software. Together they write cross-platform applications for Windows and the Mac OS. Someday they plan to rule the consumer applications market. Jbhn lives in La Honda, California, where he writes software outdoors in a large tent with its very own ISDN line. At other times you might see him running or bicycling through the redwoods. His latest hobby, casting molds from faces, was inspired by the video "Better One-Piece Head Molds From Life."•

ALAN B. HARPER (aharper@dnai.com) learned recursive descent from Jbhn 15 years ago. His latest accomplishment is a fast cross-platform persistent object store — which means he can now write programs without worrying about serialization, undo, byte order, garbage collection, or running out of memory. In his off hours, Alan can be seen with other volunteers of the Golden Gate Raptor Observatory following radio-tagged hawks as they migrate through California.•

4. All programmers copy HotApp-1 to their local machines and development continues.

Suppose I want to bring some work home with me. I use Difference to create a difference script of changes between the current version — say, HotApp-1 — and my local working copy. This difference script is usually small enough to put on a floppy disk or beam over a modem. At home, where I also have a copy of HotApp-1, I use Undifference to restore the working copy of my project. I can copy my changes made at home back to work the same way.

If I want a copy of the source for safekeeping, I either copy HotApp-1 from the shared server or, if I really care about space, I difference HotApp-1 against HotApp-0 and compress the result, which is tiny.

If the machine I'm working on doesn't get backed up automatically every day, I can use the same method to back up my work. I just difference my working copy against the most recent shared copy, compress it, and put it in a safe place — like on a server that's backed up automatically. Compressed daily difference files are so small that I can keep a year's worth in only about 20 MB.

Because these tools are based on GNU **diff**, they're free (thanks to the Free Software Foundation), the source code is available, and they're extraordinarily fast. Using Difference, Undifference, and Merge is approximately as fast as copying an equivalent number of files. We can do a complete *n*-way merge — a merge between *n* programmers — in less time than it used to take to check out just a few files with competitive alternatives. We can difference or undifference our project (about 4 MB of source) in less than one minute on a low-end Power Macintosh or a Windows NT computer, and merge two trees in less than two minutes.

Because the source is freely available, you can easily tweak the tools to add your favorite features. Of course you'll have to share those features with everyone else, but that just makes the tools better.

A REAL-LIFE EXAMPLE

At Eclectus, our current project is a program that runs on both the Mac OS and Windows platforms, each developed at a different location. Let's say Alan and John are each modifying their own copy of HotApp-0.

1. When they decide it's time to merge their changes, Alan beams John his diffs against HotApp-0 via e-mail.
2. John uses Undifference on Alan's diffs to reconstruct Alan's source code tree. Now both John's source and Alan's source are on John's machine.
3. John merges Alan's source code with his own to create a new version, naming it HotApp-1, and makes any changes necessary to compile HotApp-1 on Windows.
4. John uses Difference on HotApp-1 against HotApp-0 and beams these diffs to Alan via e-mail.
5. Alan undifferences these diffs against HotApp-0 to construct HotApp-1, the same new version John now has. Alan might have to make some changes to get HotApp-1 to compile on the Macintosh because of John's recent changes that were automatically merged — for example, John might have forgotten to **ifdef** a Windows-specific piece of code. These changes will show up after the next merge of HotApp.

Development continues, and we repeat this process whenever we feel it's time to merge again. We usually use a modem or e-mail to exchange the diffs, since they're small.

This process generalizes to more than two programmers since the merge utility will handle any number of modified versions. Deciding when to merge is up to all the programmers. We usually merge when somebody wants to make their changes available to everyone else and nobody has totally broken their version. Sometimes months pass between merges, sometimes we merge twice a day. Near the end of a development cycle we like to merge at least once a day so that a recent copy is always available for testing.

DETAILS

Automatic merging, tiny diffs files, no time-consuming file checkout? Sounds great, but there are a lot of details you're probably wondering about.

What do the merges look like?

To merge changes, we use a command-line environment (MPW, ToolServer, and so on). In the above example, we would use the following Macintosh command to merge changes:

```
merge :HotApp-0 :HotApp-John :HotApp-Alan :HotApp-1
```

In this example, HotApp-0 is a directory tree containing the original code that both John and Alan started with. HotApp-John is a directory tree containing John's version after his changes, and HotApp-Alan contains Alan's version. HotApp-1 will contain the result of the merge when the command is finished. If we had more programmers, we'd just include their directories after Alan's. (With several programmers, you might think that the time involved in sending complete source trees back and forth would be prohibitive, but remember that the only thing that needs to be sent is a small diffs

file, and then the source tree can be rebuilt quickly and locally.)

When conflicts occur in the merge, the conflicting files are renamed by prefixing an exclamation point (!) to the filename, and the conflicting lines of code are marked in the merged file. Usually conflicts are solved by editing the merged file and taking one person's changes. After conflicts are resolved, you rename the file back to its original name.

Are automatic merges really safe?
Many programmers are suspicious of automatic merges. What if Joe makes a change that's incompatible with Helen's, but they don't change the same line? Merge doesn't identify this snafu. To avoid or identify bugs caused by incompatible changes, Joe and Helen must talk to each other or look at all their changes. If you're so inclined, you can give Merge an option that will list all the changes everyone has made so that you can review them individually.

Even so, after seeing automatic merges being used for over 10 years in lots of different projects with lots of programmers, we've found that the time saved and the elimination of human error from manual merges more than compensates for rare cases of incompatible changes. Of course, as you near the shipping date, it makes sense to have every change to the project carefully code-reviewed.

Can I rename a file?
Yes, as long as you do it right after a merge, when there's only a single copy of the source tree; otherwise Merge treats renamed files as new and they won't get properly merged with the previous source. But since Merge identifies files that were added or deleted from each person's source tree, it's easy to detect these situations.

What happens to junk files?
Imagine you created some temporary file named Junk. Merge lists this file as a newly created file found in your source tree.

If you like, these tools can also ignore certain files based on their filename extension. We like to put all our derived files, like object files, in a subdirectory named Derived.i and have Merge, Difference, and Undifference ignore files and directories that end in ".i". Just edit a table in the source code to define which extensions you want to ignore.

What happens if I rearrange a lot of code?
If two or more people modify the same routine and one of them moves the routine to a new place, Merge won't be able to merge the changes automatically. This case is treated the same as if two or more people modified the same line: you have to look at the changes and sort them out by hand. We often delay rearranging a lot of code until after a merge, but before we distribute the merged version.

How do you pick up someone's bug fix?
If someone has a fix to a bug that they're not ready to merge, but it's holding up your work, you can just copy their code containing the fix and put it in your project. Merge doesn't treat this as a conflict even though more than one person changed the code, because their changes were identical.

What about binary files?
Merge automatically handles one person changing a binary file, but not more than one. In the latter case you must merge by hand. For this reason, we try to keep as much of our project as possible in text rather than binary files.

Does this really work with more than two programmers?
T/Maker successfully used our tools for a number of years while developing WriteNow For Macintosh with five programmers. Various companies with many programmers working on multiple projects have been using successive automatic two-way merges for years.

What about cross-platform support?
Merge, Difference, and Undifference are written in C and currently run on the Mac OS (MPW with CodeWarrior 10 tools and libraries), Windows NT or 95 (Microsoft Visual C++ 4.0), and NeXT (GNU C). If you need to support a new platform, these tools should be easy to port as long as your platform has an ANSI C compiler.

Some special issues arise when working on cross-platform projects. Let's suppose I have all the code for a Macintosh project and I want to move it to Windows NT or 95.

1. I difference my source code tree against an empty directory on the Macintosh, resulting in a text file that I move to Windows.
2. Windows uses different end-of-line characters, so I run a little utility, MacToWin, to change the end-of-line characters. Going in the opposite direction, CodeWarrior IDE and BBEdit on Macintosh can automatically change end-of-line characters.
3. I undifference against an empty directory on Windows and my files are reconstructed. Even binary files are transferred correctly.

4. Finally, I name the directory containing the code HotApp-0 on both Windows and Macintosh.

We continue to develop as before, except we each have a separate copy of HotApp-0 on our respective platforms, rather than a shared copy on a single server. If I were to share a single server from both platforms, I would have two separate directory trees, one for each platform, each with the correct end-of-line characters. This would let you use tools like **grep** in each platform to search through the files in the project. When you need to synchronize the two source trees, you'll convert one of them to the other platform for the merge.

If you work on two different platforms at the same time, you could make a separate copy on each platform and treat each one as if it were owned by a different programmer.

Better yet, if you use CodeWarrior or BBEdit for editing on the Macintosh, the end-of-line problem goes away completely and you can keep just one source tree for both platforms.

What do you do with resource forks on a Macintosh?
When you use Undifference and Merge on a Macintosh, they take the resource fork for the new file from the original directory tree — that is, HotApp-0 in the above example. This means that after a merge, any changes you made to the resource fork since the last merge must be made by hand. However, once it's in a directory tree that everyone copies — HotApp-1 in the example — the resource fork will automatically be propagated to new versions by Undifference and Merge.

When you use Undifference and Merge on other platforms to incorporate changes from a Macintosh, resource forks are ignored. This means that you need to store any resources that you regularly edit in a text file and use a resource compiler like Rez.

What about changes to date and time?
Undifference preserves the date and time of a file in a new modified directory tree when there were no changes to the file. This means that you can iterate through many merge cycles and files that weren't changed will still have the same dates and times.

What about non-ASCII characters?
The Mac OS, Windows, and UNIX® platforms differ in their interpretation of characters outside the 7-bit ASCII range. These characters are usually not a problem when they're in a file used on only one platform. However, when one of these characters occurs in a file used on more than one platform, it's often a bug. Difference and Undifference identify files that contain non-ASCII characters to help detect this potential problem.

What about long filenames?
When developing on more than one platform, you're limited to the lowest common denominator filename length and path length, since the same file must have the same name on all platforms.

What about change comments?
The tools don't require you to add change comments. If you want them, you must manually add them when you edit your code.

What is the Free Software Foundation?
These tools exist because of the Free Software Foundation. They provide the source code to many useful programs, including the GNU utilities upon which these tools were based. (GNU is short for "GNU's Not UNIX.") Note that any modifications to their code must be made freely available under the same terms. You can contact them via e-mail at gnu@prep.ai.mit.edu.

BEGGING TO DIFFER

These tools grew out of our frustration with existing source code control systems. After using them for many years, we've found them to be indispensable to our development. They've served us well because they're simple, fast, and easy to adapt to new situations. We hope that you'll enjoy using them as much as we do — and if you don't, perhaps you'll improve them.

Thanks to Helen Casabona, Pete Gontier, Andy Jeffrey, and Tim Maroney for reviewing this column.

***develop* welcomes guest columns** on a variety of subjects. Please submit your column draft or idea to develop@apple.com.

MacApp Debugging Aids

While working on Twist Down Lists, a recordable MacApp implementation of hierarchical lists, I developed several useful debugging aids for detecting memory leaks and access faults and managing memory usage problems. Here I describe how to use these debugging aids for more trouble-free MacApp programming.

CONRAD KOPALA

In the article "Displaying Hierarchical Lists" in *develop* Issue 18, Martin Minow suggests that MacApp offers "flexible libraries for displaying and managing structured data." I accepted his challenge and decided to create a Twist Down Lists application with MacApp version 3.3.1. As complete as MacApp is, you still have to test your application to make sure it works. Among the problems I encountered, perhaps none were more frustrating than the insidious memory leak and the dreaded access fault. After discovering the *n*th memory leak and the *m*th access fault in my Twist Down Lists application, I decided that the situation was unacceptable — there had to be a better way!

To solve these problems, I developed several debugging techniques. These techniques were useful to me, so I decided to share them with you in this article. Here are some of them:

- *Object counting* lets you quickly discover memory leaks.
- *Memory display* helps you gauge the size of a memory leak.
- *Object display* helps you identify the cause of a memory leak and an access fault.
- *Object heap discipline* helps your application manage tight memory situations by allowing you to erect a barrier to further expansion of the object heap.
- *Failure handling* lets you force a failure in any spot in your code.

Accompanying this article on this issue's CD and *develop*'s Web site is the complete Twist Down Lists application, which you can look at to see the implementation of all the debugging aids described in this article. Also provided are two engineering notes, "EN1 – Object Counting and Display" and "EN2 – Object Heap Discipline," which go into the gory details of implementing these debugging aids, and copies of the four MacApp files UObject.h, UObject.cp, PlatformMemory.h, and PlatformMemory.cp, which I modified to incorporate the debugging aids and which you can substitute for the original files (or similarly modify them yourself).

CONRAD KOPALA (ckopala@aol.com) believes you should never trust a computer you can't program. He's been a student of MacApp for the last six years and just recently thinks he might know a smidgen about it. In the past, Conrad was an electrical engineering professor and held positions with IBM and MCI. Now he does whatever he wants.

Most of these debugging techniques are specifically for MacApp version 3.3.1. Later versions may already incorporate similar debugging features.•

OBJECT COUNTING

I've found object counting to be the fastest way to discover memory leaks. To maintain a running count of the number of objects in existence, I use a global variable named gObjectCount. Whenever a TObject is created or cloned, gObjectCount is incremented; when a TObject is destroyed, gObjectCount is decremented. The variable is incremented in the TObject constructor or TObject::ShallowClone and is decremented in the TObject destructor.

To print the current value of gObjectCount, I use a global function named PrintObjectCount. You can call this function at any point in the application where you think it's useful. In my experience, one of the best places to test the value of gObjectCount is at the beginning of the function TYourApplication::DoSetupMenus. That point represents a set of stable application states that you should always be able to return to. By monitoring the value of gObjectCount as the application runs, you can obtain a set of characteristic values for gObjectCount. Any variation in these values should be investigated as a possible memory leak.

For example, for Twist Down Lists, the object count just after startup is 49. After a twistDownDocument is opened and closed, this count increases to 52. This increase is a consequence of adding a print handler to the view; a TPrintInfo and two TDependencies objects are created but never freed. Then, if you change the font size by choosing the Other menu item, the object count increases to 55. In this case, the TDialogTEView, TAdornerList, and TScroller objects are created when a new font size is entered in the TNumberText; they're never freed. Thereafter, the quiescent value of gObjectCount remains unchanged.

By using object counting, I've discovered TObject-based memory leaks in just minutes. To implement it, you need to make changes to UObject.h and UObject.cp, as described in "EN1 – Object Counting and Display." Or you can include the substitute UObject.h and UObject.cp files that I've provided.

MEMORY DISPLAY

My global function DisplayMemoryInfo displays the amount of free memory, the size of the temporary reserve, the size of the permanent reserve, the object heap size, the amount of memory available in the object heap, and the amount of object heap space used. If you have a memory leak, this function can give you information about the size of the leak. As with object counting, you can get a set of characteristic values as you run the application. The most useful of these indicators is the amount of object heap space used. In my experience, it makes the most sense to call this function at the beginning of the function TYourApplication::DoSetupMenus when you also display the object count.

Realize that each time the object heap is expanded, an overhead of 20 bytes is incurred. As a result, the amount of object heap space slowly increases until the object heap reaches its maximum size. So if you see the amount of space used in the object heap increasing by some multiple of 20, it might just be attributable to object heap overhead.

To implement memory display, you need to make changes to UObject.h and UObject.cp, as described in "EN1 – Object Counting and Display." Or you can include the substitute UObject.h and UObject.cp files that I've provided.

OBJECT DISPLAY

While object counting and memory display let you quickly discover a memory leak, it's object display that helps you to identify the cause of the memory leak or an access fault. Turning on object display means that when a TObject-based object is constructed, a message — including "who, what, and where" — appears in the debugging window. Likewise, when the object is destroyed, a similar message appears.

You can use a Simple Input-Output Window (SIOW) instead of your debugger's log window to display this information if you prefer.•

When an object is created, if object display is on, the debugger log window displays a message similar to the following:

```
Construct TSomeMacAppObject @0x2D6ACA4 Id=74 Size=108 ObjCnt = 73
#Construct TMyObject @0x2D6ACA4 Id=74 Size=108 ObjCnt = 73
```

When the object is destroyed, the log window displays a message like this:

```
#Destruct TMyObject @0x2D6ACA4 Id=74 Size=108 ObjCnt = 73
Destruct TSomeMacAppObject @0x2D6ACA4 Id=74 Size=108 ObjCnt = 73
```

Each line gives the class name of the object, its location in the object heap, its class ID, its size in bytes, and the current value of gObjectCount. In addition, the message tells you whether the object was created or destroyed.

So why are there two lines for construction and destruction? When an object like TMyObject is created, its TObject constructor is executed first, followed by the constructors for any MacApp objects in the descendant chain, ending with the constructor for TMyObject. In other words, objects are built from the bottom up. As each constructor does its thing, it's given the chance to display a message identifying itself. So when a new object is created, a series of messages is displayed that identify each stage of the construction process.

When the object TMyObject is destroyed, the process is reversed, with the destructor for TMyObject first displaying a message identifying itself, followed by the destructors for any MacApp objects in the descendant chain and ending with the destructor for TObject. Objects are destroyed from the top down.

Running an application with object display on provides a wealth of information about what an application is doing — information that you can't get any other way. It's also a *great* way to find out what MacApp is doing. As described later in the section "Implementing Object Display," you can use flags to specify how much information to display.

DETECTING MEMORY LEAKS

Of course, when tracking down a memory leak, you're interested in finding an object that was created but never destroyed. To find this object, it's necessary to match object destructions with constructions. The leftover construction is the offending object that wasn't destroyed. You match constructions and destructions by using the addresses provided in the object display.

Be careful when matching object destructions and constructions, because MacApp will reuse space in the object heap. I've often seen MacApp make a TAppleEvent, shortly thereafter free it, and then go on to make another TAppleEvent and store it at exactly the same address.•

If your debugger allows you to save the contents of the log window, sorting it on the address field would bunch all items with the same address together. That would make it much easier to match destructions with constructions. If you assign each object a serial number in its constructor, it would be even easier to do the matching.

Consider a real example. The MacApp example application IconEdit has a memory leak. (I found the leak because I used the application as a template.) Listing 1 shows the offending code.

Listing 1. An example of a memory leak

```
void TIconDocument::DoMenuCommand (CommandNumber aCommandNumber)
{
    switch (aCommandNumber) {
        case cSetColor:
            {
                CRGBColor    newColor;
                CStr255      thePrompt = "Pick a new color";
                if (GetColor(kBestSystemLocation, thePrompt, fColor,
                        newColor)) {
                    if (TOSADispatcher::fgDispatcher->GetDefaultTarget()
                            ->IsRecordingOn()) {
                        TSetPropertyEvent *appleEvent = new TSetPropertyEvent;
                        appleEvent->ISetPropertyEvent(gServerAddress,
                                        kAENoReply, this, pColor);
                        CTempDesc theNewColor;
                        theNewColor.PutRGBColor(newColor);
                        appleEvent->WriteParameter(keyAEData, theNewColor);
                        appleEvent->Send();  // <-- the problem
                    }
                    else {
                        TSetColorCommand *aSetColorCommand =
                                            new TSetColorCommand();
                        aSetColorCommand->ISetColorCommand(this, newColor);
                        PostCommand(aSetColorCommand);
                    }
                }
            }
            break;
        default:
            Inherited::DoMenuCommand(aCommandNumber);
            break;
    }
}
```

With object counting and display, it took only minutes to discover the leak and identify the offending objects. Deciding how to eliminate the leak took a little longer. The leak arises because TAppleEvent::Send returns a reply TAppleEvent and neither it nor the TAppleEvent that was sent is freed. This leak is fixed by using the code snippet

```
TAppleEvent * theReply = theEvent->Send();
FreeIfObject(theEvent);
FreeIfObject(theReply);
```

in place of

```
appl eEvent->Send();
```

Listing 1 is an example of a small memory leak, only 64 bytes. Because of its small size, it's virtually undetectable by means other than object display. These small memory leaks are a very serious problem because they fragment the object heap. Suppose that every time a command is executed, a 64-byte memory leak is created and they're uniformly distributed across the object heap. Now suppose the application needs to create an object that's too big to fit in any of the available gaps in the object heap. Under these circumstances, the application would come to a grinding halt and the only thing the user could do is quit and restart the application (if the computer doesn't crash).

DETECTING ACCESS FAULTS

Discovering access faults is easy. The Power Mac Debugger loudly, almost proudly, proclaims, "Access Fault." If luck is with you, your machine doesn't crash or lock up. Identifying the cause of an access fault is another matter.

If the access fault involves a TObject-based object, that means the application attempted to access an object that doesn't exist. There are two ways that can happen: Perhaps the object was created, then destroyed, and now the application attempts to access it. Or maybe it was never created in the first place.

Object display can help you identify the offending object by providing an ordered record of what was created and what was destroyed. If you've been testing with object display on, you will have become familiar with what your application is doing. Then the trick is to single step up to the point of the access fault without failing. At that point, you should know which object the application is attempting to access. You can carefully examine the results of the object display to determine the source of the problem.

Access fault of the first kind. One type of access fault, which I'll call an access fault of the first kind, arises from creating a TObject-based object, freeing it, and then attempting to access it. Because it was freed, it no longer exists, so attempting to access it causes an access fault.

When I was first teaching myself about MacApp's scripting capability, I made the mistake of taking some MacApp code out of context and using it as a template in Twist Down Lists. It was clearly the wrong thing to do because it resulted in an access fault of the first kind. My mistake is illustrated by the following code, which I wrote in TTwistDownApp::GetContainedObject. I show it here so that you can try it and see for yourself how object display helps you find the first type of access fault.

```
TTwistDownDocument*  theTwistDownDocument  = NULL;
theTwistDownDocument  = (TTwistDownDocument*) aDocument;
theTwistDownView = theTwistDownDocument->fTwistDownView;
TOSADispatcher::gDispatcher->AddTemporaryToken(theTwistDownView);
result  = theTwistDownView;
return result;
```

Of course, in due time, MacApp freed the temporary token and, later on when the application attempted to access fTwistDownView, an access fault was generated. Running the application with object display on clearly showed twistDownView being destroyed: you can't miss it and you know it's wrong. Then, a little bit of single

stepping led me to the culprit. I recognized that I shouldn't have told fgDispatcher to add fTwistDownView as a temporary token. I fixed this by deleting the statement that tells fgDispatcher to add it, and then carried on.

Access fault of the second kind. Another kind of access fault, which I'll call an access fault of the second kind, arises from attempting to access a TObject-based object that was never created. The only access fault of this kind that I've encountered arose when I ran a script that asked the application to access a document when there were no documents. In this case, TApplication::GetContainedObject doesn't verify that the document exists before attempting to use it.

For this situation, the problem was immediately obvious. It was easily fixed by inserting into TTwistDownApp::GetContainedObject the code shown in Listing 2, which makes sure the document exists before attempting to access it.

Listing 2. A solution to the GetContainedObject problem

```
...
else if (desiredType == cDocument && selectionForm == formName) {
    CStr255 theName;
    selectionData.GetString(theName);
    CNoGhostDocsIterator iter(this);

    for (TDocument* aDocument = iter.FirstDocument(); iter.More();
            aDocument = iter.NextDocument()) {
        if (aDocument != NULL) {
            CStr255 name = gEmptyString;
            aDocument->GetTitle(name);
            if (name == theName) {
                theTwistDownDocument = (TTwistDownDocument*)aDocument;
                result = theTwistDownDocument;
                return result;
            }
        }
    }
}
```

This case demonstrates the wisdom of trying to break your application by attempting to get it to do outlandish things that no sane person would try. That's precisely how I stumbled on this one.

Access fault of the third kind. All other access faults I've defined as access faults of the third kind: they are, strictly speaking, outside the scope of MacApp. They arise from mistakes you made when working with the system software — for example, failing to clear a parameter block before using it. As a result, object display isn't quite as helpful at tracking down these access faults as it is with finding access faults involving TObject-based objects. If you're lucky, object display will point you in the general area of the problem.

The MacApp application IconEdit gives us an example. Along with the other files that accompany this article, I've provided a test script, a modified version of the IconEdit source code, and many IconEdit documents to help you conduct the following experiment:

1. Set the partition size of IconEdit to its minimum of 1506.
2. Make and save about 25 IconEdit documents.
3. Quit IconEdit to quickly get rid of all the open documents.
4. Make a script that tells IconEdit to open all the saved documents.
5. Run the script.

When IconEdit runs out of memory while being driven by the script, it will generate an access fault of the third kind and drop into MacsBug with a bus error. This should occur after the 22nd document has been opened and the script is telling IconEdit to open the 24th saved document. The 23rd document has failed to open for lack of memory, and the application is attempting to recover, yet the script has gone beyond that point and is telling the application to open the 24th document. (Note that you'd need to use a lot more documents to duplicate this condition if you didn't compile with the substitute PlatformMemory files, which implement object heap discipline, as described later.)

The problem occurs when the application attempts to return an out-of-memory message to the script. Specifically, TAppleEvent::WriteLong generates another error when it attempts to complete the reply Apple event that's supposed to tell the script about the out-of-memory condition.

This is a case where the techniques described in this article don't help you very much and may actually hinder you. I didn't do anything wrong, but I did spend time proving that I didn't do anything wrong. Once I got assurance that all the TObject-descended objects in my application appeared to be OK, I decided to see if IconEdit had the same problem. It did.

In due time, I noticed that the TServerCommand constructor sets fSuspendTheEvent to FALSE. TServerCommand is an ancestor of TODocCommand, which is responsible for opening existing documents. At that point, I had nothing to lose by setting its value to TRUE. That fixed the problem.

Be warned that the experts will tell you that fSuspendTheEvent should never be set to TRUE because doing so can be dangerous. I've disregarded their advice with no better rationale than that it appears to allow IconEdit and Twist Down Lists to survive without crashing when I run a script that sends **open document** commands until the application fails for lack of memory. As of this writing I haven't found a more acceptable workaround.•

IMPLEMENTING OBJECT DISPLAY

To implement object display, you can plug in the substitute UObject.h and UObject.cp files. (Implementation details are provided in "EN1 – Object Counting and Display.") In addition, four new methods need to be added to TObject:

```
#if qDebug
void TObject::PrintConstructorClassInfo();
void TObject::PrintDestructorClassInfo();
void TObject::PrintAppConstructorClassInfo();
void TObject::PrintAppDestructorClassInfo();
#endif
```

The constructors and destructors of all objects for which you want to be able to display object information must be modified to call the appropriate method. For MacApp objects, use the following code in the constructor:

```
#if qDebug
this->PrintConstructorClassInfo();
#endif
```

and this code in the destructor:

```
#if qDebug
this->PrintDestructorClassInfo();
#endif
```

You may not want to call these methods in TEvent and TToolboxEvent. The Macintosh specializes in generating events, so displaying object information for them can be overwhelming. In your application objects, use the following code in the constructor:

```
#if qDebug
this->PrintAppConstructorClassInfo();
#endif
```

and this code in the destructor:

```
#if qDebug
this->PrintAppDestructorClassInfo();
#endif
```

These calls should always be placed in the same relative position in constructors and destructors. The very beginning or the very end are the two most obvious choices. Keep in mind that although constructors don't generally make other objects, destructors frequently free other objects. If these methods are invoked at random places in the constructors and destructors, the resulting object information displayed in the log window will be very hard to interpret.

There are three flags that you can use to control the amount of object information that's displayed: gPrintBaseClassInfo, gPrintMacAppClassInfo, and gPrintAppClassInfo. These flags determine whether object information is displayed at the TObject level, for MacApp objects, or for your application's objects, respectively. All three flags can be set with scriptable menu commands. However, it's probably best to set gPrintBaseClassInfo programmatically to avoid being inundated with object information for every TToolboxEvent that's generated. Simply surround the suspect code as follows:

```
gPrintBaseClassInfo = TRUE;
...    // suspect code here
gPrintBaseClassInfo = FALSE;
```

In my experience, it's usually enough to display object information at the application level and the MacApp level. However, some MacApp objects don't have constructors and some don't have destructors. If you suspect that those objects are the source of a problem, it may be useful to display object information at the TObject level.

OBJECT HEAP DISCIPLINE

MacApp uses its own object heap to store TObject-derived objects. The heap grows as new objects are created — by taking memory from free memory. Once memory has been allocated to the object heap, it's never returned to free memory. As things stand, the developer has little control over this situation.

According to conventional wisdom, the point of greatest memory usage occurs during printing. With Twist Down Lists, memory usage problems occur when the application runs out of memory while loading a hierarchical list, especially with 680x0 versions. That meant I had two problems to deal with while testing recovery from an out-of-memory condition when loading a list: the recovery itself and the lack of available memory in which to load required code segments. I made the segment loading problem go away by implementing object heap discipline, which let me concentrate on testing failure recovery. Object heap discipline allows you to erect a barrier to further expansion of the object heap right where you want it. At the same time, it allows you to leave as much memory as is required to load code segments without having to fuss with 'seg!' and 'res!' resources.

When the object heap runs out of space, a request for a new block of memory is made with a call to the global function PlatformAllocateBlock(size_t size). The trick is to force PlatformAllocateBlock to reject the request when you want it to.

To do that, I created the global down-counter gOHRemainingIncrements to maintain a count of the number of times the object heap will be allowed to expand. Each time PlatformAllocateBlock allocates memory to the object heap, it decrements gOHRemainingIncrements. When gOHRemainingIncrements reaches 0, PlatformAllocateBlock will no longer honor requests for additional memory. The revised version of PlatformAllocateBlock is shown in Listing 3.

Listing 3. Revised PlatformAllocateBlock

```
void *PlatformAllocateBlock(size_t size)
{
    Boolean  heapPerm;

    if (gUMemoryInitialized)
        heapPerm = PermAllocation(TRUE);

    void *ptr = NULL;                       // added
    // void *ptr = NewPtr(size);            // commented out
    if (gOHRemainingIncrements > 0) {       // added
        ptr = NewPtr(size);                 // added
        gOHRemainingIncrements--;           // added
    }                                       // added

    if (gUMemoryInitialized)
        PermAllocation(heapPerm);           // Reset perm flag before
                                            // possible Failure
    FailNIL(ptr);

    return ptr;
}
```

The initial value of gOHRemainingIncrements is set to 3 just to be safe. During initialization, MacApp makes two allocations to the object heap; if gOHRemainingIncrements is 0, the application doesn't start up because of lack of memory. The second of those allocations sets up the initial size of the object heap. If your 'mem!' resource specifies a small value for the initial size of the object heap, the initial value of gOHRemainingIncrements might have to be larger than 3.

Immediately following the call to InitUMacApp in **main**, the value of gOHRemainingIncrements is set with a call to the InitMaxObjectHeapSize global function, which is shown in Listing 4.

```
gOHRemainingIncrements = InitMaxObjectHeapSize();
```

Listing 4. Determining the number of times the object heap will be allowed to expand

```
short InitMaxObjectHeapSize()
{
    long  freeMem = FreeMem();
    Size  heapSizeIncrement = gSizeHeapIncrement;
    short theNumber = 0;

    if (freeMem > kFreeMemReserve)
        theNumber = (freeMem - kFreeMemReserve) / heapSizeIncrement;
    if (theNumber >= 1)
        theNumber = theNumber - 1;
    else
        theNumber = 0;

    return theNumber;     // The number of times we'll let the object heap
                          // be expanded
}
```

As you can see, I use a very simple algorithm to determine the number of times the object heap will be allowed to expand and still leave in free memory at least the number of bytes specified by kFreeMemReserve.

The changes you need to make to MacApp to implement object heap discipline are described in detail in "EN2 – Object Heap Discipline." The substitute PlatformMemory.h and PlatformMemory.cp files are also provided.

FAILURE HANDLING

One very good reason to use MacApp is its integrated failure handling scheme. Of course, all failure recovery paths must be tested. In a well-crafted application, failures should occur only while the application is attempting to create a new object when there's insufficient space in the object heap for it. To test these situations, the application must be forced to fail at selected points. It's not enough to adjust the partition size and hope for a failure.

Ideally, you would be able to set a failure point with a debugger in a similar manner to setting a breakpoint. That's not presently possible. Instead, in Twist Down Lists, I added a global flag gFailHere, which is set and cleared with a scriptable menu command. There are several ways to use this flag:

- Insert the following code at an appropriate place in the application (this is the simplest way):

  ```
  if (gFailHere)
      Failure(errFailHere, 0);
  ```

- Force a failure just after a new object has been created:

```
TSomeObject*  someObject  = new TSomeObject;
FailNIL(someObject);
someObject->ISomeObject();
if  (gFailHere)
    Failure(errFailHere,  0);
```

- Force the failure in ISomeObject:

```
TSomeObject::ISomeObject()
{
    this->IObject();
    if  (gFailHere)
        Failure(errFailHere,  0);
}
```

Other ways of using this technique to force a failure require application-specific knowledge. In the case of Twist Down Lists, it's often convenient to give the name FailHere to a file or folder on the volume you're going to open. With the following code, when a twistDownElement named FailHere is encountered, the failure will be triggered:

```
if  (gFailHere)  {
    CStr63  failHereText  = "FailHere";
    CStr63  displayedText  = gEmptyString;
    twistDownElement->GetDisplayedText(displayedText);
    if  (failHereText  == displayedText)
        Failure(errFailHere,  0);
}
```

It must be possible to set and clear the gFailHere flag from a script. An application can encounter the same failure conditions whether driven from a script or from the user interface. The failure recovery path is, however, a little different. When the application is being driven by a script, an Apple event must be sent to the script telling it that a failure was encountered and what the failure was. MacApp will handle the overhead, but you must do your part: you must test it to make sure it works and returns appropriate error information to the script.

Were it not for the fact that all the list processing methods in TTwistDownDocument are recursive, I probably wouldn't have felt the need to implement gFailHere. Failure can occur if the application attempts to make a twistDownElement or a twistDownControl when there isn't enough memory in the object heap to do it and the object heap can't be further expanded. The failure might occur several levels into the recursion. You can't call ReSignal to handle the failure because you'll jump all the way up to the method that started the recursion. Instead, you must save the failure information, work your way back up the recursion, and then signal the failure.

Using gFailHere turned out to be the best way to test the failure handling. Object counting, memory display, and object display were very useful in testing recovery from these kinds of failures. Object counting and memory display verified that everything that needed to be freed was freed. Using object display to match constructions with destructions gave further confirmation that the recovery was successful.

HAPPY DEBUGGING

The debugging aids I developed illustrate the power of MacApp. By modifying the central organizing object, TObject, you can make many new capabilities, such as object counting, memory display, and object display, extend to the objects that inherit

from it. In addition, you can easily modify the memory management scheme of MacApp, so implementing object heap discipline isn't hard at all.

Now that I have these debugging aids, I no longer fear the dreaded memory leak and access fault. Object counting, memory display, and object display don't exactly sound an alarm when there's a TObject-based memory leak, but they come pretty close. And without object display, finding an access fault was like looking for a needle in a haystack.

The faster you can fix your mistakes, the faster you can finish your applications. I hope my debugging aids will help you get those applications out even quicker.

RELATED READING

- *Programmer's Guide to MacApp* (Apple Computer, Inc., 1996). Available on the Web at http://www.devtools.apple.com/macapp.
- "Displaying Hierarchical Lists" by Martin Minow, *develop* Issue 18, and "An Object-Oriented Approach to Hierarchical Lists" by Jan Bruyndonckx, *develop* Issue 21.
- "A Reassuring Progress Indicator for MacApp" by James Plamondon, *FrameWorks* Volume 5, Number 3, June 1991, page 46.

Thanks to our technical reviewers Tom Becker, Geoff Clapp, Mike Rossetti, Merwyn Welcome, and Jason Yeo.•

MPW TIPS AND TRICKS

Automated Editing With StreamEdit

TIM MARONEY

In this column in Issue 26 of *develop*, I showed you a wide range of scriptable editing commands available from the MPW Shell. This time I'll discuss a single tool that provides a powerful self-contained text-editing scripting language, StreamEdit.

Why would you want to use StreamEdit instead of the other text-editing features of the MPW Shell?

- Performance — A StreamEdit script is faster than an MPW script containing various Replace and Find commands.
- Self-containment — Because StreamEdit is a self-contained tool, you can run it from within ToolServer, unlike the scriptable editing commands discussed in Issue 26, which are available only in the MPW Shell itself. This means you can use StreamEdit to create lightweight drag-and-drop grinder AppleScript scripts that send StreamEdit commands to ToolServer.
- Consistency — Keeping all your editing in a single scripting language confers the elusive mystical boon of code consistency, making your system easier to maintain and modify in the future.

GETTING TO KNOW YOU

StreamEdit is based very closely on the hoary UNIX tool named **sed**. If you already know **sed**, much of this will be familiar, but StreamEdit isn't directly compatible with **sed** scripts.

An MPW version of sed is available as part of the GNU free software project. One place you can find it is http:/ / sunsite.cnam.fr/ packages/ gnu/ cygnus/ mac/ .•

StreamEdit implements a pattern-matching language. Every time a particular pattern is matched, a sequence of commands will be executed. As in most pattern-matching languages, StreamEdit's scripts are lists of pattern/command pairs, with the pattern coming before the command. The input file or files are read through the script interpreter, which searches for instances of the patterns and executes the corresponding commands. Anything that doesn't match a pattern is passed through unchanged.

StreamEdit scans one line at a time through the input, matching its current line to every pattern in its script. After processing each line, it writes out the modified line. The result is a concatenation of three internal buffers: the *insert buffer*, then the *edit buffer*, and finally the *append buffer*. The edit buffer gets filled with the current line, while the other buffers are empty at the start. The Insert and Append commands place text in the insert and append buffers, allowing you to add text to the beginning and end of the output line. The Change, Delete, and Replace commands modify the contents of the edit buffer.

SHARING ADDRESSES

As usual, MPW uses words in ways previously unknown in human speech. In StreamEdit, patterns are referred to as "addresses." There are two kinds of addresses: line numbers and regular expressions. Line numbers ought to be self-explanatory, but it may help to note that the numbers must be Arabic numerals rather than Roman, and must be in base 10 rather than the hexadecimal or sexagesimal number systems. There are three special line numbers:

- the bullet symbol (•, Option-8), meaning the point before the first line (enabling you to add a line before the first line, for example)
- the infinity symbol (∞, Option-5), meaning the point after the last line
- dollar sign ($), meaning the last line

The keyboard shortcuts, as always in this column, are for American QWERTY keyboards; if you've got some other type of keyboard, you're on your own.•

Regular expressions are expressions that manage their diets sensibly. They can be used for searching, and were explained in detail in Issue 26. In StreamEdit addresses, though, regular expressions find the entire line containing the pattern, rather than just the pattern. Regular expressions are denoted by slashes. Only forward slashes are used (StreamEdit doesn't have a

TIM MARONEY has appeared professionally in newspapers, magazines, compact discs, videotape, and of course, computer software. Tim is a technical lead in human interface software at Apple and is editing a series of books for a horror publisher. His skin burns easily in the sun and tans in the moon. He uses white T-shirts only for house painting and car repair.•

backward search mode, having been frightened at an early age by the legends of Eurydice and Lot's wife). Three new constructs have been added to regular expressions in StreamEdit:

- **ç** (Option-C), which indicates a case-sensitive search
- // (two slashes), which means the last regular expression that was matched
- ≤*variable*≥ (a variable name embedded in inequality operators, here overloaded as a special kind of angle brackets, and typed as Option-comma and Option-period), which means the text of an expanded StreamEdit variable, treated as literal text to be matched rather than as a regular expression

StreamEdit has variables that can be set with the Set command (more on this later) or from the command line using the **-set** *variable* [**=***value*] option.

You can form more complex addresses using a few operators. The Boolean and, or, and not operators are the same as in C (**&&**, **||**, and **!**, respectively). Parentheses can be used for grouping within addresses. The comma operator matches the range of lines specified; for example, **3,5** matches lines 3 through 5. A range address matches each of the lines in the range, if any. It can be thought of as matching more than once: it fires off the accompanying command on the first line matched, the last one matched, and all lines in between. If the termination condition is never met, the address continues to match until the end of input. This could happen if you specify a range of lines ending at line 15, for instance, and there are only ten lines in the file, or if your range termination condition is a regular expression that doesn't appear anywhere in the input.

TAKING ACTION

Matching patterns is very nice, but what do you do once you match them? Statements in StreamEdit attach actions to patterns. An action consists of one or more commands, separated by semicolons or by the end of a line. There's no begin or end bracketing as in Pascal or C. Addresses and commands are syntactically distinct, so the script interpreter can figure out where the list of commands for a pattern ends and the next pattern begins.

Editing commands

- **Insert** *text* [**-n**] — Adds the specified text to the start of the line by putting it in the insert buffer. The **-n** option (in this command and in Append and Change) prevents adding a newline character when the line is written out.
- **Append** *text* [**-n**] — Adds the specified text to the end of the line by putting it in the append buffer.
- **Change** *text* [**-n**] — Changes the line to the specified text by replacing the contents of the edit buffer.
- **Delete** — Clears the edit buffer.
- **Replace** [**-c** *count*] /*pattern*/ *text* — Replaces the pattern with the specified text. This is the second part of a two-step matching process: first the address matches a line, then Replace searches in the edit buffer and replaces the pattern. The count argument indicates the maximum number of times to perform the replacement in the line. It can be a positive integer or infinity (∞). The default count is 1.

Control commands

- **Exit** [*status*] — Stops StreamEdit with the given error status. The default is 0, which means execution completed with no errors. Any nonzero error status indicates a problem, and unless the built-in MPW variable Exit is set to something other than 0, this will stop execution of the script (if any) from which the StreamEdit command was executed.
- **Next** — Somewhat like the C keyword **continue**. When a Next command is executed, all pending changes are written out and no more addresses are matched against the current line; that is, StreamEdit immediately goes on to the next line without matching the rest of the rules against the current edit buffer.
- **Set** *variable text* [**-i** | **-a**] — Much like the MPW Shell Set command. The variable is set to the specified text. The **-i** and **-a** options allow text to be added to any existing setting of the variable at the start or the end, respectively.

Output commands

- **Print** [*text*] [**-appendto** | **-to** *file*] — Writes output to a specified file. If *text* is empty, the current line is printed without modification. The **-appendto** and **-to** options write at the end of the file or overwrite the file, respectively. If no file is specified, standard output is used. If the filename is empty, nothing gets printed.
- **Option AutoDelete** — Deletes all input lines, leaving only output from Next and Print commands. You can get the same effect by specifying the **-d** option on the StreamEdit command line or by including this in the script:

```
/≈/     Delete
```

The text arguments to these commands are usually literal text, denoted by single or double quotes. There are a few other forms as well:

- An unquoted variable name can be used, in which case the variable is expanded; no brackets need be (or even may be) supplied.
- A period means the current input line up to but not including the newline character at the end.
- As discussed in Issue 26, you can use ® (Option-R) followed by a digit to mean the expression with that number matched in the pattern.
- You can read text from a file with **-from** *filename*, which reads the next line of text from the specified file. The filename is usually literal text, but it could also be a variable, the current input line (denoted by a period), or a ® expression.

A HYPOTHETICAL EXAMPLE

Let's say you're the director of corporate communications at a major computer maker and, without any warning except for inventory backlogs larger than the gross national products of many developing countries, you experience a sudden transition in chief executive officers, corporate policy, and product line. Your quarterly report (10-Q) is due in the SEC's EDGAR database tomorrow. Fortunately the SEC requires the cutting-edge ASCII format for its filings, and you realize that you can automate 90% of the tedious changes with a single StreamEdit script.

```
# Change nickname of CEO
/Diesel/
Replace // 'Flyboy'

# Change corporate policy
/1,$/
Replace /capture market share/ 'survive'

# Remove lines referring to obsolete products
/PowerTalk/ ll /eWorld/
Delete

# Change developer relations strategy
/third-party developers/
Replace /evangelize/ 'listen to'

# Mark lines referring to old schedules with a
# distinctive string at the start of the line
# for manual editing later
/1996/
Insert 'WHOOPS: '

# Add new final line of report
∞
Append 'May God have mercy on our souls.'
```

CONTROL YOURSELF

StreamEdit is almost too powerful. People have used it for everything, including pretty-printing source code, converting files to HTML, and postprocessing object files for dynamic linking tools. If you use it for finding incriminating passages in coworkers' e-mail, karma may get you, but the limitations of the tool won't. Use your powers for good rather than evil, and a grateful world will thank you.

Thanks to Arno Gourdol, Alex McKale, and Robert Ulrich for reviewing this column.

Chiropractic for Your Misaligned Data

Misalignment occurs when a program accesses data in a way that's not in sync with the processor's internal paths. This can slow down performance a little or a lot, depending on the CPU architecture. But finding these areas in code can be very difficult. We'll demonstrate the cause and cost of alignment problems and then show you a couple of tools you can use to detect them in your programs.

KEVIN LOONEY AND CRAIG ANDERSON

Sometimes Macintosh application performance is limited by architectural factors that can't be remedied, like the raw speed of the I/O or memory subsystem. But the programmer *does* have control over some factors that affect the speed of the memory subsystem and thus application performance — such as how data is aligned and accessed within memory. By default, most compilers will do the appropriate alignment for PowerPC™ code. However, alignment options offered for backward compatibility with the 680x0 architecture can cause significant overhead.

Misalignment is a difficult performance problem to detect. Traditional debugging and performance tools typically don't help you find misaligned accesses. On top of this, misalignment problems manifest themselves differently on different CPU architectures.

In this article, we'll define misalignment, describe how it's caused, discuss the overhead penalties for accessing misaligned data on various microprocessors, and introduce some tools designed to aid in the detection of misaligned accesses in code. These tools accompany the article on this issue's CD and *develop*'s Web site. Armed with these tools and what you learn in this article, you can perform chiropractic adjustments on your programs to solve their data alignment problems.

WHAT IS MISALIGNMENT AND WHY SHOULD I CARE?

A piece of data is properly aligned when it resides at a memory address that a processor can access efficiently. If it doesn't reside at such an address, it's said to be

KEVIN LOONEY (looney@apple.com) is a research scientist for Apple's Performance Evaluation Group, which does performance analysis of applications, systems, and hardware. He previously wrote performance and debugging tools as part of the Core Tools Group at Apple. Outside the confines of Apple, Kevin can be found moonlighting as a pianist/ synthesist and Web designer. He mainly ponders two questions in life: why are things taking so long, and what would cause someone with a degree in artificial intelligence to study performance issues?•

CRAIG ANDERSON (c.s.anderson@ieee.org) was formerly a senior performance analyst for Apple's Performance Evaluation Group. He's now at a startup company. Before working at Apple, he spent many years researching and writing his best-selling work *Improving the Performance of Bus-Based Multiprocessors*. Craig enjoys cooking with Mollie Katzen and praticing katas with sensei Huber. He also takes delight in reading fine literature, such as *The Gulag Archipelago, The Shipping News*, and *It's Obvious You Won't Survive by Your Wits Alone.*•

misaligned. In the PowerPC architecture, 32-bit and 64-bit floating-point numbers are misaligned when they reside at addresses not divisible by 4. Misalignment exceptions are taken based on the specific microprocessor.

Whether a data item is aligned depends not only on its address and the processor that's performing the access, but also on the size of the item. In general, data of size s is aligned if the least significant n bits of its address are 0, where $n = \log_2(s)$. Hence, 1-byte items are always aligned, while 2-byte items are aligned on even addresses and 4-byte items are aligned if the address is evenly divisible by 4. This alignment policy is often called *natural alignment* and is the recommended data alignment for code to run well on all current and future PowerPC processors.

Accessing misaligned data can be quite costly, depending on the microprocessor your program is running on. We'll demonstrate just how costly in a minute, but in general, misaligned memory accesses take from 2 to 80 times longer than aligned accesses on 603, 604, and future PowerPC microprocessors. A misaligned access can require more time to perform for two reasons:

- It may require two requests to the memory system instead of just one.
- It may cause the processor to take an unaligned access exception, a costly penalty.

WHAT CAUSES MISALIGNMENT?

Misaligned accesses can involve variables located on the stack or on the heap. The type of compiler and the compiler settings that you use will determine whether misaligned accesses occur. Improper structure placement and incorrect pointer arithmetic can also cause misaligned accesses. These problems can be found with the tools and techniques described below, but this article generally focuses on alignment problems that aren't caused by programmatic errors.

Most compilers for the Macintosh allow you to choose among various alignment options. Some compilers default to 2-byte alignment so that data alignment in PowerPC code mimics alignment on the 680x0 processor. While using this option means that structures written to disk in binary format can be accessed easily by both architectures, it also permits alignment problems in the PowerPC architecture. Both improper structure padding and misaligned stack parameters can result in misaligned accesses.

IMPROPER STRUCTURE PADDING

When a **float** field occurs in a structure, improper padding by the compiler will cause the **float** field to be misaligned. The example in Listing 1 uses MPW's alignment pragmas to illustrate this.

Listing 1. An example of a poorly aligned structure

```
#pragma options align=mac68k
typedef struct sPoorlyAlignedStruct {
    char        fCharField;
    float       fFloatField;
    char        fSecondCharField;
} sPoorlyAlignedStruct;

sPoorlyAlignedStruct    gPoorlyAlignedStruct;
#pragma options align=reset
```

In this example, a compiler that did no padding would align fFloatField on an offset of one byte from the structure's base address. Since compilers (and memory allocators) usually align the base of a structure on a boundary of at least four bytes (and multiples of four bytes), every access to fFloatField will cause a misalignment error. Also, fFloatField will be misaligned in statically or dynamically allocated arrays, since the lengths of structures are padded so that each structure that's an array element starts on a 4-byte aligned address.

A compiler with a 2-byte padding setting would align fFloatField on an even address, but this would still cause misalignment when that address isn't divisible by 4. Compilers using the **mac68k** pragma (as shown in Listing 1) cause 2-byte alignment, putting fFloatField on an even but often misaligned address for PowerPC processors.

A compiler with a 4-byte padding setting would always align the field properly.

MISALIGNED STACK PARAMETERS

Besides affecting the alignment of data in a structure, compiler settings can affect the way data structures are placed on the stack. Consider this function declaration:

```
void FunctionFoo (sPoorlyAlignedStruct firstParam, float floatParam)
```

In this example, the parameters are placed on the stack (even though PowerPC compilers use registers if possible). A compiler using a 680x0 2-byte padding option may align firstParam.fFloatField on an even address, but if the address isn't divisible by 4 this will cause a misalignment every time that parameter field is accessed within FunctionFoo. It won't, however, change the alignment of other parameters on the stack.

On the PowerPC processor, nonstructure parameters are usually placed in registers. There are no alignment problems when accessing registers.

THE COST OF MISALIGNMENTS

To demonstrate the cost of misalignments, we've written the code in Listing 2, which generates both aligned and misaligned accesses in the course of a million iterations. It accesses a byte array in different ways — data writes of integers, floats, and doubles — and at different offsets. In a portion of the code not shown, accesses are confined to within a single page of memory, and interrupts are turned off. Running this code enabled us to calculate the difference in performance between aligned and misaligned accesses. This code (with slight modifications for the various compilers) was compiled with the Symantec, MrC, and Metrowerks compilers. All compilers behaved similarly.

Table 1 shows the results. Overhead is calculated as the percentage difference between the time required for aligned and misaligned accesses. Our experiments showed that misaligned accesses at different offsets seemed to pay the same penalty (excluding cases where the two accesses required to retrieve the data straddle a memory page boundary, which is every 4K of memory).

Misaligned integer accesses result in a relatively small penalty for PowerPC 601, 603, and 604 CPUs. However, there's no guarantee that future microprocessors will provide hardware support for integer misalignment. Floating-point misalignments are severely penalized by the 604 implementation. In fact, while a misaligned access takes 1.5 times as long as an aligned access on the 601, it takes more than 80 times as long as an aligned access on the 604. The 601 pays a penalty only when an access crosses a page boundary (this will be verified later). Misaligned accesses to doubles on a 601 result in nearly double the overhead of misaligned accesses to floats. On the

Listing 2. Generating accesses for comparison of access time

```
#define kNumAccessesPerCycle  200
#define kNumCycles            5000
// Number of total accesses = kNumAccessesPerCycle * kNumCycles
// 1000000                  = 200                   * 5000
#define kTableSize   1608  // Table size needed for 200 separate aligned
                           // accesses on the largest data type (doubles)

typedef enum ECType { eLong, eFloat, eDouble };

void main(void)
{
   double AlignedTimeFloat = AlignLoop(0, eFloat);
   double Misaligned1TimeFloat = AlignLoop(1, eFloat);
   double Overhead1Float = (((Misaligned1TimeFloat - AlignedTimeFloat)
                              * 100) / AlignedTimeFloat);
   double avgOverhead1Float = (Misaligned1TimeFloat - AlignedTimeFloat)
                              / kNumTotalAccesses;
   ...
}

// The function AlignLoop measures the time of a loop of "writes" to a
// byte array. The writes are either aligned or misaligned, based on the
// offset parameter, which should be between 0 and 7. The type should be
// eLong, eFloat, or eDouble.
double AlignLoop(short offset, ECType type)
{
   UnsignedWide   startTime, stopTime;
   double         start, stop;
   char           bytetable[kTableSize];
   long           j, k;

   // Get starting timestamp.
   Microseconds(&startTime);

   switch (type) {
      case eLong:
         {
            long *longPtr = (long *) &bytetable[offset];
            for (j = 0; j < kNumCycles; j++)
               for (k = 0; k < kNumAccessesPerCycle; k++)
                  longPtr[k] = 1;
         }
         break;
      case eFloat:
         {
            float *floatPtr = (float *) &bytetable[offset];
            for (j = 0; j < kNumCycles; j++)
               for (k = 0; k < kNumAccessesPerCycle; k++)
                  floatPtr[k] = 1.0;
         }
         break;
```

(continued on next page)

Listing 2. Generating accesses for comparison of access time *(continued)*

```
        case eDouble:
            {
                double *doublePtr = (double *) &bytetable[offset];
                for (j = 0; j < kNumCycles; j++)
                    for (k = 0; k < kNumAccessesPerCycle; k++)
                        doublePtr[k] = 1.0;
            }
            break;
    }

    // Get ending timestamp.
    Microseconds(&stopTime);

    // Move the values to doubles.
    start = (((double) ULONG_MAX + 1) * startTime.hi) + startTime.lo;
    stop  = (((double) ULONG_MAX + 1) * stopTime.hi) + stopTime.lo;

    return stop - start;
}
```

Table 1. Misalignment overhead for basic data types, native PowerPC code

CPU and data	Aligned total access time (μsec)	Misaligned total access time (μsec)	Overhead
PowerPC 601 integers	113439	119573	5.4%
PowerPC 601 floats	63234	94505	50.0%
PowerPC 601 doubles	63251	113306	79.1%
PowerPC 604 integers	687	695	1.1%
PowerPC 604 floats	261	23753	9009.0%
PowerPC 604 doubles	262	22546	8509.5%

Note: Tests were run at 80 MHz on the PowerPC 601 and 132 MHz on the PowerPC 604.

604, however, doubles and floats suffer nearly the same overhead penalty. Misaligned accesses for doubles occur on any address not divisible by 8 on the 601, and any address not divisible by 4 on the 604. It's important to note that all memory accesses (aligned and misaligned) result in some timing penalty.

When we ran these experiments as emulated code (compiled for 680x0), float and double accesses showed no significant overhead (less than 8%). The 68040LC emulator doesn't do PowerPC floating-point loads/stores when processing floating-point data; it avoids alignment exceptions by doing integer emulation of a floating-point unit and loading and storing data 16 bits at a time.

Our code paints a worst-case scenario; worst case or not, the results indicate that there's plenty of motivation to avoid misaligned accesses in native PowerPC code. Perhaps the biggest problem facing the programmer, however, is the detection of these problems in application code. We'll look now at two tools that are useful for detecting and pinpointing alignment problems.

TOOLS FOR DETECTING MISALIGNMENTS

Apple's Performance Evaluation Group has developed two tools for detecting misalignments that cause exceptions:

- PPCInfoSampler, a tool to detect high levels of misalignment exceptions over general application workloads
- the Misalignment Instrument Library (MIL), a set of functions useful for profiling misalignments in specific regions of code

PPCInfoSampler is useful for determining whether your code has misalignment problems. If misaligned accesses are detected, the MIL can be used to pinpoint which parts of your code are causing the misalignments. We'll describe each tool in greater detail before discussing how to correct misalignments that you identify.

PPCINFOSAMPLER

PPCInfoSampler is a control panel that when activated records information about the PowerPC exception services and emulator at 100-millisecond intervals. The information recorded includes counts of mode switches, interrupts, misalignment exceptions, and page faults. See Table 2 for a list of PPCInfoSampler output categories. The output saved from PPCInfoSampler is in tab-delimited format and is best viewed from a spreadsheet program.

Table 2. PPCInfoSampler output categories

Output category	Explanation
Time Delta (millis)	Elapsed milliseconds since last sample of "exception services" registers
Microseconds time	Microseconds (calculated from timebase) since PPCInfoSampler was enabled
Timebase Ticks	A reading of the 64-bit timebase register
MixedMode switches	Number of mode switches into PowerPC code
Data Page Faults	Number of page faults
ExternalIntCount	Number of external processor interrupts
MisalignmentCount	Number of misaligned accesses that caused an exception
FPUReloadCount	Number of reloads of the FPU register state
DecrementerIntCount	Number of interrupts caused by the PowerPC decrementer register
EmulatedUnimpInstCount	Number of instructions that are emulated in exception services
Timebase Ticks 68k	Number of timebase ticks spent in 680x0 code
Timebase Ticks PPC	Number of timebase ticks spent in PowerPC code
Level n Int Ticks	Number of timebase ticks that expired per interrupt level
Level n interrupts	Number of interrupts that occurred at each interrupt level

Note: With the exception of microseconds time, each measurement is per sample and isn't cumulative with the next interval.

To use PPCInfoSampler, you must first drop it into your Control Panels folder and reboot. The tool installs code in the system heap that waits for an action to occur. There are two ways to activate the sampling mechanism:

- You can use the keyboard shortcut Command-Option-Z to start the sampler instantly at any time. When you start the sampler, an 8-pixel line will flash in the upper left corner of your main screen. It will continue to flash for each sample that's recorded. To stop the sampler, use the keyboard shortcut again.
- You can use the control panel interface, as shown in Figure 1, to start, stop, and save a sample.

For our purposes, let's focus on the number of misalignment exceptions. To determine in general whether you have misalignment problems in your application,

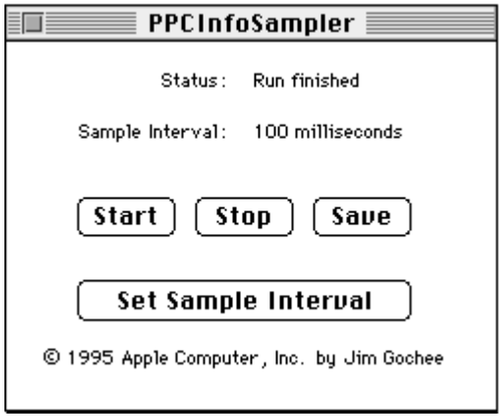


Figure 1. PPCInfoSampler control panel interface

think of the operations or "workloads" (such as saving to disk) that force your application to access many data structures. Run PPCInfoSampler during these workloads to determine whether any misalignments are occurring. Any misalignment count greater than 0 should be investigated and, if possible, corrected.

Table 3 shows an example of a misalignment count generated by PPCInfoSampler. The program that was being executed during this count displayed bursty misalignment characteristics. That is, during some 100-millisecond intervals no misalignments were happening; during other intervals, large numbers were happening.

Table 3. Sample output from PPCInfoSampler

Milliseconds	Misalignment count
100	0
100	0
100	11652
100	43694
100	42931
100	43695
100	43679
100	43705
100	42942
100	31213
100	0
100	0
100	0
100	14510
100	44135
100	44667
100	44470
100	44347
100	44323
100	44303
100	6416
100	0
100	0

THE MISALIGNMENT INSTRUMENT LIBRARY

Where PPCInfoSampler allows you to detect misalignment activity in broad 100-millisecond intervals, the Misalignment Instrument Library (MIL) allows you to then

make educated guesses about where to further instrument your code to pinpoint where the misalignment is happening. The MIL consists of two routines:

```
void initMsalignRegs(void);                 // Initializes our misalignment
                                            // counter (do this only once)
unsigned long getMsalignments(void);        // Returns the total number of
                                            // misalignment exceptions since
                                            // the initMsalignRegs call
```

With these calls to the MIL, you can profile strategic portions of your source for the application's workloads. Iteratively narrow the focus of your profile by moving the instrumentation in the code until you can determine where the misaligned accesses are and the structures that they're associated with.

Listing 3 is a sample of application code instrumented with the MIL calls. In this sample, we use the MIL to display the different floating-point exception-handling properties of the 601 and 604 CPUs.

Listing 3. Sample application code using the MIL

```
#define kNumAccessesPerCycle  200
#define kNumCycles            5000
#define kTableSize   804   // Table size needed for 200 separate aligned
                           // accesses on the largest data type (floats)
unsigned long gNumberOfMsalignments = 0;    // Msalignments forced by
                                            // program
unsigned long gReportedMsalignments = 0;    // Msalignments reported
                                            // by exception services
void main(void)
{
   float MsalignedTime = misalignLoop(false);
   printf(">*> Forced number of misalignments: %d\n",
            gNumberOfMsalignments);
   printf(">*> Reported number of misalignments: %d\n",
            gReportedMsalignments);
}

// The function misalignLoop measures the time of a loop of "writes" to
// a byte array. The writes are either aligned or misaligned, based on
// the align parameter.
double misalignLoop(boolean align)
{
   UnsignedWide   startTime, stopTime;
   float          start, stop;
   short          alignIndex = (align)?0:1;

   // • MIL instrumentation code: initialize misalignment counter.
   initMsalignRegs();

   // Get starting timestamp.
   Microseconds(&startTime);
```

(continued on next page)

Listing 3. Sample application code using the MIL *(continued)*

```
    for (long j = 0; j < kNumCycles; j++) {
        char byteTable[kTableSize];
        float *floatPtr = (float *) &byteTable[alignIndex];
        for (long k = 0; k < kNumAccessesPerCycle; k++) {
            gNumberOfMsalignments++;
            floatPtr[k] = 1;
        }
    }

    // Get ending timestamp.
    Microseconds(&stopTime);

    // • MIL instrumentation code: get number of misalignments.
    gReportedMsalignments = getMsalignments();

    // Move the values to doubles.
    start = (((double) ULONG_MAX + 1) * startTime.hi) + startTime.lo;
    stop = (((double) ULONG_MAX + 1) * stopTime.hi) + stopTime.lo;

    return stop - start;
}
```

The results of running this code are shown in Table 4. There's a large discrepancy between the number of misalignments generated and the number reported by the MIL on the 601. The 601 architecture internally fixes float and double misalignments in hardware. However, the 601 can't fix misalignments across page boundaries, so it takes a misalignment exception. Thus, only those page boundary cases are reported. The 603/604 architecture doesn't handle misalignments in hardware, and it takes an exception in all cases.

Table 4. Number of misalignments generated and reported

CPU	Number of misalignments	Reported number of misalignments
PowerPC 601	1000000	120
PowerPC 604	1000000	1000000

NOW FOR THE CHIROPRACTIC ADJUSTMENTS

You've determined with the help of PPCInfoSampler that you have misalignment problems in your application. You've used the MIL to determine where the misaligned accesses are and the structures that they're associated with. Now it's time to think about these structures.

Is there a reason why they can't be naturally aligned (as we described early in the article)? If there are structures in the parameter block passed to a Toolbox call, fields within these structures may not naturally align, but this is something the programmer can't do much about until the system provides an alternate API. Perhaps there are binary data files created and accessed from 680x0 legacy code. Is it really necessary to still be supporting data files formatted to 2-byte alignment? Can you provide a version

mechanism for your data files, such that newer versions write and read to natural PowerPC alignment? Ask these questions, and naturally align structures as much as possible.

You can ensure proper structure alignment by ordering the fields in your structures by hand, from largest to smallest, instead of relying on a compiler to pad the fields. This will require you to do more work but will remove reliance on any particular padding strategy.

Another possible scenario is the case where data files are shared across multiple platforms. Alignment strategies on Intel and other x86 processors aren't the same as on PowerPC processors. There are two possible approaches to this scenario, given an application on Windows and one on the Mac OS that share the same data files:

- Try compiling structures on the Windows application with 4-byte alignment, according to PowerPC natural alignment. This is a "least common denominator" approach.
- If rebuilding a Windows application isn't a viable option, load the data from disk and convert the data structures to an aligned structure that's used internally. The performance tradeoff depends on how often the misaligned structure is used.

IT'S THAT SIMPLE

Misaligned memory accesses can take a real toll on your application's performance, requiring from 2 to 80 times longer than aligned accesses on newer PowerPC CPUs. If you do what we've described in this article, you can detect and pinpoint misalignments and fix them so that your code will run efficiently now and on future processors (which won't include hardware to fix misaligned accesses for any misaligned data type) and won't be penalized by the lack of hardware support in future implementations of PowerPC architecture. Isn't it worth a few simple adjustments now to know that your code's future is secure?

Thanks to our technical reviewers Justin Bishop, Dave Evans, and Jim Gochee. The authors would like to acknowledge the help given by members of the Performance Evaluation Group (a subgroup of the Architecture and Technology Group in Apple's Hardware Division), including Marianne Hsiung, Tom Adams, and Scott McMahon. We'd also like to thank Jim Gochee for his toolsets and valuable insights.

Macintosh Q & A

Q *My application animates moving geometries in QuickDraw 3D. Recently I've been seeing a lot of screen flicker, and faces of geometries that should be behind other faces are showing through. What's going on?*

A The flickering problem is probably happening because double buffering is turned off (call Q3DrawContext_SetDoubleBufferState to turn it on) or because double buffer bypass is set on the interactive renderer and the scene is taking longer than a screen refresh to render. See page 12-8 of *3D Graphics Programming With QuickDraw 3D* for more information.

Your second problem is likely the result of having an excessively large difference between **hither** and **yon** (and, as a result, not having enough *z* resolution to resolve depth differences). Experiment with greater **hither** values and smaller **yon** values to see if the bleed-through goes away.

Q *We're using QuickDraw 3D and applying UV attributes to our geometries so that we can texture-map them. There are, however, two sorts of UVs: surface UVs and shading UVs. Which one should we use to get the textures to map correctly and what does the other one do?*

A At the time of this writing (QuickDraw 3D 1.5), only the surface UVs are supported. The shading UVs will be used in a future version to support advanced shading renderers.

Q *What are view hints in the QuickDraw 3D metafile format (3DMF)?*

A The concept of view hints was included early on in the development of QuickDraw 3D. It became apparent that the settings for determining how a scene should be rendered aren't always transportable from one application to another (for example, settings such as the camera location, lighting, and camera type). The idea of a view hint is that it sets up a series of hints that tell the reading application how the author of a metafile intended the geometries within the metafile to be rendered. The fact that these are hints implies that the reading application can ignore them.

Rather than writing out the lighting information to the metafile as absolute objects, we recommend creating a view in the normal manner, adding lighting, camera, renderer, and other information as usual, and then extracting the view hints from the view with Q3ViewHints_New(theView). You pass in a view object to this function, and it returns a view hints object that includes the view configuration for the view you pass in. The Tumbler and Podium sample code that comes with the QuickDraw 3D release illustrates how to use view hints read from a metafile to configure a view. Look in the file Tumbler_document.c for details.

Q *What's the best way to do collision detection in QuickDraw 3D?*

A QuickDraw 3D doesn't directly provide collision detection, but you can use the bounding boxes or bounding spheres of the geometries to determine whether the bounds intersect. You can easily calculate bounding boxes and spheres on either individual geometries or groups of geometries. If the bounds intersect, you can either assume the objects have collided or test further, depending on your application.

Q *In the TQ3CameraData record, is the position of the point of interest relative to the camera location used for anything other than the view direction of the camera?*

A The point of interest is an absolute position — it's not relative to anything. As the camera's location changes, the camera "turns" to keep the point of interest in view.

Q *TQ3HitData's distance field is described for window-point picking as "the distance from window point's location on the camera frustum (in world space) to the point of intersection with the picked object." Does this refer to the center of the intersection of the pixel's frustum with the hither plane, or maybe the yon plane? Or is it the camera location?*

A TQ3HitData's distance field is the world space distance from the origin of the picking ray to the intersection point. Effectively, this is the distance from the camera's location to the geometry intersection point in world coordinates.

Q *When I print with background printing enabled, and PrintMonitor fires up and tries to post an error, my application hangs, and no dialog ever appears. What can I do?*

A Make sure the bits in your SIZE resource are set correctly. This behavior appears when you've got your "MultiFinder aware" bit set but don't have your "accepts Suspend/Resume events" bit set.

Q *I can't figure out how to get the default settings for the features in a given QuickDraw GX font. Any ideas?*

A A routine to do this, GXGetFontDefaultFeatures, was added after the release of QuickDraw GX 1.0. This routine will retrieve those layout features defined as default by a given font. It's fully documented in Technote 1028, "Inside Macintosh: GX Series Addenda."

Q *I want to turn my font (generated with a third-party font design program) into a true QuickDraw GX font with a customized features menu. How do I do this? What programs are available for GX font design? Will I be able to add a features menu to my font after generating it with a non-GX font design program?*

A For custom design of QuickDraw GX features in a font, you should use TrueEdit, which is available (along with other font tools) in the QuickDraw GX folder on the Mac OS SDK edition of the *Developer CD Series*, or via ftp in this directory: ftp://ftpdev.info.apple.com/Developer_Services/Development_Kits/QuickDraw_GX/Goodies/Font_Tools/.

The general process of designing a QuickDraw GX font starts with building all the glyphs you're interested in and hinting them, just as you would for a non-GX font. Once you have the glyph repertoire, use TrueEdit to add all the GX tables. You should be able to add the tables for the various menu features with TrueEdit just fine after you've built the font with another font design program.

Q *I read somewhere that you don't have to call CloseOpenTransport if you're writing an application. Is this true?*

A Yes and no. The original Open Transport programming documentation stated that calling CloseOpenTransport was optional for applications. There is, however, a bug in Open Transport 1.1 and earlier whereby native PowerPC applications aren't properly cleaned up when they terminate unless CloseOpenTransport is called.

Here are some rules of thumb:

- Nonapplication code must always call CloseOpenTransport when it terminates.
- It's best if 680x0 applications call CloseOpenTransport, but they'll be cleaned up automatically even if they don't.
- Make sure that PowerPC applications running under Open Transport 1.1 or earlier call CloseOpenTransport when terminating.

One way of ensuring that you comply with the third item is to use a CFM terminate procedure in your main application fragment, like this:

```
static Boolean gOTInited = false;
void CFMTerminate(void)
{
    if (gOTInited) {
        gOTInited = false;
        (void) CloseOpenTransport();
    }
}

void main(void)
{
    OSStatus err;
    err = InitOpenTransport();
    gOTInited = (err == noErr);

    ...    // the rest of your application

    if (gOTInited) {
        (void) CloseOpenTransport();
        gOTInited = false;
    }
}
```

In general, when the Mac OS provides an automatic cleanup mechanism, it's normally intended as a "safety net." It's always a good idea to do your own cleanup, at least for normal application termination.

Q *I'm using OTScheduleSystemTask to schedule a task to run at non-interrupt time. Will this be cleaned up automatically when I call CloseOpenTransport?*

A No. You must explicitly clean up any pending system tasks by calling the routine OTDestroySystemTask. See Technote 1059, "On Improving Open Transport Network Server Performance," for a good discussion of this.

Q *How do I map Open Transport error numbers to their names?*

A There are two ways to do this. The first is the OTErr MacsBug dcmd that ships with the debugging version of Open Transport. This dcmd allows you to quickly map an error number to a (hopefully) meaningful error name. Once you install it in your Debugger Prefs file, you can type, for example, "oterr -3271" in MacsBug and get the name for that error.

The second solution is to read the latest OpenTransport.h header (for Open Transport version 1.1.1 or later), where the error numbers are now spelled out in an easy-to-read format.

Q *I'm writing an Open Transport server product and will be implementing hand-off endpoints. What is the maximum* ***qlen*** *value that limits the number of hand-off endpoints that can be implemented?*

A There's a maximum **qlen** value for each protocol, but maximum values that are true today for Open Transport may change in the future, so we recommend that you set the **qlen** value to a *desired* value. If the desired value is greater than the number of hand-off endpoints that the underlying protocol can support, the protocol can specify its own maximum **qlen** value for the server endpoint. After making the OTBind call, take a look at the **qlen** field of the TBind structure to see whether the protocol imposed a limit on the **qlen** value.

Q *I'm developing a plug-in (let's call it MyPlugIn) for an application (let's say CoolApp). I'd like to create a special icon for the documents my plug-in creates, but they're really CoolApp documents. However, if I add a BNDL resource to MyPlugIn, all other CoolApp plug-ins become MyPlugIn documents. What can I do?*

A Just register a new, unique creator code and use that for the BNDL, but not for the plug-in. Contrary to popular belief, the owner code for a BNDL doesn't have to match the creator code of the file it's in.

Another possibility is to create custom icons for documents you create, but this has a couple of drawbacks: it takes the Finder longer to display them, and you'd be storing redundant data if the icons are all the same.

Finally, you should also add 'STR ' resources with IDs -16396 and -16397 so that the Finder can display a meaningful message if an application can't be found to open the file. (See *Inside Macintosh: Macintosh Toolbox Essentials*, pages 7-27 to 7-30.)

Q *I'm using MPW and MacApp from E.T.O. 20. Where are the 411 help files?*

A Starting with E.T.O. 20, the 411 files have been converted to QuickView™ format and can be accessed through the Info menu of the MPW Shell.

Q *Why is Apple now making the latest versions of MacApp available under a "release" approach, instead of the previous "product" approach?*

A In response to the many messages we received from developers requesting early access to new framework features and timely support for new technologies, Apple has implemented a release approach with MacApp that allows us to get new improvements and features in our frameworks into your hands more quickly than was possible with the product approach.

Previously, our product approach required that we implement all planned features before the MacApp product was considered final. We found that this was keeping us from getting new features to you simply because other more time-consuming work was delaying the completion of the product. Under the new approach, each framework release will be made up of features at various levels of certification. Most features will be of final quality, while others may be of beta or alpha quality. You can choose the features to build your MacApp-based application with, and by doing so you'll choose the quality level of the resulting application. Our build tools will indicate the quality level you've chosen from the build flags you've passed to them. The release notes that accompany each MacApp release will list the features included in the framework and their quality status.

Over the course of multiple releases, every feature will proceed through alpha, beta, and final quality phases. Some features will move rapidly from development into final quality, perhaps in as little time as one release, while other features may require several releases. This approach ensures that the software features Apple provides to you are of the highest possible quality, while still allowing you to experiment with new, unproved features. Apple will create a new release version of MacApp approximately once every six to nine months, and the MacApp product will ship once each E.T.O. delivery cycle. Releases of MacApp that occur between E.T.O. shipment dates will be posted to the Web at http://www.devtools.apple.com/macapp/.

Note that for each MacApp release, you should *always* refer to the release notes for the quality status. For some releases, we recommend that you don't build applications that you intend to rely on as final-quality applications. Releases that have final-quality status are suitable for building final-quality MacApp-based applications.

Q *When linking my application in MPW, I get this error message:*

```
### Link: Error: Can't open object file for input. (Error 32)
ETO#19:Libraries:CLibraries:CSANELib.o is not an object file.
```

Why am I getting this message?

A If you look at the CSANELib.o itself, you'll see that it's really just a text file containing this:

```
The library "CSANELib.o" is obsolete beginning with the PreRelease
environment of ETO 18 and the Latest environment on MPW Pro 19.

Use the "{Libraries}MathLib.o" library instead.

"CSANELib.o" is incompatible with the SC/SCpp compilers,
and is incompatible with the new FPCE floating point model.
See the header file <fp.h> for more details.

Please read the release notes.

You may safely delete this file when you are sure that all of your Make
files have been updated.
```

In fact, as of E.T.O. 20, all of the following libraries are obsolete: Runtime.o, Complex.o, Complex881.o, CPlusLib881.o, CPlusOldStreams881.o,

CPlusOStreams.o, CSANELib.o, CSANELib881.o, Math.o, Math881.o, AppleScriptLib.xcoff, CPlusLib.o, InterfaceLib.xcoff, MathLib.xcoff, ObjectSupportLib.xcoff, QuickTimeLib.xcoff, SpeechLib.xcoff, and StdCLib.xcoff. These "libraries" are now text documents, similar to the above. Each of them contains information about the libraries you should use instead.

Q *I understand how to use PrGeneral with the getRslDataOp opcode to get the resolutions that a printer supports, but when I ask the LaserWriter driver about resolutions, it always tells me I'm talking to a 300 dpi printer, even when I know the printer is, say, 600 dpi. I want to print at the maximum resolution available. How can I do that?*

A Currently, the LaserWriter driver always returns a range of 25 to 1500 dpi for valid resolutions, and a fixed resolution of 300 dpi. The range of 25 to 1500 means that your application can ask for any resolution in that range, and the driver will allow it, as explained in Technote PR 07, "PrGeneral." However, given that LaserWriters and other PostScript printers can support only a limited number of resolutions, you may not get optimal results if you pick the wrong resolution for a printer, even if the driver lets you. You'll also end up sending more data than is needed if you're generating bitmaps at a higher resolution than the printer can print, which can slow printing down significantly. Furthermore, some extremely high-resolution devices may be able to print at a resolution higher than 1500 dpi, but the range returned by PrGeneral was chosen quite a while ago and hasn't been changed for application compatibility reasons.

Currently, the only way to get the actual resolution(s) supported by a PostScript printer (short of querying the printer directly) is to ask the LaserWriter driver for the PPD file and parse it yourself, looking for the valid resolutions. To get the PPD file, call PrGeneral with the PSPrimaryPPDOp opcode, which is 15. This is documented in the Macintosh Technical Q&A QD 01, "PPDs." When Apple makes the functions in LaserWriter 8.4's PPD Library public, you'll also be able to use those functions to get the information from the PPD, but the API to that library hasn't yet been made available as of this writing.

To parse the PPD file, you'll need to consult the PPD specification maintained by Adobe. That document can be found at Adobe's ftp site, at the location ftp://ftp.adobe.com/pub/adobe/devrelations/devtechnotes/psfiles/.

Rather than go to all that work, however, a better approach might be to change your code so that you don't need to know the printer's resolution in order to generate high-quality output. Some of the various ways to solve this problem are listed below; the approach you take will depend on your application requirements. Also, see the Print Hints column in this issue of *develop*, which discusses this very problem (among others).

- If your application is such that you require a PostScript printer, you could generate your own PostScript code to achieve high-quality results.
- You could generate your own PostScript code but also use a QuickDraw representation so that images will print correctly to StyleWriters and other QuickDraw printers. This solution is recommended if you're writing a program that draws curves, such as a graphing program. For best results, break down the curves you need to draw into small portions that can be accurately represented by the QuickDraw primitives you have at your command.

- You could use QuickDraw only, but draw in such a way that you'll get high-resolution results. To do this, use objects such as lines, ovals, and arcs rather than bitmaps. The endpoints of the objects will be limited by the resolution at which you draw, but the objects will be drawn at device resolution.

Q *I noticed that several QuickTime Music Architecture routines (TuneResume, TuneFlush, TuneGetState) are missing in the latest headers. What gives?*

A The routines TuneResume, TuneFlush, and TuneGetState were poorly defined, and in fact were unimplemented in QuickTime 2.0 and 2.1. They've been removed from the headers.

Q *_StuffXNoteEvent, a QuickTime Music Architecture macro, has vanished from the latest headers. Why?*

A Whoops. A mistake was made, and _StuffXNoteEvent didn't make it into the final release header. You can copy the macro from the older header file if you need it, but for consistency you should rename it qtma__StuffXNoteEvent. Here it is as well:

```
#define qtma_StuffXNoteEvent(w1, w2, instrument, pitch, volume, duration) \
    w1 = (kXNoteEventType << kXEventTypeFieldPos) \
    |   ((long)(instrument) << kXEventInstrumentFieldPos) \
    |   ((long)(pitch) << kXNoteEventPitchFieldPos), \
    w2 =  (kXEventLengthBits << kEventLengthFieldPos) \
    |   ((long)(duration) << kXNoteEventDurationFieldPos) \
    |   ((long)(volume) << kXNoteEventVolumeFieldPos)
```

Q *Why did the old _EventLength(x) macro get split into two macros, qtma_EventLengthForward(xP, ulen) and qtma_EventLengthBackward(xP, ulen)?*

A _EventLength(x) had to be changed because of some new event types. We needed separate macros to determine the length from the first long word of a music event and from the last one. Typically, you'll be using qtma_EventLengthForward.

Q *Inside Macintosh: QuickTime Components states that the limit on data transfer rates for the base media handler (and therefore all media handlers derived from it) is 32 kilobits per second. Is that true?*

A Starting with QuickTime 2.0, when the data handler API became publicly available, that statement is no longer accurate. In fact, QuickTime imposes no limitation on performance at all; the hardware you're using is the only limiting factor.

Q *I heard at Apple's Worldwide Developers Conference last May that the QuickTime for Windows installer can be customized through a ".INI" file. Where can I get more information about this?*

A Following is the format for the optional configuration file that can be used with the installer for QuickTime for Windows version 2.1.2. If used, the file must be named QTINSTAL.INI, and it must be located in the same directory as the installer, QTINSTAL.EXE. (The 32-bit installer is called QT32INST.EXE, and the corresponding ".INI" file must be called QT32INST.INI.)

For all of the options listed except DialogStyle, a value of 1 enables the option and a value of 0 disables it. The default for all values is 1, unless otherwise noted. Although all combinations of options are designed to work (that is, the program will run correctly), not all combinations will yield a viable result. For example, creating a program group without unpacking the files would probably not be a good idea.

```
; All QTWinstaller options must be in the following section:
[Options]
; The QuickTime background can be suppressed when QTINSTAL is to be
; called from another program:
;    1 - shows a background window with QuickTime banner.
;    0 - shows no background window.
StandAlone=1
; A Dialog style may be specified by one of the following values:
;    1 - Thin frame
;    2 - System menu
;    3 - Thin frame and system menu (default)
DialogStyle=3
; If the following option is set, the client area of all dialogs will
; have a 3-D look.
Ctl3D=1
; If the following option is set, the installer will display the QTW
; end-user license agreement, with Agree/Disagree buttons.
DisplayLicenseAgreement=1
; If the following option is set, an opening dialog will prompt the user
; whether to begin the installation or to exit.
PromptToBegin=1
; If the following option is set, existing-version checking will be
; enabled and the installer will evaluate the PromptToDeleteVersions
; option.  If CheckExistingVersions is set to zero, the installer will
; not check for existing versions.
CheckExistingVersions=1
; If CheckExistingVersions and the following option are set, the
; installer will prompt the user before doing a search operation for all
; out-of-date QTWinstallations on the machine.  For each installation
; found, a dialog will ask the user whether to mark it for deletion.  If
; this option is set to zero, the search will be unconditional and all
; installations found will be marked for deletion.
PromptToDeleteVersions=1
; If the following option is set, a "do you want to continue" dialog will
; appear before any files are deleted or the hard disk is modified in any
; way.
PromptToComplete=1
; If the following option is set, all files will be unpacked from the
; executable and written to disk.  Care and consideration should be used
; before setting this option to zero, since a zero value means no files
; will be installed.
UnpackFiles=1
; If the following option is set, the Windows INI files will be updated.
UpdateIniFiles=1
; If the following option is set, Program Manager groups will be created.
CreateGroups=1
; If CreateGroups is set and the following option is used, the specified
; name will be used as the group name (that is, the name as it displays
; in the window titlebar, not the group filename) that the installer will
```

```
; use when installing the QuickTime icons.  For example, the option shown
; would use the group name "My Group."  If the group does not exist it
; will be created.  This option can be used to add the QuickTime
; applications to an existing group file.  The string used to specify a
; group name should be tested in actual use, since there is a practical
; upper limit to the number of characters Windows will use in a window
; title.
GroupName=My Group
; If the following option is set, a success dialog will indicate whether
; the installation has completed successfully.
SuccessDialog=1
; If SuccessDialog and the following option are both set, the installer
; will launch Movie Player with a sample movie to verify that QTW has
; been installed successfully.
PlaySampleMovie=1
```

The Installer also always creates a file in the Windows directory called RESULT.QTW that looks like this:

```
[QTWInstall 16]
Complete=1
[QTWInstall 32]
Complete=1
```

If Complete equals 0, the installation didn't finish for some reason (any reason, such as the user canceling, or running out of disk space, or whatever). This enables the title installer to tell whether the QuickTime for Windows installation was successful and to respond appropriately.

Q *I'm playing four QuickTime movies simultaneously from a Director project. Each movie has a single music track with no other video or sound tracks, and two of the movies use more than one instrument. The Director project lets the user control the volume level of each movie independently. The application works great on the Macintosh with QuickTime 2.1, but under Windows with QuickTime 2.11 only one music track plays at a time. Is it possible to hear all four music tracks at once under QuickTime for Windows 2.11?*

A You can do live mixing of your four QuickTime movies only if your Windows system has four MIDI output devices. Most systems have only one. All Windows applications suffer from this limitation unless they're clever enough to mix the tracks on the fly, but none seem to do this. For now, you must pre-mix the four music tracks from the four movies into one music track in one movie. You won't be able to do live mixing unless you write your own MIDI sequencer.

Q *Why do my eyes water when I chop onions?*

A Onions contain sulfur compounds, and when you cut into them they release sulfur dioxide. When the sulfur dioxide gas dissolves in your tears (which are mostly water), sulfurous acid is produced: $SO_2 + H_2O \rightarrow H_2SO_3$. The acid burns your eyes, which respond by generating more tears in an attempt to flush away the acid. See http://www.superscience.com/onions.html as well.

These answers are supplied by the technical gurus in Apple's Developer Support Center. For more answers, see the Technical Q&As on this issue's CD or on the World Wide Web at http://www.devworld.apple.com/dev/techqa.shtml. (Older Q&As can be found in the Q&A Technotes, which are those numbered in the 500s.)•

THE VETERAN NEOPHYTE

Confessions of a Veteran Technical Writer

TIM MONROE

I've been a technical writer long enough to have learned a few trade secrets, if you will, that guide me in my daily work and (sometimes, I hope) help me to do it a little better. Whether you're reading manuals for content, working with technical writers to document your own software, or even writing documentation yourself, understanding these secrets might be of value to you. I've long had it in mind to write a book about these and other topics, sort of a general discourse on technical thought and how to do it better. Until I actually write that book, however, the following confessions will have to do.

These confessions should help you understand some of the tasks that some technical writers typically perform and some of the conflicting forces that shape final documents. (Note the profusion of the word *some*: your mileage may vary.) For fun — and for another reason that I won't tell you yet — each confession is introduced by an appropriate palindrome (a word or phrase that reads the same forward and backward).

MADAM, I'M ADAM

As you probably know, Adam was given the task of naming the plants and animals. In a way, Adam was the first technical writer, for an important part of technical writing is to systematize and regularize the nomenclature used in some specific area of interest. It's rare for the engineers developing software to pay very close attention to naming issues. Indeed, the rule is quite the contrary: more often than not, the technical specifications written by engineers are rife with conflicting, inconsistent vocabulary. It nearly always falls to the writers and their editors to clean things up and present the technology using clear, regular, and concise terminology.

There's a moderately foolproof way to discern which writers take this job seriously and which do not: just look at a document's glossary. For my money, every draft of every technical document should include a robust glossary that defines the special terms and concepts used in the document. You simply cannot write any significant part of a technical document without grappling with naming issues. A working, growing glossary is the writer's proof that he or she is actively thinking about these issues and is developing a preferred vocabulary to describe the technology being documented.

Don't be misled by the fact that the glossary is usually one of the last items in a book: as I see it, the glossary ought to be written concurrently with the book, not after it. (The index is another issue altogether: a book cannot be properly indexed until it's nearing completion. Trying to do an index while a document is still changing usually results in tremendous frustration.)

SUE US, ONO, SUE US!

At some point in the distant past, Apple Computer reached an agreement with the Beatle's record company, Apple Records, that allowed Apple Computer to use the name "Apple" so long as it did not engage in certain markets, such as music recording. At some later point in the distant past, Apple Records sued Apple Computer, alleging certain violations of that agreement. Suddenly, the lawyers at Apple Computer were intensely interested in the sound and music capabilities of the Macintosh hardware and system software.

At that moment, I happened to be finishing up the Sound Manager chapter of *Inside Macintosh* Volume VI. Apple's lawyers decided that a number of API elements smacked too much of music and needed therefore to be changed. For instance, it was thought that since music is composed of individual notes, the word *note* should not occur anywhere in the documentation in any sound-related sense. As a result, what was hitherto known as the noteCmd constant was changed to freqDurationCmd (the idea being that playing a note is just playing a frequency for a specific duration). The legal department demanded a number of other changes, which led to some last-minute rewriting and reindexing as the book neared publication. And, of course, the engineers had to issue new header files to reflect the new names.

I don't imagine that Yoko Ono and other Apple Records executives have spent much time reading *Inside Macintosh*, so I doubt that my efforts were all that critical. Nonetheless, I learned my lesson. I confess: I've come to appreciate the difficult job done by the Apple legal department. I now pay close attention to

TIM MONROE (monroe@apple.com) recently became a Senior Software Engineer in the QuickTime VR group at Apple. He already misses the smart rams and star rats in the technical writing group at Apple Developer Relations. In his spare time, he likes to stack cats and drive his race car to the local llama mall.

the lists of trademarks distributed by the editors in our department, and I'm constantly on the lookout for API elements that might step on someone's copyrighted toes.

YAWN WAY

Let's face it: technical writing isn't creative writing. It's not designed to enthrall, just to educate and to serve as a useful reference during your daily work. To be useful, a technical document has to be complete. It also has to be consistent in style and vocabulary with other similar documents — in my case, with other *Inside Macintosh* books. ("No manual is an island.") Providing documents that are both complete and consistent usually means following a pretty strict set of rules and guidelines governing the style, organization, and content of the document. It can get to be drudgery, sometimes. Yawn.

On the upside, having rules and other established methods to follow can be incredibly liberating. These shackles free you from constantly having to rethink major issues about a document you're writing. Once you figure out what goes where, in a general sense, you almost don't need to think any more. You just take the API elements that need documenting, plug them into the correct paragraph formats, and fill in the appropriate information. It can get to be too easy, sometimes. Yawn.

Following rules, though laudable, is not without its own problems. The requirement that every constant in an API be precisely described can result in some pretty silly stuff. For instance, page 2-33 in the book *Inside Macintosh: PowerPC System Software* takes the trouble to inform us that the constant kRegisterD0 stands for the register D0. Did anyone have any doubt about that? It can get to be too dumb, sometimes. Yawn.

Have we reached the stage where the API is self-documenting, where just the names of functions and constants give us all the information we need to use them effectively? I don't think so. It's just plain dangerous to start having writers decide what's too obvious to need description and what isn't. In my mind, every element of an API should be fully documented, even if we end up with a few odd cases where there really is nothing more to say. Remember, degenerate cases like these are a direct result of systematically applying rules and established methods. They're a clear signal that the writer is doing things exactly right.

IF I HAD A HI-FI...

…I'd play it real loud, and I'd turn the bass way up. That's the only way to play reggae music, which is what I listen to mostly. My taste in music is quirky, but that's a problem of mine generally. I confess: I adopt methods that help me get my work done, even if those methods are quirky. I revel in quirks, because they often pay off.

So I'll confess another quirk of mine: I write my documents backward. For each chapter, I start at the end and write the summary first. This actually makes good sense, since what I'm documenting is usually an API, as defined by a header file. The summary is really just an improved header file. It's improved in part because it attempts to impart more order and consistency than you'll find in a typical header file. It's important to keep in mind that header files are designed for compilers, not for humans. There's lots of junk in these files that has absolutely no meaning to a programmer. The summary provides an ordered distillation of the header file to its key components.

Once I've written my summary, I have a good head start on the reference section. That's because the summary already contains intelligent groupings and orderings for the basic language elements. To write the reference section, I simply need to "fill in the blanks" provided by the summary: Each constant needs a precise definition. Each data structure needs to have its use explained, and each of its fields must be fully described. Each function has parameters that need describing, and it probably returns a value that must be described.

Only when I've finished the summary and reference sections do I even think about the first parts of the chapter, the About and Using sections. At that point, I need to turn down the music and do some real thinking.

GOD? DOG?

Good technical writing is far too often simply taken for granted. Partly that's an occupational hazard: when it's done right, good documentation is unobtrusive: it purposely doesn't try to be cheeky or clever. More important, good technical writing is unobtrusive because it doesn't jar the reader with confusing organization, sloppy diction, or bad transitions from one topic to the next. Its job is to conform to established styles and norms, and to present information as straightforwardly and clearly as possible. Nonetheless, it bugs me that I've never seen a single review of any *Inside Macintosh* book. Even magazines that are geared specifically at Macintosh programmers, like *MacTech Magazine*, never actually bother to review these important books. Third-party books get plenty of discussion, but not the primary documents they all draw on.

I see some other signs that technical writing is taken for granted — like product managers who try to bring a writer onto a project two weeks before the CDs are to be pressed, and engineers who would rather have a root canal than review the chapter describing the technology they've slaved over for months or perhaps years. What these people are missing is that good documentation can add a considerable amount of value to an engineering

product. Documentation is the first and most important bridge between the engineers and the developers using their technology. If the documentation paints a compelling reason to adopt the technology and facilitates that adoption by providing useful sample code and complete descriptions of the API, the writer is a god. Occasionally, the "wow" factor emerges: documentation that is so compelling that it opens your eyes wider and makes your fingers itchy for some coding.

The flip side of the "wow" factor is the "dud" factor: documentation that is so patently bad it almost single-handedly ensures limited adoption for the technology it describes. I've seen some really good technologies languish for years, for no other discernible reason than that they're tied to some really lousy documentation. The best software technology and the best API cannot survive a mauling by a dog technical writer.

"OTTO," MY MOTTO

Palindromes delight us because they call attention to a fairly rare phenomenon: a sequence of letters that is meaningful and that reads the same forward or backward. Language wasn't designed to be palindromic, and there is no cognitive benefit — no additional information — in a particular phrase or sentence that happens to be reversible. It's usually a serendipitous accident that some long sentence should be palindromic.

At their best, palindromes tweak our sense of order. As I've suggested above, order itself is deceptive. The linear order of presentation that you find in technical documentation like *Inside Macintosh* reflects, in all likelihood, neither the order in which the document was written nor the order in which you're most likely to access the information in it. In fact, the principle according to which most good documentation is organized is a complex hybrid of at least two ordering schemes.

On one hand, there is a principle governing how you should be able to learn from a document: you should be able to pick up a document (a book or a chapter) and read it from start to finish with good comprehension. Concepts should be explained in a clear, cumulative order, and tasks should be explained in the order they need to be performed. I like to call this a *pedagogically linear* path through the document. The good writer progressively reveals more and more of the technology as he or she goes along. This is the main principle that governs the opening sections of an *Inside Macintosh* chapter (the About and Using sections).

On the other hand, there is a principle governing how you should be able to find information in a document. Technical documentation is, above all, a type of reference material. You're constantly jumping into the middle of a chapter to find the meaning of a constant, or the type of a parameter to a function, or some similar piece of information. It's common to call this *random access* to the information, but I prefer to avoid that term, since it might suggest that the information itself is ordered randomly. In my opinion, pure reference material (such as that found in the reference and summary sections of *Inside Mac* chapters) must be organized hierarchically, where the items are intelligently divided into groups, which are themselves further subdivided into groups, until every item can be reached by a meaningful path from the top of the hierarchy. I like to call this the *hierarchically linear* path to the information. (For reference material, the main competitor to a hierarchical organization is the standard alphabetical organization, which some of you might prefer. Personally, I like to have things grouped by functional similarity, not ordered — "one darn thing after another" — by name.)

The pedagogical order and the hierarchical order are clearly different ways of organizing information. If you follow a pedagogical path, you might not get where you want to go very quickly, but you'll get a complete picture of the terrain as you pass through it. If you follow a hierarchical path, you can get where you want to go pretty quickly, but only if you already know where you're going. So, to use the hierarchical path, you must already have traveled the pedagogical path. These two ordering principles depend on each other and should never be separated.

As you can see, order dominates my mind, at least when I'm writing technical documentation. That's why my motto is a palindrome. Attention to order — and clearly understanding exactly what kind of order is relevant to the task at hand — is the foundation of all good technical writing.

RELATED READING

- *Go Hang a Salami! I'm a Lasagna Hog! and Other Palindromes* by Jon Agee (Farrar, Straus, and Giroux, 1991).
- *Inside Macintosh* by Apple Computer, Inc. (Addison-Wesley, 1992 and following).

Thanks to Dave Bice, Sharon Everson, and Antonio Padial for reviewing this column.•

Newton Q & A: Ask the Llama

Q *I was wondering why my autopart is taking up so much heap space after it's installed. The InstallScript and RemoveScript are quite small:*

```
constant kGlobalDataSym := '|Globals:MYSIG|;

InstallScript := func(partFrame, removeFrame) begin
    if NOT GetGlobals().(kGlobalDataSym) exists then
        GetGlobals().(EnsureInternal(kGlobalDataSym)) := {};
    GetGlobals().(kGlobalDataSym).(EnsureInternal(kAppSymbol))
        := GetLayout("MyTool.t");
end;

RemoveScript := func(removeFrame) begin
    local NGP := GetGlobals().(kGlobalDataSym);
    if hasSlot(NGP, kAppSymbol) then
        RemoveSlot(NGP, kAppSymbol);
end;
```

The template in MyTool.t is only a simple proto with a few slots in the base view: a symbol, viewBounds, viewFlags, viewJustify, viewFormat, viewClickScript, and three methods. When installed, the autopart takes up about 640 bytes of heap space. Is this because of the physical representation in the extras drawer?

A A minimal autopart with no RemoveScript will take up about 240 bytes. One with a RemoveScript will take 350 bytes or more, depending on the size of the RemoveScript. The 240 bytes are used by the system to keep track of the package. This includes the name, the extras drawer entry, and any symbols that you passed to EnsureInternal.

Trying your code showed that it used 444 bytes. Of course, we don't have the MyTool.t layout in the package, but that doesn't matter since you don't call EnsureInternal on the layout. Using 444 bytes isn't out of line considering the minimum size of an autopart.

Of course, you could make your code a little smaller. For example, in your RemoveScript you check that kGlobalDataSym is a known global variable. This isn't required, since RemoveSlot will do the right thing if kAppSymbol isn't in your global data frame. Note that you may want to make sure your global data frame exists:

```
RemoveScript := func(removeFrame)
    RemoveSlot(GetGlobalVar(kGlobalDataSym), kAppSymbol);
```

Also, you use GetGlobals, which is a deprecated function for Newton OS 2.0. You should be using GetGlobalVar, DefGlobalVar, and UnDefGlobalVar.

Q *I'm using a protoPeoplePicker in Newton OS 2.0 and it keeps showing an "Untitled Person." Is this an opportunity to create someone or is there really a gremlin inside the machine?*

The llama is the unofficial mascot of the Developer Technical Support group in Apple's Newton Systems Group. Send your Newton-related questions to dr.llama@newton.apple.com. The first time we use a question from you, we'll send you a T-shirt.•

A There are only two reasons for the appearance of the "Untitled Person" entry. One is that you mistakenly entered that person in your card file, which can happen if you edit an entry and accidentally erase the name. The more likely reason is that this is your hacking Newton device. If so, you probably didn't go through the Setup application and set an owner. "Untitled Person" is the default owner of the machine. We recommend that you set up a default hacking MessagePad, back it up using NBU (Newton Backup Utility), and then use that backup to create development units.

Q *I've read the article in Newton Technology Journal on package freezing and I'm still a bit confused. Can you confirm that the following questions and answers are right?*

- *Does my form part still have a slot in the root after being frozen? No.*
- *Is the RemoveScript called when it's frozen? Yes.*
- *Is the InstallScript called when it's unfrozen? Yes.*
- *Can I prevent freezing/unfreezing? No.*
- *Can I get a message indicating freezing vs. pulling the card? No.*
- *What happens when the user tries to put away an item routed to a frozen application? A "can't find application" error.*

A Congratulations, you understand package freezing correctly. And you win an all-expenses paid visit to your nearest Green Giant food processing plant.

Q *There are times in my application when I want to perform the same operation on a whole bunch of soup entries. Right now the Xmit form of the calls takes quite a while and results in a lot of messages flying around. Is there a better way to do this?*

A Yes. You can use **nil** for the argument that tells the system which application is performing the change. This tells the system not to transmit the change notification. Then you can use XmitSoupChange to send a **whatThe** notification. This will inform other applications that something changed, but not give specifics of the change. As an example, here's a code snippet to remove all entries in a given cursor:

```
// create a function we can map over
local myRemoveFunction := func(entry) EntryRemoveFromSoupXmit(entry, nil);

// remove the entries
MapCursor(removeCursor, myRemoveFunction);

// now inform registered soups that something has changed
XmitSoupChange(kSoupName, kAppSymbol, 'whatThe, nil);
```

Q *We're having a problem with compiled NewtonScript code. We have a function that takes an **int** as a parameter. We use the **int** type indicator in the function definition. However, it's possible for the parameter to be **nil**. For compiled code this results in a type mismatch error. Is there a workaround for this?*

A There is a way to work around this problem; however, you'll pay a performance penalty. It's much better to redesign your API to accept only integers. If you do want a workaround, you can use the type-checking functions to make sure the

parameter is an integer before you use it like one. Here's some code that will work:

```
func native(x) begin
   local int intx;

   if IsInteger(x) then begin
      // now you can use intx and it'll be faster than using x
      intx := x;
   end else begin
      // do whatever else is appropriate; it isn't an integer
      ...
   end;
end;
```

Q *In our application we sometimes have to show the user a big error message, which unfortunately is too large for the Notify mechanism. Is there a way we can add a button to the Notify slip? If not, is there some other mechanism we can use?*

A The answer is simple: just don't let your user generate such a large error. But seriously, there is no way to modify the slip that comes up from Notify. Here are two ways to solve your problem:

- Have the Notify message tell the user to click on some user interface element in your application for more information. This is the recommended solution.
- Instead of Notify, use a dialog to inform the user of the error. Since you control the dialog, you control which buttons are in it. You could set up such a dialog with AsyncConfirm.

Q *We need a way to find out whether a particular view inherits from a particular proto. For example:*

```
aView := {_proto: anotherView,
          // more slots...}

anotherView := {_proto: protoFloater,
                // more slots...}
```

Is there a function I could call that would take aView and return true if it's an instance of protoFloater?

A There is no built-in function that will do this, but it's relatively simple to write:

```
func(frame, prot) begin
   while (frame AND (frame <> prot))
      frame := frame._proto;
   return (frame <> nil)
end;
```

Q *How do I define a pickerDef column to be "lastname, firstname" in a single column?*

A You can specify a Get method in your pickerDef and modify your columns appropriately. As an example, take a look at the protoListPicker sample on the Newton Developer CD. One of the subsamples is called ListPickerSoup. The

default is to display the first item in the first column and the second item in the second column. The original pickerDef is defined as follows (from pickerDef.f):

```
DefConst('kMyBasicSoupDataDef, {
   _proto: protoNameRefDataDef,  // required
   validationFrame: nil,         // used if editing is supported
   name: "Random Data ",         // name at top left of picker if
                                 // foldersTabs are present
   HitItem: func(tapInfo, context) begin
      context:ThingChosen(tapInfo);
   end,
   });
```

Then in myListPicker in the listPickerSoup.t layout file, the pickerDef slot is:

```
{_proto: kMyBasicSoupDataDef,    // defined in the pickerDef.f file
   class: 'nameRef,              // always include
   columns: [
      // Column 1
      {
         fieldPath: 'first,      // field to display in column
         tapWidth: 100,          // width for checkbox & name combined,
                                 // offset from the right margin
         doRowHilite: true,
      },
      // Column 2
      {
         fieldPath: 'second,     // field to display in column
         tapWidth: 0,            // width from preceding column to
                                 // right bounds
         doRowHilite: true,
      },
   ],
}
```

To modify this sample to show one column in the format “second, first”, you would add the following Get method to kMyBasicSoupDataDef:

```
Get: func(object, fieldPath, format) begin
   if fieldPath = 'second AND
      (format = 'text OR format = 'sortText) then
   begin
      local realData := EntryFromObj(object);
      if realData then     // format is "second, first"
         return (realData.second & ", " & realData.first);
      else
         return "- -";
   end else
      inherited:?Get(object, fieldPath, format);
end
```

This Get method will return the correct string for a column that displays the slot '**second**. It will also sort on a string that’s in the format “second, first”.

The other thing you need to do is modify the columns definition. Simply remove the first column, so that the pickerDef in myListPicker looks like this:

```
{_proto: kMyBasicSoupDataDef,     // defined in the pickerDef.f file
   class: 'nameRef,               // always include
   columns:
   [
      // Column 2
      {
         fieldPath: 'second,      // slot to display in column
         tapWidth: 0,             // width from preceding column to
                                  // right bounds
         doRowHilite: true,
      },
   ],
}
```

Q *I have a pick list that takes quite a while to create. I'd like to use a weak array so that I don't have to keep creating the list. That way I get garbage collection for free. But I don't want to make the array "weak" until after the pick list has been opened by the picker so that items don't accidentally get garbage collected. How do I turn a regular array into a weak array? Or will this work at all?*

A You can't turn a regular array into a weak array; an array is one or the other. But using a weak array should work, with these minor modifications to your code: Create a slot in your picker (say myWeakArray) and initialize it to a weak array. Create your regular array of pick items as usual. Let the user pop up the picker; then in either the pickCancelledScript or the pickActionScript, set the first element of myWeakArray to the array of list items. Next time you want to construct the pick list, check for the first element of myWeakArray. If it exists, you have your pick list; if not, create a new one.

Q *I'm using one of the **newt** name views to select a name. Whenever the people picker comes up, it's viewing "All Names." How can I programmatically change the default folder used by the picker?*

A You need to change the Picker method of the **newt** name flavor as follows:

```
Picker: func()
   protoPeoplePopup:New('|nameRef.peoplel, nil, self,
      {labelsFilter: <symbol-for-desired-folder>});
```

The final argument to the New method is a frame of options. Each slot/value pair is used to set up a slot/value pair in the protoPeoplePicker. So grab the symbol for the default folder that you want and set the labelsFilter of the protoPeoplePopup.

Q *What is the path to true enlightenment and wisdom?*

A Simple: Buy and read all the books that claim to show you such a path. As you read, make a list of the major points. Take that list, cross off the contradictions, take the inverse of what's left, and then get a life. Alternatively, go and code another thousand lines of NewtonScript.

Thanks to jXopher Bell, Henry Cate, Bob Ebert, David Fedor, Ryan Robertson, Jim Schram, Maurice Sharp, Bruce Thompson, and Scott Wright for these answers.

If you need more answers, take a look at the Newton developer Web page, at http://www.devworld.apple.com/dev/newtondev.shtml.

Folder Fun

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. Even if you never run into the particular problems being solved here, you'll learn some valuable debugging techniques that will help you solve your own programming conundrums. And you'll also learn interesting Macintosh trivia.

KONSTANTIN OTHMER AND BRUCE LEAK

KON It's you again.

BAL It seems our bid for drumming up Puzzle Page authors hasn't gone well.

KON I guess everyone's waiting for the Mac OS 8 release. From what I hear, they'll be waiting quite a while.

BAL I've got a lot of work to do. Can we get on with this?

KON OK. Have you ever heard of Retrospect Remote?

BAL Yeah, we run it. It's that backup program. It slows your machine to a crawl when it's doing its thing, but we've never had a problem with it. Our engineers don't allow it on their machines, though.

KON Apparently someone went around one morning and installed it while we were all sleeping. It's been running great for the last year, but suddenly one of the machines started showing an execution error.

BAL Easy enough. Go check out the logs and see what it's complaining about.

100 KON It says folders are nested too deep on that machine. Sure enough, the machine has a folder called "<unknown name>," and inside that folder is another called "<unknown name>," and so on. It was mixed in with our project files.

BAL What system are you running?

KON A Power Mac 7100/66 with 32 MB of RAM and an 800 MB hard disk, running system version 7.1.2 — the "last trusted system," according to Chris Derossi.

KONSTANTIN OTHMER AND BRUCE LEAK are cashing in on the enormous popularity of the Puzzle Page by starting their own online magazine. After rejecting titles like *MacsBug Life* and *Dead Mackerel*, they've settled on *Mired* as the name for their daily diary of debugging. The first issue will include hard-hitting articles like "MacsBug: The Best Command-Line Interface for the Mac?" and "Celebrity dcmds." Watch for it soon in your local cybernews shop!•

BAL What happens when you use the Finder to get info on the folder?

90 KON It says there are 99 items inside this one for zero K of disk space.

BAL Go in a level and try the same thing. Still 99 items?

KON Yep.

BAL Go in a few more levels and rename one of the folders. Come back out and go back in. Anything unusual?

KON Nope. The renamed folder kept its new name. The Finder tells you there are 99 items inside it.

BAL Just keep opening folders. How deep does it go?

80 KON After opening folders for a few minutes, you get too bored to continue.

BAL Rename every tenth folder or so, to create some landmarks. Dig down doing this. Can you Command-click on the window title and see the whole hierarchy?

KON Yes. You're doing one heck of a test on the pop-up menu manager.

BAL So, what happens?

70 KON After you do this about 150 times, the Finder displays the familiar "out of memory" message. It says you might try closing some windows to make more memory available. I have the feeling this might be one case where that advice actually works!

BAL Do a Get Info on the hard drive. How many items does it have?

KON 16,774 items on disk.

BAL Somehow I was expecting a larger number! Try copying the folder.

KON We crash.

BAL Where? Any clues?

KON In CopyDoubler.

BAL That sounds like a subject for another column. Hold down the Control key when copying the folder to tell CopyDoubler to stay out of it.

60 KON The Finder puts up the copy dialog, waits about two minutes, advances the thermometer the entire way in about three seconds, and then waits another two minutes before finishing. The copy dialog says there are 100 items to be copied. You now have two folder hierarchies.

BAL That's interesting. I've seen the Finder copy way more than 100 items at one time, but somehow its knowledge of folder nesting is limited to 99 or 100. Do a Get Info on the drive again and see how many new items were created.

KON Now there are 18,679 items on disk — 1,905 new ones.

BAL Copy it to a file server.

55 KON OK. Now the Finder does the copy the same as before, except after the thermometer is finished, the dialog stays up, waiting for a long time.

BAL How long?

KON Suppose we go to a long dinner at the Peppermill Lounge. When we come back it still won't be done.

BAL Look at the folder hierarchy on the network from another Mac.

KON While it's still copying?

BAL Sure.

50 KON You see a bunch of folders, as you'd expect, except the network is really slow. Apparently the Finder is generating a lot of network traffic trying to create all those directories.

BAL Stop that file copy.

45 KON The Stop button in the copy dialog is highlighted (indicating that you pressed it), and the network is back to normal, but the copy dialog doesn't go away.

BAL Reboot and look at the folders in Standard File.

KON No new clues. How deep do you want do go?

BAL Hmm. Try running Norton Utilities on the disk.

KON There were a few bundle bits that are wrong. But once those are fixed, Norton says the disk is fine.

BAL Try DropStuffing the folder.

KON DropStuff crashes.

BAL Try Disk First Aid.

40 KON It tells you that the folder nesting exceeds the Finder-recommended nesting of 100. But other than that, it says the disk is fine.

BAL I guess those folders are really there. The system is running into a bunch of boundary conditions trying to deal with them. The question is how they got there, and I'm not sure how we figure that out given that all we have is a smoking gun.

KON Wait a second…the column isn't supposed to end this way!

BAL OK. I've got an idea. I'll try looking at these folders from MPW with a **files -r -c** command.

35 KON MPW prints out a bunch of folders and then crashes. The interesting thing is that it crashed right after printing out the 100th level down, which we know from the "rename every tenth folder" exercise we did earlier.

BAL Well, there are some mysteries that we should be able to solve. Clearly, some software on this machine created the folder called "<unknown name>," probably when a directory with a null name or some other exception condition was encountered.

KON OK.

BAL It's probably the Finder, the File Manager, or one of the applications that's commonly used on the machine. Let's ask Paul Mercer if he knows anything about it.

30 Paul First of all, the Finder isn't going to rename a folder on you. And second, even if it did, no one on the Finder team would give the folder a name that contains an angle bracket — that's just too unpleasant aesthetically. Maybe the File Manager would do such a thing, though.

KON We could try Dave Feldman. I'm not sure he'd have the same issues with angle brackets that Paul has.

25 Dave No issue with angle brackets here! But the File Manager doesn't ever attempt to detect or fix damaged catalog info. It will rebuild the volume bitmap (that long pause when remounting a disk after a system crash), but it leaves everything else alone. No matter how much it's abused by the Finder, the File Manager will never surreptitiously set a folder to "<unknown name>." Tell Paul I said what does he know, he's been gone from Apple long enough to have forgotten what little he once knew about the File Manager. Furthermore, the File Manager doesn't care how deep you nest folders; that's a Finder problem.

BAL Well, the system hasn't changed much since either of those guys left Apple, so it's probably something else. Maybe you can get someone at Apple to search all the system sources and see if they can come up with a hit on "unknown name."

20 KON I talked to Jim Luther about it and he said it shows up in only one place — the Alias Manager. Apparently the Alias Manager will store the name of the user who created the alias as part of the alias record. If a user who logs on as a guest creates an alias, the name of the alias creator is set to "unknown name," but there are no angle brackets.

BAL Let's grep the hard disk and see if we can find any hits on "<unknown name>."

KON How do you propose we do that?

BAL Have you seen AltaVista?

KON You mean the search engine on the Internet? Totally awesome. When you do a query they can instantly give you a list of the top hits from any Web site. But what does that have to do with this puzzle?

BAL Well, I figure they can search huge amounts of data way faster than we could ever do it on our local machine. So we dump the entire contents of the hard disk to a Web page, register it with AltaVista, and perform the search.

KON A little unrealistic, but not a bad idea.

BAL Ron Avitzur thought it up.

KON For those keeping score, you're approaching 15.

BAL OK, OK. I'll use Norton Disk Editor and search the whole volume for the string "<unknown name>."

15 KON The first hit is in the catalog. I get the feeling you're going to find at least 1905 of these on this disk, probably more since you duplicated the folders so many times.

BAL OK. Try it on one of your other development machines.

10 KON The string is found in a bunch of places but the sectors aren't owned by a file. But then the needle in the haystack pokes your probing finger — you find the string in the MPW Shell application.

BAL Open up MPW Shell with ResEdit and find out which resource it's in.

KON Duh! Wrong tool for the job! ResEdit can't search the entire resource fork.

BAL OK. Use Resorcerer.

KON You find the string in CODE resource 27, called %A5Init.

BAL So, it sounds like we should contact someone in MPW land and see if they know what's going on. Here's Alex McKale; maybe he can help us.

5 Alex Sure enough, Projector will create folders with the name "<unknown name>." CheckOutDir creates a folder hierarchy that matches the project hierarchy.

BAL The project hierarchy on this machine is only three or four levels deep, not 1905! Any explanation for that?

Alex Did the machine ever crash while checking out? Maybe some script got in an infinite loop, and you hit the reset button or crashed after some time. This would mean the depth of the hierarchy is based on how long the machine was running in the loop, rather than on some magic number, such as everyone's favorite year plus 1.

KON It happened on Richard's machine. He says his Mac crashes all the time, and he'd be hard pressed to tell you what it was doing during any particular crash.

BAL Hmm. I guess we just need a plausible explanation. Any ideas, Alex?

Alex Your guess is as good as mine. CheckOutDir does very little error checking, so if the project tree got munged — for example, if there was no terminator in the project folder hierarchy — it would keep creating folders called "<unknown name>" until it hit a terminator. Give me a reproducible case, and this thing is dead meat.

KON No can do. I guess the exact cause will remain a mystery, along with the true nature of consciousness, the details of Jimmy Hoffa's demise, and the location of my other red sock. Well, at least we managed to narrow it down to Projector, so we can point both our accusatory fingers in the same direction. If nothing else, maybe this will get the Projector folks to add a little more error checking to CheckOutDir.

BAL Nasty.

KON Yeah.

SCORING

70–100 Congratulations! You win free lifetime upgrades to MPW.
45–60 You win a free copy of Mac OS 8.
25–40 You win a lifetime subscription to eWorld.
5–20 You win a dinner with Paul and Dave.•

Thanks to Chris Derossi, Dave Feldman, Jim Luther, Alex McKale, and Paul Mercer for reviewing this column.•

INDEX

For a cumulative index to all issues of *develop*, see this issue's CD.•

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

If you don't tell us, who will?

We encourage your questions, suggestions, kind words, or even gripes about *develop*. Please drop us a line and let us know what's on your mind.

Send editorial comments to develop@apple.com or to:

Caroline Rose
Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
crose@apple.com
Fax: (408)974-0544

Send technical questions about *develop* to:

Dave Johnson
Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
dkj@apple.com
Fax: (408)974-0544

If you have a subcription-related query, please contact Apple Developer Catalog at order.adc@applelink.apple.com, or call them at 1-800-282-2732 in the U.S., 1-800-637-0029 in Canada, or (716)871-6555 elsewhere.

RESOURCES

Apple provides a wealth of information, products, and services to assist developers. The Apple Developer Catalog and Apple Developer University are open to anyone who wants access to development tools and instruction. Additional information and services are available through Apple's Developer Programs.

The Apple Developer Catalog offers worldwide access to development tools, resources, training products, and information for anyone interested in developing applications on Apple platforms. This complimentary catalog features hundreds of Apple and third-party development products and offers convenient payment and shipping options, including site licensing.

Apple Developer University (DU) provides courses to get you started programming on Apple platforms, as well as advanced, in-depth training on new technologies such as QuickTime VR, QuickDraw 3D, OpenDoc, Apple Guide, and Newton.

In addition to classroom training, self-paced courses are available through the *Apple Developer Catalog*, and free introductory tutorials are provided on the DU Web site, at http://www.devworld.apple.com/dev/du.shtml.

The Macintosh Developer Program provides members with ongoing Macintosh-related technical information and services. It includes:

- The monthly Apple Developer Mailing, which includes the *Developer CD Series*.
- Macintosh technology seeding.
- Programming-level technical support via e-mail. Apple offers a number of options for varying levels of technical support.

The Newton Developer Program provides ongoing Newton-related technical information and services. It includes:

- The monthly Newton Developer Mailing.
- The quarterly Newton Developer CD.
- Newton development class discounts.
- Programming-level technical support via e-mail. Apple offers a number of options for varying levels of technical support.

The Apple Media Program (AMP) provides resources to keep multimedia developers up-to-date on Apple's offerings for authoring and playback. For more about the benefits and resources of this program, visit the AMP Web site at http://www.amp.apple.com.

Apple Developer Catalog To order a product or receive a catalog, call 1-800-282-2732 in the U.S., 1-800-637-0029 in Canada, (716)871-6555 internationally, or (716)871-6511 for fax. You can also send e-mail to order.adc@applelink.apple.com, or write *Apple Developer Catalog*, P.O. Box 319, Buffalo, NY 14207-0319. The *Apple Developer Catalog* is also on the Web at http://www.devcatalog.apple.com.

Apple Developer University Course descriptions and schedules can be found at http://www.devworld.apple.com/dev/du.shtml on the Web. You can also call (408)974-4897, fax (408)974-0544, send e-mail to devuniv@apple.com, or write Developer University, Apple Computer, Inc., 1 Infinite Loop, M/S 305-1TU, Cupertino, CA 95014.

Apple Developer Programs These programs vary on a country-by-country basis. For more information on any of Apple's developer support programs worldwide, call (408)974-4897, fax (408)974-7683, or send e-mail to devsupport@apple.com. Also see the Developer Programs Web site at http://www.devworld.apple.com/dev/programs.shtml.