GX Breaks the Space Hack • Display Manager • Using ToolServer From CodeWarrior

develop

The Apple Technical Journal

Issue 25 March 1996

Generating QuickTime VR Movies From QuickDraw 3D

Flicker-Free Drawing With QuickDraw GX

NURB Curves: A Guide for the Uninitiated

Using C++ Exceptions in C

Country Stringing: Localized Strings for the Newton

$10.00

develop

EDITORIAL STAFF

Editor-in-Cheek *Caroline Rose*

Managing Editor *Toni Moccia*

Technical Buckstopper *Dave Johnson*

Bookmark CD Leader *Alex Dosher*

Able Assistant *Meredith Best*

Our Boss *Mark Bloomquist*

His Boss *Dennis Matthews*

Review Board *Brian Bechtel*, *Dave Radcliffe*, *Jim Reekes*, *Bryan K. "Beaker" Ressler*, *Larry Rosenstein*, *Andy Shebanow*, *Nick Thompson*, *Gregg Williams*

Contributing Editors *Lorraine Anderson*, *Steve Chernicoff*, *Linda Fogel*, *Toni Haskell*, *Judy Helfand*, *Cheryl Potter*, *Joan Stigliani*

Indexer *Marc Savage*

ART & PRODUCTION

Art Direction *Lisa Ferdinandsen*

Technical Illustration *John Ryan*, *Laurie Wigham*

Formatting *Forbes Mill Press*

Production *Diane Wilcox*

Photography *Sharon Beals*, *Naomi Chesler*

Cover Illustration *Graham Metcalfe of Metcalfe/Shubert Design*

ISSN #1047-0735.

Printed on recycled paper

THINGS TO KNOW

develop, The Apple Technical Journal, a quarterly publication of Apple Computer's Developer Press group, is published in March, June, September, and December. *develop* articles and code have been reviewed for robustness by Apple engineers.

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 monthly *Developer CD Series*, available through the Apple Developer Catalog. Included on the CD are this issue and all back issues of *develop* along with the code that the articles describe. (The code is updated periodically, so always use the most recent CD.) The CD also contains Technical Notes, 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 on either the Bookmark CD or the Reference Library or Tool Chest edition of the *Developer CD Series*. Much of the CD contents, including the *develop* issues and code, are also available in the Developer Services area on AppleLink and at ftp.info.apple.com. See also the World Wide Web site for Apple Developer Services and Products, at http://dev.info.apple.com.

Macintosh Technical Notes. A designation like "(QT 4)" after a reference to a Macintosh Technical Note or Macintosh Technical Q&A in *develop* indicates the category and number of the Note on this issue's CD. (QT is the QuickTime category.) The new (uncategorized) Technotes are designated by number alone.

E-mail addresses. Many e-mail addresses that are mentioned in *develop* are AppleLink addresses. On the Internet, AppleLink address XXX translates to xxx@applelink.apple.com. NewtonMail address XXX translates to xxx@online.apple.com.

CONTACTING US

Feedback. Send editorial comments or suggestions to Caroline Rose at AppleLink CROSE, Internet crose@applelink.apple.com, or fax (408)974-6395. Send technical questions about *develop* to Dave Johnson at AppleLink JOHNSON.DK, Internet dkj@apple.com, CompuServe 75300,715, or fax (408)974-6395. Or write to Caroline or Dave at Apple Computer, Inc., 1 Infinite Loop, Cupertino, CA 95014.

Article submissions. Ask for our Author's Guidelines and a submission form at AppleLink DEVELOP, Internet develop@applelink.apple.com, or fax (408)974-6395. Or write to Caroline Rose at the above address.

Subscriptions and back issues. You can subscribe to *develop* through the Apple Developer Catalog (see ordering information below) or use the subscription card in this issue. You can also order printed back issues through the catalog. The one-year U.S. subscription price is $30 (for 4 issues and 4 *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).

Apple Developer Catalog. To order *develop* or other products through the catalog, or to make subscription-related queries, 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 send e-mail to AppleLink APDA, Internet apda@applelink.apple.com, America Online APDAorder, or CompuServe 76666,2405. Or write Apple Developer Catalog, Apple Computer, Inc., P.O. Box 319, Buffalo, NY 14207-0319. For all subscription changes or queries, *please be sure to include your name, address, and account number as it appears on your mailing label.*

ARTICLES

COLUMNS

EDITOR'S NOTE

CAROLINE ROSE

I'd like to make some general comments on user interface that I've been collecting for a while. Please understand I don't claim to be an expert on the subject — and, as always, the usual disclaimer applies that my opinions aren't necessarily those of Apple Computer the corporate entity. I'm just one person who has the luxury (as well as the burden) of filling an editorial page, and this is what's on my mind.

First I'd like to make a pitch for a more obvious and responsive channel for users to give feedback on the experience of working with your software: What bugs them about it? Do they have suggestions for improvement? It seems that such feedback gets lost if it's delivered through the usual customer support (bug-reporting) vehicle. Or maybe it gets delivered to the wrong person, like the engineer who designed and implemented that feature in the first place and can give 99 excellent reasons for why it was done that way. When I first returned to Apple from NeXT and had to use a lot of new applications, I encountered a number of interface glitches that were short of being bugs but made using the applications unnecessarily awkward. I suggested a few simple fixes through the customer support line, sensing some interest but the assignment of a low priority. Alas, those annoying features are all still there five years and a couple of upgrades later.

This ties in with a peeve that I share with many of my computer-using friends who don't work in this industry and so have that valuable perspective of a pure end user: Please avoid making spurious interface "improvements" that change the basic way I work with your application. Be sure there's a real benefit to the user before you change command names, rearrange them in menus, or put new features in my face rather than making them options I can explore at will. You may think you're making the interface friendlier, but in fact you may be alienating your existing customer base. The changes that I most appreciate are those that smooth out the rough spots of the interface as it is, so that, for example, I won't have to resize every mail window to fit my screen, or select the text in the Find box every time I do a new search. I want these nuisances eliminated; instead, I find the upgrade to be just another big nuisance.

We interrupt this list of grievances to announce another list we just learned about: *develop* has been chosen by Internet Valley, Inc. as one of the top 100 computer-related magazines and journals on the Web (http://www.internetvalley.com/top100mag.html). I'm happy enough about this to quit my griping about user interface for now — though I do see a tie-in here: we've always encouraged and responded to comments from our readers, and we've tweaked our content and format accordingly rather than done a complete overhaul. Thanks for all the valuable feedback over the years; please keep it coming so that we can keep improving.

Caroline Rose
Editor

CAROLINE ROSE (AppleLink CROSE) has been a technical editor for so long that she says she can do it with her eyes closed. And that's exactly what she did after being felled by a detached retina during her last vacation. Lying down for over two weeks with her lids shut and not much to do except listen to books on tape, she welcomed the occasional phone call on a *develop*-related editorial question. Getting back to work and realizing how much catching up she had to do was a real eye-opening experience.•

LETTERS

WEB FIRST, THEN PRINTED COPY

develop is absolutely the coolest publication for a Mac developer. I thought I would drop a line to say "thanks" for putting the next issue up on the Web a full month before it will arrive at my home. At least this way I can get a partial fix!

Good job!

— Rob Newberry

I just noticed that you've released *develop* Issue 24 online. I'm a subscriber, yet I have to call or send e-mail to you each time to remind you to send my issue!

Your magazine is terrific, but the service is quite the opposite.

— Carl Limisco

As a service to developers who may want access to content as soon as it's finalized, develop *content is uploaded to the Web within three days of issue completion. The print and CD-ROM production processes, however, consume more time and thus result in the delay between when you may first see content on the Web and when you receive your copy with its CD in the mail.*

In the case of Issue 24, this period was extended due to technical difficulties with generating the mailing information. Starting with Issue 23, we switched to APDA for distribution of develop. *There have been a few snags in the transition, but we're confident that subscribers will experience improved service. Meanwhile, we apologize for any problems.*

— Diane Wilcox

PUZZLE PAGE SLIP-UP

When I received *develop* Issue 24, I was shocked to find a bug in the Puzzle Page. When BAL is explaining how LockPixels and UnlockPixels work, he mentions that the PixMap baseAddr can be either a handle or a pointer, and that a flag in rowBytes identifies which state the baseAddr is in. This is wrong; that information is stored in the pmVersion field of the PixMap. There aren't any bits to spare in rowBytes.

Other than that, it was a great Puzzle Page, as usual.

— Cameron Esfahani

You're right; you caught this slip-up by the puzzlemeisters themselves. Say, if you're so good, why not write your own Puzzle Page? [Readers: See the puzzle Cameron coauthors in this issue.]

— Caroline Rose

MULTIPANE FIXES — AND ABOUT USING OUR CODE

The code accompanying Norman Franke's article on multipane dialogs (*develop* Issue 23) is great. I had it up and running in a PowerPlant application in less than an hour. But I found some bugs; for example, in the routines T2PMPDAction and friends, you lock down theData, and I suspect you should be locking down tmpData. Before I get down and dirty, I was wondering if you knew of any other bugs already present.

Also, the code needs an extra routine to generate the data handle without displaying the dialog so that one might set the initial values (as opposed to using factory defaults).

SEND US *YOUR* EXCUSE FOR NOT WRITING

Well, actually, we'd rather receive letters regarding articles published in *develop*. Letters should be addressed to Caroline Rose — or, if technical *develop*-related questions, to Dave Johnson — at AppleLink CROSE or JOHNSON.DK (Internet crose@applelink.apple.com or 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). Please send all subscription-related queries to AppleLink APDA (Internet apda@applelink.apple.com).•

When I get the PowerPlant classes working and debugged, I'd like to distribute them on the Internet (free). May I include your code (possibly modified)?

Again, thanks for a great article.

— Gordon Watts

Norman has provided a newer version of his code as of Issue 24's CD. He's fixed a lot of bugs and also now provides PowerPlant classes; see the README file for details.

You may redistribute the MPDialogs source if you like, as long as it's part of your own thing and not just a redistribution of the original package. For instance, you'll probably not be distributing Norman's sample or its source, but just the files MPDialogs.c and MPDialogs.h. Please include a pointer to where they came from, since presumably the code will change over time (bug fixes and so on).

By the way, you can contact Norman directly at franke@eworld.com.

— Dave Johnson

TECHNOTES AND Q&AS: BETTER THAN EVER

The observant among you will notice a change in Technical Notes on this issue's CD (and on the World Wide Web and the other myriad places where they can be found). The old Macintosh Technical Notes are still around, but now there are also new Notes, going simply by the name "Technotes." The old Notes will eventually evolve into the new scheme. We talked with Technote leader Tom Maremaa, from Apple's Developer Technical Support group, for the scoop on this.

"The old Notes have a rich and varied history at Apple, and have served developers well in the past," Tom said. "We wanted to continue that tradition — but with changes, something on the order of *Technotes: The Next Generation*."

First, Tom hopes you'll agree that the biggest improvement is in the quality of the new Technotes. They receive far more review by Apple engineers than the old Notes did, and they're better edited and formatted, so you should find them a lot more readable and reliable. Technotes will also be timelier: more of them will focus on hot new technologies, such as QuickTime VR and QuickDraw 3D, with updates and additions posted regularly on the Web at http://dev.info.apple.com/technotes/Main.html. They'll migrate to *develop*'s CD and other such locations, but you'll no longer have to wait that long for the latest and greatest information.

You'll notice that Technotes are numbered sequentially, starting from 1001, rather than divided into functional categories. Tom found that placing a Note into a single category was becoming increasingly difficult and arbitrary; often a topic would span more than one category or wouldn't quite fit into any existing category. Locating a Note on a particular subject is easier than ever thanks to the improved searching tools that are now available: you can use Acrobat's search mechanism on the CD or the excellent search facility provided on Apple's Web pages.

"Providing developers with the ability to search fast and effectively through the whole body of Technotes, particularly on the Web," said Tom, "has been a major goal in the project. It's there *now*. Check it out!"

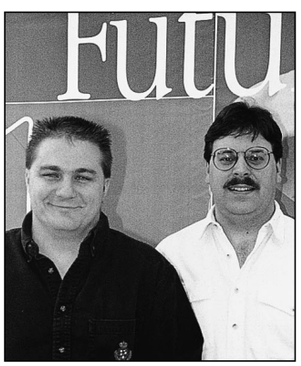
The old Macintosh Technical Notes are gradually being cleaned up: over time they'll be updated and worked into the new scheme, or deleted if obsolete. Should you look for an old Note by category and number, you'll find a "stub" indicating its current status if it's been revised or removed. In particular, the old Q&A Technical Notes are being discontinued; new Q&As are being released as "Macintosh Technical Q&As" (they're on the Web at http://dev.info.apple.com/techqa/Main.html).

For those of you who like to have Notes in printed form, you can still order a printed copy (of both the old and the new Notes). See the Technotes Web page or the latest Apple Developer Catalog for details.

Finally, Tom would like to point out that Technotes can be submitted by outside authors (although Caroline asks that you first consider whether *develop* might be a more appropriate vehicle for your handiwork :-). If your Note is published, you'll receive YATS (Yet Another T-Shirt) along with some other goodies, including a chance to participate in Apple developer kitchens and other special events. For more information, or just to let us know what you think of all these changes, write to AppleLink DEVFEEDBACK (devfeedback@applelink.apple.com).

Generating QuickTime VR Movies From QuickDraw 3D

QuickTime VR is a new technology from Apple that provides users with a virtual reality experience through interactive panoramic and object movies. You can generate images for QuickTime VR movies with either a real camera or a three-dimensional rendering system such as QuickDraw 3D. Here you'll learn how to create images from QuickDraw 3D models and generate movies from these images with the QuickTime VR Authoring Tools Suite version 1.0.



PETE FALCO AND PHILIP MCBRIDE

QuickTime VR lets you create two kinds of interactive virtual reality movies: *panoramic movies* and *object movies*. In a panoramic movie, users can interactively view a scene at nearly all camera angles from a particular point in space, which gives them the impression of being there and looking around. In an object movie, users can interactively spin an object around and thereby see it from all sides. Panoramic and object movies can be linked together or used separately.

QuickTime VR has several advantages over three-dimensional modeling systems for making interactive movies. Its movie files are much smaller than complex 3D models in situations where complete interactivity with the scene isn't necessary, or where the scene contains complex objects or large numbers of textures. With QuickTime VR, the complexity of the scene and the number of textures used are irrelevant to runtime performance, so even users with lower-end machines can effectively interact with the scene. Finally, a QuickTime VR scene needs only a few megabytes of free space in memory, much less than the enormous amount of RAM usually taken up by complex 3D scenes.

You can create QuickTime VR movies using either digitized images captured from a real camera or synthetic images generated by a 3D rendering system, such as QuickDraw 3D. In this article, you'll learn how to generate images with QuickDraw 3D and convert them to QuickTime VR movies. To make a panoramic movie, you create a panoramic image from a 3D scene, generate a linear QuickTime movie from the image, and convert the linear movie to an interactive panoramic movie using the

PETE FALCO (AppleLink FALCOP) is a member of Apple's QuickTime VR team. Since finishing school at Rensselaer Polytechnic Institute in upstate New York, where he spent the last six years in the rainy, snowy weather of Troy, he's found the sunny weather of California a welcome treat and vows he'll never leave this area. His latest projects include working on the next release of QuickTime VR as well as integrating all of Apple's multimedia technologies with QuickTime VR.•

PHILIP MCBRIDE (mcbride@apple.com) has been working on multimedia tools and the underlying media technologies since he's been at Apple. Most recently this included helping to add QuickTime VR to Apple Media Tool 2.0. While not working with digital multimedia, Philip likes to work with real multimedia by sculpting. In fact, he's been developing a new product that will involve clay, a mouse, and a bottle of cabernet. The details are sketchy, but he has a cool T-shirt for it.•

QuickTime VR Authoring Tools Suite (ATS). (The ATS is a set of tools that you use from within MPW, the Macintosh Programmer's Workshop.) To create an object movie, you generate a series of images from a 3D model, add the images to a linear QuickTime movie, and then use the QuickTime VR ATS to convert the linear movie to an interactive object movie.

Before we get into the specifics of making movies, we'll explore the basic concepts of QuickTime VR. We assume you have a general understanding of QuickDraw 3D, which you can get by reading "QuickDraw 3D: A New Dimension for Macintosh Graphics" in *develop* Issue 22 and "The Basics of QuickDraw 3D Geometries" in *develop* Issue 23. You can learn all about QuickDraw 3D in the book *3D Graphics Programming With QuickDraw 3D*.

This issue's CD contains all the code necessary to generate panoramic images and linear object movies from QuickDraw 3D models. For brevity, the listings in the article omit error handling; the code on the CD includes the complete versions of these functions.

QUICKTIME VR BASICS

The basic components of QuickTime VR movies are panoramas, nodes, objects, and scenes.

- A *panorama* is an image spanning 360° or less in a real or virtual scene. The image is viewed from a particular location in the scene, called a *node*. A single-node panoramic movie enables a user to look in all directions from that location.
- An *object* is an interactive item that can be viewed from all angles. Object movies can be linked to panoramic movies in a scriptable authoring environment, enabling a user to pick up and turn the objects from within a panorama. Object movies can also be used independently of panoramic movies.

  In our case, the object of the movie is generated from a QuickDraw 3D model, which contains a single geometric object (or group of objects); we'll use the term *model* in this article to refer to the object in a QuickTime VR object movie.
- A *scene* is a collection of several panoramas or nodes, a panorama with one or more objects, or several panoramas and objects all linked together by interactive hot spots. In a multinode scene, a user can navigate from node to node to move about the scene.

SHOOTING AN OBJECT

For object movies, you need to photograph the model (or the real object) from all directions, as shown in Figure 1. All vertical camera positions above the center of the model are considered positive, and all positions below it are considered negative. The vertical position with the camera directly above the model looking down at it is called *vertical pan 90°*; the vertical position directly below and looking up is called *vertical pan -90°*. *Vertical pan 0°* is at the model's center (equator). Horizontal positions are measured in degrees from *horizontal pan 0°* to *360°*. *Horizontal pan 0°* is typically at the back of the model.

Images must be stored as frames in row order from top to bottom in a linear QuickTime movie. For best results, we (along with the QuickTime VR documentation) recommend that you have a frame every 10° between positions in both the horizontal and vertical direction. If you shoot at increments greater than 10°, the motion of the model in the

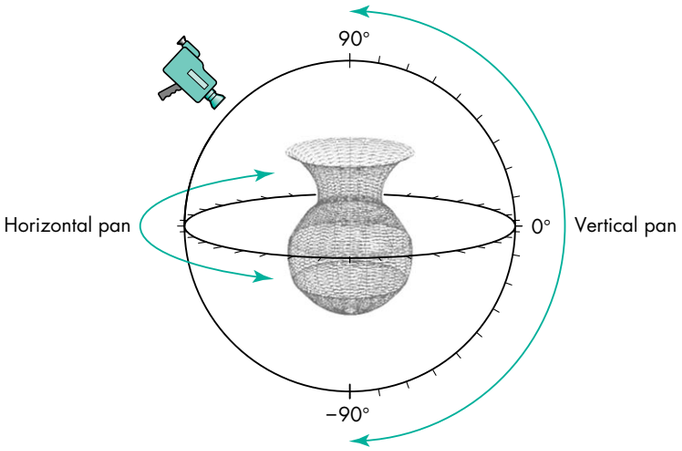


Figure 1. Shooting a model (or a real object)

QuickTime VR movie will be choppier when the user turns it. If you shoot at increments of less than 10°, the motion will be smoother, but you'll need more disk space to store all the frames. Whatever increment you choose, it should be consistent between all horizontal and vertical frames for the object and divide evenly into the horizontal and vertical pan ranges.

Your first frame at each horizontal position should be of the back of the model, so that the frame showing the front of the model is halfway through the series at that horizontal position. This improves disk access time at run time since the user will most likely be looking at the front of the model.

SHOOTING A PANORAMA

If you're using a real camera to shoot a panorama, you need to take the appropriate number of equally spaced pictures in a circle, as shown in Figure 2.

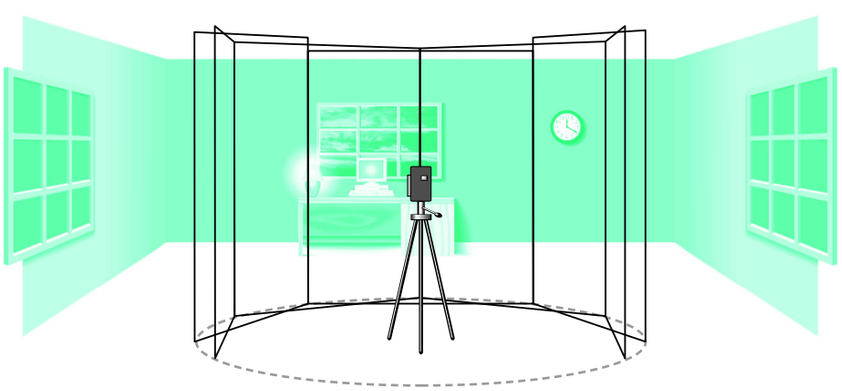


Figure 2. Shooting a panorama

Although this sounds simple, there are a few things you must be aware of. First, you need to make sure you're taking the right number of shots for the lens you're using (see Chapter 6 in Volume 1 of the QuickTime VR ATS documentation for a full explanation of this). The entire camera rig should be level at all times, and the nodal point of the lens should be directly over the point of rotation of the rig. For best

results, you should also have a consistent overlap between images; the more overlap, the better (30% to 50% is recommended). Finally, you should maintain consistency between images in each panorama by using similar exposure and a fixed focus.

Because this can get quite complicated, Apple strongly recommends the use of a professional photographer for making any production-quality titles. However, one way around this is to use rendered data, as we do in this article. The programmatic control we have over the "virtual" camera in a 3D environment such as QuickDraw 3D eliminates all of the problems just mentioned.

MAKING MOVIES WITH THE SAMPLE CODE

The sample code on this issue's CD enables you to make object and panoramic movies from any 3DMF file (a file that conforms to the QuickDraw 3D Object Metafile standard). For either type of movie, the code creates a new document record structure, reads in the model from a 3DMF file, renders the images, and writes out the images in a form that the QuickTime VR tools can work with.

Here we'll look at the first few steps, which are common to both types of movies. The other steps for making QuickTime VR movies — rendering and writing out the images and converting linear movies to interactive movies — are different for object and panoramic moviemaking and are described later.

CREATING A NEW DOCUMENT

All of our scene information is stored in a document record structure, shown in Listing 1.

Listing 1. The document record structure for a scene

```
typedef struct _DocumentRecord {
   CWindowPtr        theWindow;              // Display window
   FSSpec            theFileSpec;            // Model file specification
   short             fRefNum;                // and reference
   GWorldPtr         drawContextOffScreen;   // Offscreen buffer
   TQ3GroupObject    documentGroup;       // Main group for the document
   TQ3ViewObject     theView;             // The document's view object
   TQ3Matrix4x4      modelRotation;          // The model transform
   TQ3Point3D        documentGroupCenter;    // Center of the model
   ...   // Miscellaneous view, model, and QuickTime file details
} DocumentRecord, *DocumentPtr, **DocumentHandle;
```

The MyNewDocument function (Listing 2) creates the document record structure and sets up the view, camera, and other elements associated with the scene. It also adds the background buffer and window used to display the rendered images of the scene.

CREATING THE CAMERA

The camera used to render the images for the movies is created by the MyNewCamera function, shown in Listing 3.

For object movies, we set the field of view to approximately 30°. This is not a fixed number; you can use any number that you see fit, based generally on the aspect ratio of your viewing window and how much information you'd like to display inside it.

Listing 2. Creating a new document record structure

```
DocumentPtr MyNewDocument()
{
   DocumentPtr             theDocument;
   CWindowPtr              theWindow;
   TQ3DrawContextObject    theDrawContext;
   Rect                    myBounds = kMyBoundsRect;
   TQ3CameraObject         camera = NULL;
   RGBColor                blackColor = kMyBlackColor;
   ...

   theDocument = (DocumentPtr)NewPtrClear(sizeof(DocumentRecord));

   // Create the window for the document record and add references to
   // the document record.
   theWindow = (CWindowPtr)NewCWindow(0L, &myBounds,
         "\pRendering Window", true, documentProc, (WindowPtr)-1L,
         true, NULL);
   theDocument->theWindow = theWindow;

   // Create and set up the offscreen GWorld/context.
   // ** Notice that QuickDraw 3D prefers direct color. **
   NewGWorld(&theDocument->drawContextOffScreen, 32,
         &theWindow->portRect, nil, nil, 0L);
   ...
   SetGWorld(theDocument->drawContextOffScreen, nil);
   EraseRect(&theDocument->drawContextOffScreen->portRect);
   ...

   // Create the new pixmap draw context.
   theDrawContext = MyNewDrawContext(theDocument);

   // Create the view and set up the view attributes.
   ...
   // Initialize the model rotation and transitions used for object
   // movie rotations.
   Q3Matrix4x4_SetIdentity(&theDocument->modelRotation);

   // Add more model and view properties to the document record.
   ...
   // Create the camera and add it to the view.
   camera = MyNewCamera(theDocument->theWindow);
   Q3View_SetCamera(theDocument->theView, camera);
   Q3Object_Dispose(camera);

   // Add the renderer to the view. Set the window's GWorld.
   Q3View_SetRendererByType(theDocument->theView,
         kQ3RendererTypeInteractive);
   SetGWorld(theWindow, nil);

   return (theDocument);
}
```

Listing 3. Creating the rendering camera

```
TQ3CameraObject MyNewCamera(CWindowPtr theWindow)
{
   TQ3ViewAngleAspectCameraData perspectiveData;
   TQ3CameraObject              camera;
   // For object movies, we set the field of view to 30 degrees (or
   // 30.0*kQ3Pi/180.0 radians). For panoramic movies, we set it to
   // 74 degrees (or 74*kQ3Pi/180.0 radians). QuickDraw 3D requires
   // angles to be in radians, while QuickTime VR requires them to
   // be in degrees.
   float                        fieldOfView = 30.0*kQ3Pi/180.0;
   TQ3Status                    returnVal = kQ3Failure;

   // Assign default placement.
   perspectiveData.cameraData.placement.cameraLocation = kMyDefaultFrom;
   perspectiveData.cameraData.placement.pointOfInterest = kMyDefaultTo;
   perspectiveData.cameraData.placement.upVector = kMyDefaultUp;
   perspectiveData.cameraData.range.hither = kMyDefaultHither;
   perspectiveData.cameraData.range.yon = kMyDefaultYon;

   // Assign standard view port.
   perspectiveData.cameraData.viewPort.origin.x = -1.0;
   perspectiveData.cameraData.viewPort.origin.y = 1.0;
   perspectiveData.cameraData.viewPort.width = 2.0;
   perspectiveData.cameraData.viewPort.height = 2.0;

   perspectiveData.fov = fieldOfView;
   perspectiveData.aspectRatioXToY =
      (float) (theWindow->portRect.right - theWindow->portRect.left) /
      (float) (theWindow->portRect.bottom - theWindow->portRect.top);

   camera = Q3ViewAngleAspectCamera_New(&perspectiveData);

   return (camera);
}
```

For panoramic movies, we set the field of view to 74°. This matches the horizontal field of view of a 15mm lens for our image. We specify the horizontal rather than vertical field of view since our image is taller than it is wide (768 x 512 pixels), and QuickDraw 3D requires the field of view to be specified as that of the shorter side of the image (whether width or height). We calculate the horizontal field of view based on the size of our image and the known vertical field of view of a 15mm lens (97°, as specified in Chapter 9 of the QuickTime VR ATS documentation).

READING IN THE MODEL

For the model to be read from a 3DMF file, you must first create 3D file and storage objects associated with that file. Once they've been created, you build up the model by reading in all the drawable objects from the file and adding them to a group, as shown in Listing 4.

If the model includes any lighting, we use those lights; otherwise we create our own lighting for the model.

Listing 4. Reading in the model

```
TQ3Status MyReadScene(TQ3FileObject file, DocumentPtr theDocument)
{
   TQ3Object         object;
   TQ3Boolean        isEOF;
   TQ3ViewObject     view;
   TQ3Object         model;
   TQ3GroupObject    lightGroup = NULL;

   // Create the new model and get the view.
   model = Q3DisplayGroup_New();
   theDocument->documentGroup = model;
   view = theDocument->theView;

   // Collect all drawable objects (into the model) and collect any
   // lights (into the lightGroup).
   while ((isEOF = Q3File_IsEndOfFile(file)) == kQ3False) {
      object = Q3File_ReadObject(file);

      if (Q3Object_IsDrawable(object))
         Q3Group_AddObject(model, object);

      if (Q3Object_IsType(object, kQ3SharedTypeViewHints))
         if (view)
            Q3ViewHints_GetLightGroup((TQ3ViewHintsObject)object,
                  &lightGroup);

      if (object != NULL)
         Q3Object_Dispose(object);
   }

   // Add any lights found to the view. Otherwise create default lights.
   if (lightGroup) {
      Q3View_SetLightGroup(view, lightGroup);
      Q3Object_Dispose(lightGroup);
   }
   else
      MyNewLights(theDocument);

   Q3File_Close(file);
   return kQ3Success;
}
```

GETTING THE DIMENSIONS OF THE MODEL

We must know the dimensions of the entire model as well as its center in order to place the camera in its initial position and to guide both camera and model transformations. You obtain the dimensions and center of an already constructed model by getting the model's *bounding sphere* with the function MyGetBoundingSphere (Listing 5). The bounding sphere is another 3D object that fully surrounds the model and has as its center the exact center of the model.

For object movies, the bounding sphere has an additional purpose. Although a 3D model from a QuickDraw 3DMF file may contain more than one geometric object,

Listing 5. Getting the model's bounding sphere

```
void MyGetBoundingSphere(TQ3ViewObject viewObject, TQ3GroupObject
      mainGroup, TQ3BoundingSphere *viewBSphere)
{
   TQ3Status      status;

   Q3View_StartBoundingSphere(viewObject, kQ3ComputeBoundsApproximate);
   do {
      status = Q3DisplayGroup_Submit(mainGroup, viewObject);
   } while (Q3View_EndBoundingSphere(viewObject, viewBSphere) ==
         kQ3ViewStatusRetraverse);
}
```

a QuickTime VR object movie has only one geometric object or one group of objects. Thus, we use the bounding sphere to get the dimensions of the entire group of objects.

MAKING A QUICKTIME VR OBJECT MOVIE

Now we'll get into the specifics of making a QuickTime VR object movie. The MyConvert3DMFToObject function (shown in Listing 6) drives the entire process, from creating the new document to generating the linear object movie. You use the QuickTime VR ATS to generate an interactive object movie from this linear movie.

Listing 6. Converting 3DMF files to linear object movies

```
void MyConvert3DMFToObject(FSSpec *myFSS)
{
   DocumentPtr    theDocument;

   // Create the document record and make the view, camera, lights,
   // window, and so on.
   theDocument = MyNewDocument();

   // Read in the model and add it to the document record's group.
   MyOpenFile(myFSS);

   // Set up the initial camera position.
   MyInitObjCamera(theDocument);

   // Draw initial view to the screen.
   MyDrawOffScreen(theDocument);
   MyDrawOnScreen(theDocument);

   // Assign the codec type.
   theDocument->theCodecType = kMyCodec;

   // Generate all the images and add them to the movie.
   MyGenerateObjImages(theDocument, 36, 19, 360, 0, 90, -90);

   ...
}
```

DOING THE MODEL AND CAMERA WORK

Photographing a real object involves using a spherical camera rig to rotate a camera around the object. For 3D models, it's just as easy to rotate the model in front of a stationary camera. Furthermore, since the camera doesn't move in this case, the lighting is easier to manage because it doesn't need to be rotated with the camera (unless you want the object to appear to be lighted from a certain angle).

In our case, we'll render images of the model by rotating the model around two of its axes while the camera views it from the third axis; thus the camera gets a view of the model from every angle. In our case, we'll place the camera along the *z* axis and rotate the model around the *x* and *y* axes. The initial positions of the camera and the model can be seen in Figure 3.

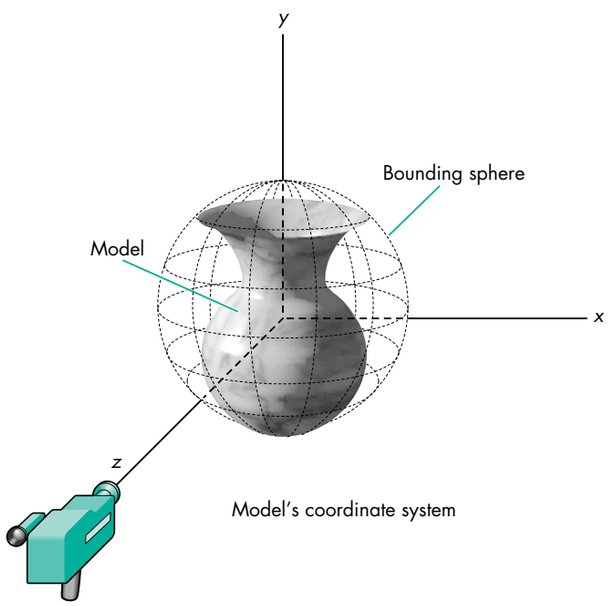


Figure 3. Initial positions of the camera and the model for object movies

The initial placement of the camera for an object movie is performed by the function MyInitObjCamera, shown in Listing 7. First we get the bounding sphere of the drawable group of the model. From this we can get the center 3D point of the drawable group (as the origin of the bounding sphere) and the radius. From the center point we place the camera a distance of five times the radius down the z axis from the object.

We rotate the model by repeatedly modifying the model's transform object. The function MyRotateObjectX rotates the object around its *x* axis (see Listing 8). An analogous function that's not shown here, MyRotateObjectY, rotates it around the *y* axis.

We step through the model rotations to create the images needed for the linear object movie with the MyGenerateObjImages function, shown in Listing 9. In this function, we iterate over *y* angles (rotating around the *x* axis) while iterating over *x* angles (rotating around the *y* axis). This stepping takes us from the position yAngle = maxVPan, xAngle = minHPan to the position yAngle = minVPan, xAngle = maxHPan – minHPan. At each step in the *x* and *y* angles, the model is

Listing 7. Setting the initial camera position for an object movie

```
TQ3Point3D MyInitObjCamera(DocumentPtr theDocument)
{
   TQ3BoundingSphere    viewBSphere;
   float                viewRadius;

   // Get the bounding sphere of the drawable group (the entire model)
   // in the view.
   MyGetBoundingSphere(theDocument->theView, theDocument->documentGroup,
         &viewBSphere);

   // Get the bounding sphere's center and radius.
   theDocument->documentGroupCenter = viewBSphere.origin;
   viewRadius = viewBSphere.radius;

   // Position the camera down the z axis from the bounding sphere based
   // on the center and the radius.
   MyPlaceCamera(theDocument, theDocument->documentGroupCenter.x,
         theDocument->documentGroupCenter.y,
         theDocument->documentGroupCenter.z +
               kMyDistanceFactor*viewRadius + 1.0,
         theDocument->documentGroupCenter.x,
         theDocument->documentGroupCenter.y,
         theDocument->documentGroupCenter.z +
               kMyDistanceFactor*viewRadius);

   return (theDocument->documentGroupCenter);
}
```

Listing 8. Rotating the model for object rendering

```
void MyRotateObjectX(DocumentPtr theDocument, float angle)
{
   TQ3Matrix4x4      tempMatrix;

   Q3Matrix4x4_SetTranslate(&tempMatrix,
         -theDocument->documentGroupCenter.x,
         -theDocument->documentGroupCenter.y,
         -theDocument->documentGroupCenter.z);
   Q3Matrix4x4_Multiply(&theDocument->modelRotation, &tempMatrix,
         &theDocument->modelRotation);

   Q3Matrix4x4_SetRotate_XYZ(&tempMatrix, angle, 0.0, 0.0);
   Q3Matrix4x4_Multiply(&theDocument->modelRotation, &tempMatrix,
         &theDocument->modelRotation);

   Q3Matrix4x4_SetTranslate(&tempMatrix,
         theDocument->documentGroupCenter.x,
         theDocument->documentGroupCenter.y,
         theDocument->documentGroupCenter.z);
   Q3Matrix4x4_Multiply(&theDocument->modelRotation, &tempMatrix,
         &theDocument->modelRotation);
}
```

Listing 9. Generating images for the linear object movie

```
void MyGenerateObjImages(DocumentPtr theDocument, short rows, short
      columns, long maxHPan, long minHPan, long maxVPan, long minVPan)
{
   float    xStart, yStart, xStep, yStep, xAngle, yAngle;

   // Assign stepping angles.
   yStep = ((float)(maxVPan-minVPan))/(float)(rows-1);
   xStep = ((float)(maxHPan-minHPan))/(float)(columns-1);

   MyPrepareDestMovie(theDocument);

   for (yAngle = maxVPan; yAngle >= minVPan; yAngle -= yStep) {
      for (xAngle = 0; xAngle <= maxHPan-minHPan; xAngle += xStep) {
         // Rotate the object to the correct position.
         xStart = (-kQ3Pi*(xAngle -
                  ((float)(maxHPan-minHPan))/2.0))/180.0;
         yStart = kQ3Pi*((float)yAngle)/180.0;
         MyRotateObjectY(theDocument, xStart);
         MyRotateObjectX(theDocument, yStart);

         // Render the model (to get a PixMap image).
         MyDrawOffScreen(theDocument);
         MyDrawOnScreen(theDocument);

         // Add the rendered PixMap image to the movie.
         MyAddImageToMovie(theDocument);

         // Undo the rotation.
         MyRotateObjectX(theDocument, -yStart);
         MyRotateObjectY(theDocument, -xStart);
      }
   }
   MyCloseDestMovie(theDocument);
}
```

rotated and then rendered. The resulting images are added to a previously created movie, as described in the next section. Note that QuickTime VR likes angles in degrees and QuickDraw 3D likes angles in radians, so we have to do these conversions.

CONSTRUCTING THE LINEAR OBJECT MOVIE

The destination movie is an ordinary QuickTime movie. The movie file, track, and track media need to be set up before rendered images can be added to the movie. The images are added as frames to a track. (See *Inside Macintosh: QuickTime* for more details about this process.) The movie is constructed in the function MyPrepareDestMovie, shown in Listing 10.

We add the rendered images to the movie with the QuickTime function AddMediaSample, which is called from within the function MyAddImageToMovie (Listing 11). MyAddImageToMovie is called after each model rendering, as seen earlier in the MyGenerateObjImages function.

Listing 10. Creating the destination linear object movie

```
OSErr MyPrepareDestMovie(DocumentPtr theDocument)
{
   long              keyFrameRate, compressedFrameSize;
   short             frameRate, width, height, i;
   CodecComponent    theCodec;
   FSSpec            theFSSpec;
   TimeScale         dstTimeScale;
   TimeValue         duration;
   CodecQ            spatialQuality;
   Str255            movieName = "\pObjectMovie";

   keyFrameRate = 1;       // Every frame must be a key frame. If
                           // not, we'll get garbage around the edges
                           // of our objects when we rotate them.
                           // QuickTime refreshes only key frames
                           // completely.
   theCodec = anyCodec;                // We'll use what's there
   spatialQuality = codecHighQuality; // and make it look pretty.
   frameRate = 10;         // This can be any value.
   dstTimeScale = 600;     // This can be any value that's a multiple
                           // of frameRate.
   duration = dstTimeScale/frameRate;

   width = theDocument->theWindow->portRect.right -
           theDocument->theWindow->portRect.left;
   height = theDocument->theWindow->portRect.bottom -
            theDocument->theWindow->portRect.top;

   theFSSpec = theDocument->theFileSpec;
   BlockMove(movieName, theFSSpec.name, movieName[0]+1);

   CreateMovieFile(&theFSSpec, creatorType, 0,
         createMovieFileDeleteCurFile, &theDocument->dstMovieRefNum,
         &theDocument->dstMovie);
   theDocument->dstTrack = NewMovieTrack(theDocument->dstMovie,
         ((long)width) << 16, ((long)height) << 16, 0);
   theDocument->dstMedia = NewTrackMedia(theDocument->dstTrack,
         VideoMediaType, dstTimeScale, 0, 0);
   BeginMediaEdits(theDocument->dstMedia);

   GetMaxCompressionSize(theDocument->drawContextOffScreen->portPixMap,
         &theDocument->drawContextOffScreen->portRect, 32,
         spatialQuality, theDocument->theCodecType, theCodec,
         &compressedFrameSize);
   theDocument->compressedData = NewHandle(compressedFrameSize);
   theDocument->compressedDataPtr =
         StripAddress(*(theDocument->compressedData));
   HLock(theDocument->compressedData);
   theDocument->idh = (ImageDescriptionHandle)NewHandle(4);
   ...
}
```

Listing 11. Adding images to the linear object movie

```
OSErr MyAddImageToMovie(DocumentPtr theDocument)
{
   CodecQ      spatialQuality;
   TimeValue   duration, currentTime;

   spatialQuality = codecHighQuality;
   duration = 60;

   LockPixels(theDocument->drawContextOffScreen->portPixMap);
   CompressImage(theDocument->drawContextOffScreen->portPixMap,
         &theDocument->drawContextOffScreen->portRect, spatialQuality,
         theDocument->theCodecType, theDocument->idh,
         theDocument->compressedDataPtr);
   UnlockPixels(theDocument->drawContextOffScreen->portPixMap);

   AddMediaSample(theDocument->dstMedia, theDocument->compressedData, 0,
         (**(theDocument->idh)).dataSize, duration,
         (SampleDescriptionHandle)theDocument->idh, 1, 0, &currentTime);
   ...
}
```

GENERATING THE INTERACTIVE OBJECT MOVIE

To generate the interactive object movie, open the linear movie you just created with the Navigable Movie Player application (in the QuickTime VR ATS) and choose the Add Navigable Data menu item. This brings up the dialog shown in Figure 4. Fill in the fields with the values shown and click OK to change the linear movie to an interactive movie. Turn the model to the position you want it to be in at the beginning of the interactive movie, choose the Set Poster View menu item, and you're done!

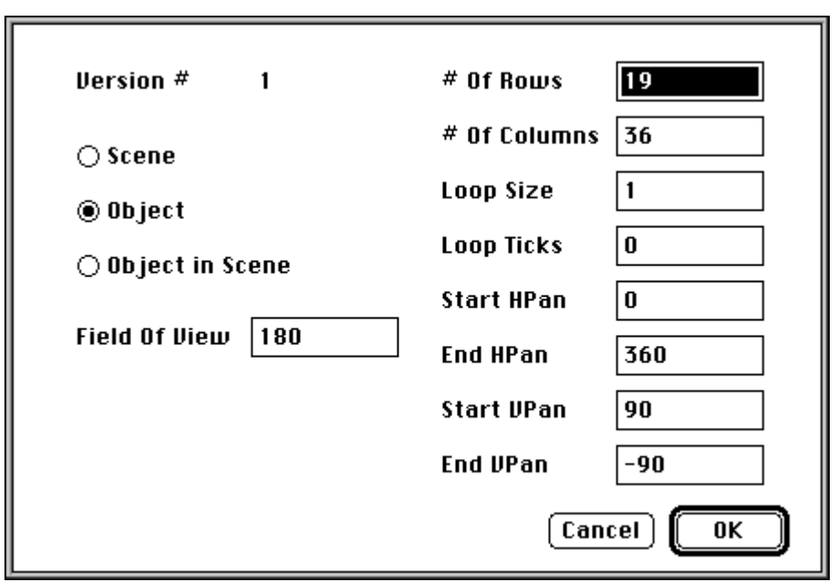


Figure 4. The Add Navigable Data dialog

MAKING A QUICKTIME VR PANORAMIC MOVIE

There are two approaches to creating panoramic movies. One way is to simulate a real camera, rotate the camera to generate a series of images, and then "stitch" the images together into a single 360° panoramic PICT file with the QuickTime VR stitching tool. This panoramic picture file can be converted first to a linear movie and then to an interactive movie with the QuickTime VR ATS. This is the technique we'll look at first. You can also render a single panoramic image directly; this avoids the need for the stitching tool and enables us to convert the image into an interactive panoramic movie with only one line of script. The setup is similar for both approaches.

Throughout the QuickTime VR documentation, examples and references assume a vertically oriented, 360° full panorama that's 768 pixels across by 2496 pixels high, captured with a 15mm lens using portrait orientation. This is exactly the panorama we'll create.

Before you begin making the movie, you need to determine the number of shots required to make your panorama. For a real 360° panorama, the number of shots is a function of the length of your lens and the amount of overlap between the shots. For a rendered panorama, however, the number of shots is a function only of the horizontal field of view. Because the camera position and lighting conditions are controlled in the code, overlap between the shots isn't necessary. You can specify any amount of overlap, but theoretically a one-pixel overlap is all that's required.

In our case, we'll simulate the camera that's recommended in the QuickTime VR documentation — that is, a camera with a 15mm lens. To be consistent with the examples in the documentation, we'll shoot 12 pictures, each 30° apart. This will give us an overlap of about 50%.

The function MyConvert3DMFToPano (Listing 12) drives the entire panoramic moviemaking process. Much like MyConvert3DMFToObject, this function creates a document, reads in the model, and renders the appropriate images.

DOING THE MODEL AND CAMERA WORK

The initial placement of the camera is performed by the function MyInitPanoCamera (Listing 13). It first gets the bounding sphere of the model's drawable group, and from the bounding sphere gets the center 3D point of the group (as the origin of the bounding sphere). Of course, you can place your camera anywhere you like in the scene. For simplicity, we placed our camera in the center. From that position, the camera is rotated to create the images.

For panoramic rendering, we rotate the camera around its *y* axis with the function MyRotateCameraY, shown in Listing 14. To do the rotation, we do the equivalent of translating to local coordinates, rotate, and then translate back to world coordinates. We do the transformation by making a rotation matrix about the local *y* axis (the up vector) at our location, getting our local *z* axis (by subtracting the point of interest from the current location), and then rotating the *z* vector around the *y* axis. We then apply this transformation to get a new point of interest. The rectangles encircling the camera in Figure 2 represent the images rendered after each camera rotation.

To create a series of images, we step through the camera rotations with the MyGeneratePanoFrames function (Listing 15). Here we rotate the camera 30° at a time, render an image, and write the image to a PICT file. This gives us 12 PICT files named 01 through 12 that can be used in a very straightforward manner by the QuickTime VR ATS. To create a single panoramic image, you use the function MyGeneratePanoMovieDirect, as shown later.

Listing 12. Converting 3DMF files to panoramic images

```
void MyConvert3DMFToPano(FSSpec *myFSS)
{
   DocumentPtr    theDocument;

   // Create the document record.
   theDocument = MyNewDocument();

   // Read the model into the document record.
   MyOpenFile(myFSS);

   // Set up the initial camera position.
   MyInitPanoCamera(theDocument);

   // Draw initial view to the screen.
   MyDrawOffScreen(theDocument);
   MyDrawOnScreen(theDocument);
   ...
   // Create a series of images to stitch together into a panorama.
   MyGeneratePanoFrames(theDocument);

   // To create a single panoramic image, call MyGeneratePanoMovieDirect
   // instead of MyGeneratePanoFrames.
   ...
}
```

Listing 13. Setting the initial camera position for a panoramic movie

```
TQ3Point3D MyInitPanoCamera(DocumentPtr theDocument)
{
   TQ3BoundingSphere    viewBSphere;
   float                viewRadius;

   // Get the bounding sphere of the drawable group (the entire model)
   // in the view.
   MyGetBoundingSphere(theDocument->theView, theDocument->documentGroup,
         &viewBSphere);

   // Get the bounding sphere's center and radius.
   theDocument->documentGroupCenter = viewBSphere.origin;
   viewRadius = viewBSphere.radius;

   // Position the camera in the center of the bounding sphere.
   MyPlaceCamera(theDocument, theDocument->documentGroupCenter.x,
         theDocument->documentGroupCenter.y,
         theDocument->documentGroupCenter.z,
         theDocument->documentGroupCenter.x,
         theDocument->documentGroupCenter.y,
         theDocument->documentGroupCenter.z + 1.0);

   return (theDocument->documentGroupCenter);
}
```

Listing 14. Rotating the camera for panoramic rendering

```
void MyRotateCameraY(DocumentPtr theDocument, float dY)
{
   TQ3CameraObject        camera;
   TQ3CameraPlacement     cameraPos;
   TQ3Matrix4x4           myRotation;
   TQ3Vector3D            initialVector, rotatedVector;

   Q3View_GetCamera(theDocument->theView, &camera);
   Q3Camera_GetPlacement(camera, &cameraPos);

   // Get the z vector.
   Q3Point3D_Subtract(&cameraPos.pointOfInterest,
         &cameraPos.cameraLocation, &initialVector);

   // Rotate around the y axis.
   Q3Matrix4x4_SetRotateAboutAxis(&myRotation,
         &cameraPos.cameraLocation, &cameraPos.upVector, dY);

   // Rotate the z vector around the y axis.
   Q3Vector3D_Transform(&initialVector, &myRotation, &rotatedVector);

   // Get the point of interest from the rotated vector. The upVector
   // doesn't change.
   Q3Point3D_Vector3D_Add(&cameraPos.cameraLocation, &rotatedVector,
         &cameraPos.pointOfInterest);

   Q3Camera_SetPlacement(camera, &cameraPos);
   Q3View_SetCamera(theDocument->theView, camera);
   Q3Object_Dispose(camera);
}
```

Listing 15. Generating a series of images for a panoramic movie

```
void MyGeneratePanoFrames(DocumentPtr theDocument)
{
   PicHandle      thePict;
   float          zAngle;
   long           counter = 0;
   Str255         fName;
   GWorldPtr      gw;
   GDHandle       gd;
   FSSpec         outSpec;

   GetGWorld(&gw, &gd);
   SetGWorld(theDocument->theWindow, nil);

   outSpec = theDocument->theFileSpec;
   for (zAngle = 360; zAngle >0; zAngle -= 30) {
      short    i;
```

(continued on next page)

Listing 15. Generating a series of images for a panoramic movie *(continued)*

```
      MyRotateCameraY(theDocument, -30.0*kQ3Pi/180.0);
      SetGWorld(theDocument->theWindow, nil);
      MyDrawOffScreen(theDocument);
      MyDrawOnScreen(theDocument);

      SetGWorld(theDocument->drawContextOffScreen, nil);
      thePict =
            OpenPicture(&theDocument->drawContextOffScreen->portRect);

      LockPixels(theDocument->drawContextOffScreen->portPixMap);
      CopyBits((BitMap*)&theDocument->drawContextOffScreen->portPixMap,
            (BitMap*)&theDocument->drawContextOffScreen->portPixMap,
            &theDocument->drawContextOffScreen->portRect,
            &theDocument->drawContextOffScreen->portRect,
            srcCopy, NULL);
      UnlockPixels(theDocument->drawContextOffScreen->portPixMap);
      ClosePicture();
      ...   // Set up the outSpec for the next image.
      MySavePICT(thePict, &outSpec);
      KillPicture(thePict);
   }
   ...
}
```

GENERATING THE INTERACTIVE PANORAMIC MOVIE

You use the MPW tools and scripts provided in the QuickTime VR ATS to generate your interactive panoramic movie. (See the QuickTime VR ATS documentation for more information.)

Stitching the images. The stitching tool, called by the Stitch768 script, joins the series of computer-rendered images of your panorama into a single 360° panoramic PICT file. It's also used to stitch digitized photographic images together. The following example shows a portion of a stitch worksheet, with appropriate values set for each of the input variables. The stitching tool will use images numbered from 01 to 12 located on the drive named HappyMac.

```
set panInFolder "HappyMac:RenderedFrames:"
set panOutFolder "HappyMac:RenderedFrames:"
set panRotate 0
set panX 100
set panDX 20
set panY 0
set panDY 10
export panInFolder panOutFolder panRotate panX panDX panY panDY
Stitch768 01-12 MyPano.srcPict
```

This script stitches your images into a vertically oriented, 360° full panorama that's 768 pixels across by 2496 pixels high, named MyPano.srcPict.

Dicing the image into a linear movie. The SrcPictToMovie script calls the dicing tool, which compresses your PICT file and divides it into equal-sized sections called *tiles*. For example, a standard-size panorama is "diced" into 24 tiles, 1 across by 24

down. Since we haven't included hot spots in our movie, our worksheet will include only one line, which calls the SrcPictToMovie script:

```
SrcPictToMovie "HappyMac:RenderedFrames:MyPano.srcPict" ∂
   "HappyMac:RenderedFrames:MyPano.srcMoov"
```

This script dices your 360° panoramic PICT into a standard QuickTime linear movie using 1-by-24 tiling and the Cinepak compressor.

Converting the linear movie to an interactive movie. The MakeSingleNodeMovie script takes the linear movie we just created and generates an interactive panoramic movie. Since we're creating a very standard type of interactive movie, this script does everything we need.

This example creates a single-node interactive panoramic movie file named My3DMovie:

```
MakeSingleNodeMovie "MyPano.srcMooV" "My3DMovie"
```

RENDERING YOUR PANORAMA DIRECTLY

You can avoid using the stitching tool by rendering your panorama directly. However, since QuickDraw 3D supports rendering directly to a plane but not to a cylinder, we have to approximate cylindrical rendering with a "slit" approach, using the cameras available to us.

The slit approach is the equivalent of using a real panoramic slit camera, which spins around, taking very thin pictures and laying them next to each other on the film. When simulating cylindrical rendering, we do the camera work described earlier, but instead of rotating 30° at a time and grabbing each frame, we rotate a very small amount each time and just grab a slit out of the middle of each frame, thus approximating a cylinder. The narrower the slit width, the closer we get to a true cylinder. If you're curious about the mathematics of slit sizes, see "Calculating the Optimal Slit Width."

In our case, the largest slit size that gives us a very small amount of error is 32, so we use this number to generate our panorama (Listing 16). To try different sizes, simply put in a different number for the slit size constant, which we've called **factor**.

The PICT file you get from this operation is oriented horizontally. However, the QuickTime VR tools expect the stitching tool output to be vertical, so you first need to rotate your PICT clockwise 90° using a PICT editor such as Photoshop or PhotoFlash. You then use the SrcPictToMovie and MakeSingleNodeMovie scripts as described above to turn the PICT into an interactive panoramic movie.

THE NEXT STEPS

So far we've made QuickTime linear object movies and panoramic PICT files that can be converted to interactive movies with the QuickTime VR ATS. There are a number of directions you can go from here. If you have your own panoramic renderer, you may want to substitute it for our slit-based rendering. Or you may want to build a full interface to allow the user to place the camera and set up all the parameters involved in QuickTime VR moviemaking. We hope to write a future article about the QuickTime VR movie file formats and how to write out QuickTime VR movie files.

QuickTime VR movies already have several diverse uses. Developers with extensive collections of 3D data sets can generate QuickTime VR movies from their data sets

Listing 16. Rendering the panorama directly

```
#define factor 32.0
void MyGeneratePanoMovieDirect(DocumentPtr theDocument)
{
   PicHandle      thePict;
   float          zAngle;
   short          i;
   GWorldPtr      gw, largeGW;
   GDHandle       gd;
   FSSpec         outSpec;
   Rect           sourceRect, destRect, largeRect = {0, 0, 768, 2496};

   GetGWorld(&gw, &gd);
   SetGWorld(theDocument->theWindow, nil);
   outSpec = theDocument->theFileSpec;

   NewGWorld(&largeGW, 32, &largeRect, nil, nil, useTempMem);
   LockPixels(largeGW->portPixMap);
   SetGWorld(largeGW, nil);
   EraseRect(&largeRect);
   sourceRect = destRect = largeRect;
   sourceRect.left = 256 - factor/2.0;
   sourceRect.right = sourceRect.left + (short)factor;
   destRect.left = 0;
   destRect.right = (short)factor;

   for (zAngle = 360.0; zAngle > 0.0; zAngle -= 360.0/(2496.0/factor)) {
      MyRotateCameraY(theDocument, -2*kQ3Pi/(2496.0/factor));

      SetGWorld(theDocument->theWindow, nil);
      MyDrawOffScreen(theDocument);
      MyDrawOnScreen(theDocument);

      SetGWorld(largeGW, nil);
      LockPixels(theDocument->drawContextOffScreen->portPixMap);
      CopyBits((BitMap*)&theDocument->drawContextOffScreen->portPixMap,
            (BitMap*)&largeGW->portPixMap, &sourceRect, &destRect,
            srcCopy, NULL);
      UnlockPixels(theDocument->drawContextOffScreen->portPixMap);
      destRect.left = destRect.left + (short)factor;
      destRect.right = destRect.right + (short)factor;
   }

   SetGWorld(largeGW, nil);
   thePict = OpenPicture(&largeGW->portRect);
   CopyBits((BitMap*)&largeGW->portPixMap, (BitMap*)&largeGW->portPixMap,
         &largeGW->portRect, &largeGW->portRect, srcCopy, NULL);
   ClosePicture();
   UnlockPixels(largeGW->portPixMap);
   DisposeGWorld(largeGW);
   MySavePICT(thePict, &outSpec);
   ...
}
```

CALCULATING THE OPTIMAL SLIT WIDTH

In the slit approach to simulating cylindrical rendering for QuickTime VR panoramic movies, narrower slits approximate cylinders better than wider ones. In our calculations, the size of the error shows us the effect of increasingly wide slits.

Our error is defined as the vertical distance between the top of our projection plane at the maximum vertical field of view and the top of the cylinder we're trying to approximate. We consider an error of less than 0.5 pixels to be acceptable. Since fractional pixels can't be drawn, errors greater than 0.5 will round up to be a full pixel error. Because this error is so small, we can use the same field of view for generating both the slits and the entire frame.

To determine the vertical error, we must first determine the maximum horizontal distance between the plane and the cylinder. This distance, labeled *y*, can be seen in the top view of our camera and cylinder, as shown in Figure 5. The two triangles formed are identical (except for their orientation). The width of our slit is 2*x*.

Given that 2*x* is the width of the slit, *y* is the distance between the plane and the cylinder, *r* is the radius of the cylinder, and α is the angle:

$c = 2\pi r$ $\qquad r = \frac{c}{2\pi}$

$\sin \alpha = \frac{x}{r}$ $\qquad \alpha = \sin^{-1}(\frac{x}{r})$

$\cos \alpha = \frac{r-y}{r} = 1 - \frac{y}{r}$ $\qquad \alpha = \cos^{-1}(1 - \frac{y}{r})$

Therefore, where *r* is the radius of the cylinder and *c* is the circumference:

$$\alpha = \sin^{-1}(\frac{x}{r}) = \cos^{-1}(1 - \frac{y}{r})$$

$$y = r\,(1 - \cos(\sin^{-1}(\frac{x}{r})))$$

$$y = \frac{c\,(1 - \cos(\sin^{-1}(\frac{2\pi x}{c})))}{2\pi}$$

Since we know the final panorama we end up with is 2496 pixels wide, we can use this as our circumference, and

$$y = 397.25\,(1 - \cos(\sin^{-1}(\frac{x}{397.25})))$$

However, this only gives us the slit width for a given distance *y*, so we must concern ourselves next with the important error, the vertical error, labeled *Ev*. A side view of the panorama showing this error appears in Figure 6.

Ev is the distance in pixels between the pixel we see on the plane and the pixel we see on the cylinder for a given field of view. Since we already have an equation for *y* in terms of our slit width (2*x*), and we know that the vertical field of view (*FOVv*) of the lens we're using is 97°, we can easily determine this error using the tangent equation

$$Ev = y \tan(\frac{FOVv}{2})$$

$$Ev = 397.25\,(1 - \cos(\sin^{-1}(\frac{x}{397.25}))) \tan(\frac{FOVv}{2})$$

Since we know that our *FOVv* is 97°, we have

$$Ev = y \tan(48.5)$$

which leaves us with a final equation of

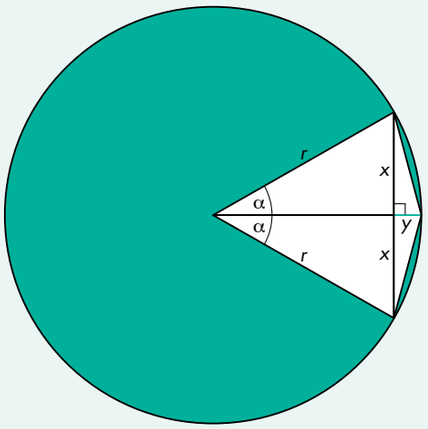
$$Ev = 397.25\,(1 - \cos(\sin^{-1}(\frac{x}{397.25}))) \tan(48.5)$$

The slit widths for various vertical errors are as follows:

Vertical error (*Ev*)	**Slit width (2*x*)**
0.14228682	20
0.20490732	24
0.27892462	28
0.36434438	32
0.4611731	36
0.56941818	40

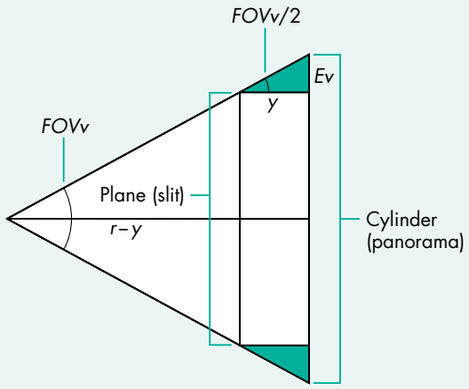
For a slit width of about 38, we have an error of less than 0.5. Theoretically, this should yield accurate pictures. Therefore, for panoramas that are 2496 pixels wide, like ours, the optimal slit width is 32 (the largest factor of 2496 that's still less than 38).

to show to potential customers; the movies display the modeled objects more effectively than a 2D representation and don't compromise the data in the process. Archaeologists can use QuickTime VR movies to record site information during digs, realtors can use them to give clients virtual tours through the property they're offering, and cities can use them to provide tourist information on kiosks. Museums



α = angle
r = radius
2*x* = width of the slit
y = distance between the plane and the cylinder

Figure 5. Determining *y*, the distance between the plane and the cylinder



Ev = vertical error
FOVv = vertical field of view
r = radius
y = distance between the plane and the cylinder

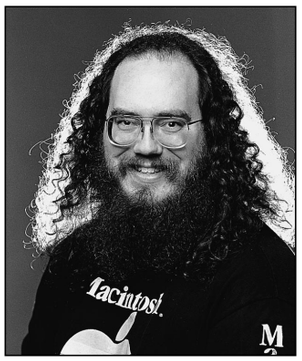
Figure 6. Determining *Ev*, the vertical error

can archive or display their collections with QuickTime VR movies. For example, Apple and the Asian Art Museum of San Francisco have put together a virtual walkthrough of one of the museum's special exhibits; you can check it out on the World Wide Web at http://sfasian.apple.com. Also, for the latest information on QuickTime VR, see http://qtvr.quicktime.apple.com. Use your imagination — the possibilities are endless!

RECOMMENDED 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.
- *3D Graphics Programming With QuickDraw 3D* (Addison-Wesley, 1995).
- *Inside Macintosh: QuickTime* and *Inside Macintosh: QuickTime Components* (Addison-Wesley, 1993).
- *One Hundred Years of Solitude* by Gabriel García Márquez (Harper & Row, 1970). This won't directly help you with QuickTime VR or QuickDraw 3D, but it's Philip's favorite novel.

Thanks to our technical reviewers Eric Chen, Michael Chen, Ken Doyle, Ian Small, and Nick Thompson. Special thanks to Chris Flick and Pablo Fernicola.•



PRINT HINTS

QuickDraw GX Breaks the Space Hack

DAVE POLASCHEK

Before QuickDraw GX, when an application that generated its own PostScript™ code wanted to make sure the printer could print a particular font, it could send one space character in the needed font. The LaserWriter driver would check the printer to see if the font was available, and if not, the driver would send the font to the printer so that it would be available to print the space character — and any other characters in that font that the application-generated PostScript code might require. The reason for using a space was simple: you didn't want to mark the page just to get a font to the printer, and a space wouldn't mark it. This technique, first described in "The Perils of PostScript" back in *develop* Issue 1, became known as the "space hack."

Unfortunately, the space hack doesn't work with QuickDraw GX. This column describes a new way for applications that generate their own PostScript code to send fonts to the printer. The code to do this is provided on this issue's CD.

QUICKDRAW GX CHANGES THE PICTURE

QuickDraw GX has a really cool imaging model, supports all kinds of whizzy features, and to top it off, introduces the long-awaited new printing architecture. But it has one snag: after all the years you've spent getting your PostScript printing tuned just the way you like it, QuickDraw GX breaks the space hack.

The space hack depends on a font's entire character set being sent to the printer in response to the need for a single character (the space character). But QuickDraw GX sends only the needed characters in a font to a printer, because it's trying to conserve memory on the printer and also because sending less data means faster transmission of that data. This isn't such a big issue with Roman fonts, where there are only 256 characters at most, but in the case of two-byte fonts such as Chinese, Japanese, and Korean fonts, where there can be tens of thousands of characters and the font can be tens of megabytes in size, sending only the required characters makes a big difference in speed.

Incidentally, with QuickDraw GX you don't need a specialized printer to print two-byte fonts. It divides fonts with more than 256 characters into several smaller fonts with new encodings containing just the characters you need, so you can print characters from the font on any PostScript printer.

THE NEW WAY TO DOWNLOAD FONTS

So QuickDraw GX has lots of advantages over QuickDraw, but the space hack is broken. What's the poor programmer to do?

You can use a new font downloading method based on calling GXFlattenFont, a handy function introduced with QuickDraw GX, to convert the font to a form that's easily sent to the printer. GXFlattenFont is intended to convert any font present on your Macintosh into the output font format of your choice. (Conversion is limited by the capabilities of the scalers present, as explained in "QuickDraw GX Font Scalers.")

QUICKDRAW GX FONT SCALERS

The QuickDraw GX Open Font Architecture accepts drop-in font scalers. A font scaler is a bit of code that takes a font of a given type and converts it to bitmaps for display. It also converts fonts to outline format and can optionally convert a font to another font format. QuickDraw GX includes three default scalers:

- the bitmap scaler, which is essentially the same as in QuickDraw
- the TrueType GX scaler, which supports the TrueType GX format
- the Type 1 scaler, which is part of Adobe™ Type Manager

All of these default scalers are capable of generating bitmaps for screen display and PostScript fonts for printing. Only the TrueType GX scaler can generate downloadable TrueType fonts.

DAVE POLASCHEK recently relocated to California to join Apple's Developer Technical Support group. He's been told that supporting printing leads to hair loss and insanity. Dave previously lived in beautiful sunny Minnesota, and wonders if he'll get used to the harsh San Francisco Bay Area winters before he's bald and crazy, or if it's already too late.•

GXFlattenFont can produce Type 1 data that's ready to be sent to your PostScript printer with no problem.

Now let's turn to the code that replaces the old space hack. The rough idea is to call GXFlattenFont on a QuickDraw font reference and a set of characters (an encoding) that you need to print, and return the result in a form that's easy to send to the printer. For simplicity, if no encoding is present, we use the standard Macintosh encoding. Listing 1 shows a font-downloading routine, FontToPict, that uses this technique if QuickDraw GX is installed. (This is a somewhat simplified version; see the CD for the full code of FontToPict and its related utility functions.)

FontToPict starts by checking to see if QuickDraw GX is installed. If not, it uses the old hack of printing a space; otherwise, it calls MakePSHandle (Listing 2), which calls the utility function ConvertQDFontToGXFont to convert the QuickDraw font reference into a QuickDraw GX font reference. MakePSHandle then checks to see if an encoding has been passed in; if not, it builds the standard Macintosh encoding. Next it calls FontToHandle, which is just a wrapper for GXFlattenFont. GXFlattenFont converts the specified font to the Type 1 format. Error-handling and cleanup code is last. Simplicity itself! The result, whether QuickDraw GX is present or not, is a PICT that you can send to the printer by calling DrawPicture once the printer port has been opened.

When calling MakePSHandle, you should specify an encoding array that contains the characters you intend to actually print. This prevents QuickDraw GX from sending the entire font to the printer and becomes very important when you make your application WorldScript aware. There's an #ifdef in the code on the CD that generates only the encoding array you need in order to use a portion of the font. As mentioned earlier, with Chinese, Japanese, and Korean fonts, sending only the characters you need can make the difference between sending a few kilobytes or many megabytes of data to the printer. If you don't use the entire font, remember to encode the characters that you want to draw, using the same encoding that you passed in to the MakePSHandle function.

You may want to have HandleSpoolProc (which is called by GXFlattenFont and included on the CD) spool directly to the printer via picture comments. This way you won't need memory available to hold the font data at the intermediate steps.

Listing 1. FontToPict

```
PicHandle FontToPict(short qdFont, short qdStyle)
{
   Rect        theRect = {0, 0, 1, 1};
   PicHandle   thePict = OpenPicture(&theRect);
   const short kPostScriptHandle = 192;

   if (GXInstalled()) {  // If QuickDraw GX is installed, use the new method.
      Handle         piccommentHdl;
      unsigned short *myEncoding = nil;

      MakePSHandle(qdFont, qdStyle, myEncoding, &piccommentHdl);
      PicComment(kPostScriptHandle, GetHandleSize(piccommentHdl), piccommentHdl);
   } else {          // If QuickDraw GX isn't installed, use the old method.
      Point penPoint;

      // We would normally set the clip here, but since we're just drawing a space there's no need.
      GetPen(&penPoint);    // Save the pen location.
      TextFont(qdFont);
      TextFace(qdStyle);
      DrawChar(' ');
      MoveTo(penPoint.h, penPoint.v);    // Restore the pen location.
   }
   ClosePicture();
   return (thePict);
)
```

Listing 2. MakePSHandle

```
OSErr MakePSHandle(short qdFont, char qdStyle, unsigned short *encodingArray, Handle *outputHandle)
{
   OSErr           status = noErr;
   gxFont          theFont;
   unsigned short  *myEncoding;
   Boolean         madeEncoding = false;

   theFont = ConvertQDFontToGXFont(qdFont, qdStyle);  // Convert to a QuickDraw GX font reference.

   // If no encoding, create the standard Macintosh encoding.
   if (!encodingArray) {
      long   returnLength;

      myEncoding = (unsigned short *)NewPtrClear(256 * sizeof(short));
      returnLength = MakeMac8BitEncoding(theFont, myEncoding);
      if (returnLength != 256) {
         DebugStr("\pHmm. We didn't get a full encoding.");
         return (returnLength);    // Pass the error along.
      }
      madeEncoding = true;
   } else {
      myEncoding = encodingArray;
   }

   *outputHandle = FontToHandle(theFont, myEncoding);
   if (madeEncoding) DisposePtr((Ptr)myEncoding);

   status = MemError();
   if (status == noErr) {
      status = GXGetGraphicsError(nil);
      if (status != noErr) {
         DisposeHandle(*outputHandle);
         *outputHandle = nil;
      }
   }
   return (status);
}
```

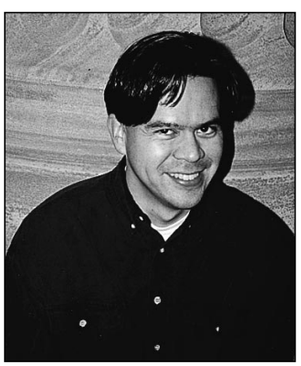
DOWNLOADING HAPPINESS

The new font downloading method takes a little more work but produces better results in your printer font handling. You can easily send needed fonts to the printer, either the whole font or only the characters you'll be using. As a side benefit, you get support for two-byte font systems without having to write custom code for handling the large fonts or, worse yet, having to depend on the fonts being installed on the printer in a specific manner. Even if you're not ready to add QuickDraw GX imaging to your application today, adding QuickDraw GX compatibility improves the printing experience for your customers.

Thanks to Dan Lipton for providing the idea and core code illustrating the new font downloading method, and to Pete "Luke" Alexander, Dave Hersey, and Dan Lipton for reviewing this column.•

Flicker-Free Drawing With QuickDraw GX

Does your QuickDraw GX application have a look reminiscent of the old silent movies? If so, it suffers from flicker. But don't despair — help is as near as this issue's CD, where you'll find a ready-to-use library for doing memory-efficient, flicker-free drawing inside a window. This article explores the problem of flicker and its solutions and walks you through the code.



HUGO M. AYALA

My first encounter with the idea of flicker-free drawing happened when I was a 12-year-old kid reading my father's copy of *Nibble*, a journal about programming the Apple II. A review of new products mentioned a program that had impeccable animation and guessed that the programmer was most likely using "page switching" to produce flicker-free drawing. Page switching (or page flipping) took advantage of the fact that the Apple II could use more than one location in memory (more than one *page*) to hold the screen image. Given enough memory, a programmer could set things up so that there was a second "offscreen" page to draw into while the first was being shown on the screen. Switching back and forth between these two pages made flicker-free drawing possible.

Today's hardware bears little resemblance to the Apple II, but the technique for doing flicker-free drawing is essentially the same. It involves *double buffering* (also known as *screen buffering*) — causing all objects to be drawn first into an offscreen buffer and then copying that entire buffer to the front buffer (the window). Both this and the Apple II method eliminate the need to erase the old position of a moving image directly on the screen before drawing its new position, which is the primary cause of flicker.

The library that accompanies this article manages an offscreen buffer for a QuickDraw GX view port. Using it will enable you to give your QuickDraw GX application a more professional feel by removing flicker. You could use the offscreen library provided with QuickDraw GX to do screen buffering, but because it's a much more general-purpose tool, you would have to handle most of the minutiae of examining screen devices, filling out the bitmap data structures, and allocating and

HUGO M. AYALA (hugo@mit.edu, http://web.mit.edu/hugo/www) spent five years working on QuickDraw GX as a development engineer at Apple before returning to MIT to pursue a Ph.D. in mechanical engineering. His current research interest is how to design the undercarriage of large earth-moving equipment so that it doesn't get thrashed so fast by rocks and dirt. To pay for the Ph.D., he moonlights doing computer graphics work, which has been his hobby since he was a lad. After finishing his Ph.D., Hugo plans to branch off into drawing comic strips, like the one that he's been drawing for his school newspaper. If you ever try to give Hugo directions, you need to know that he's directionally challenged — he really can't tell his left from his right.•

releasing the memory yourself. The library provided on this issue's CD does all of that for you.

I'll walk you through the library code, illustrated by the sample application called Flicker Free on the CD, but first I'll give some background on the problem of flicker and its solutions. This article assumes that you already know a thing or two about QuickDraw GX; if you don't, see the article "Getting Started With QuickDraw GX" in *develop* Issue 15. The essential references are *Inside Macintosh: QuickDraw GX Objects* and *Inside Macintosh: QuickDraw GX Graphics*.

FLICKER — ITS CAUSES AND SOLUTIONS

For a dramatic illustration of flicker, run the sample application Flicker Free (you'll need a color Macintosh with QuickDraw GX installed). You'll see a window filled with fifty circles bouncing around in different directions (see Figure 1).

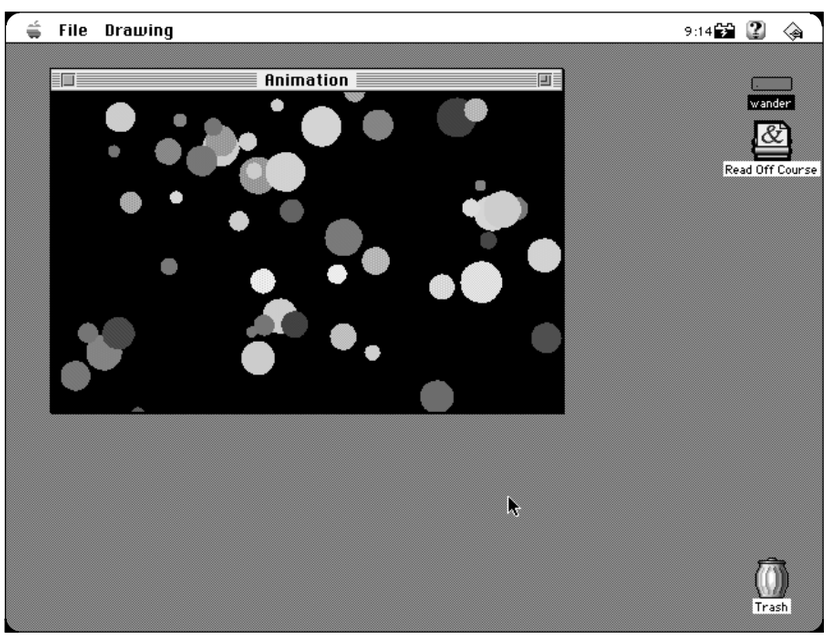


Figure 1. The startup screen from the sample application Flicker Free

The Drawing menu in the Flicker Free application offers a choice of buffering methods: full screen buffering, no screen buffering, and half and half. The program starts up in half-and-half mode: the left side of the window (the side with the Apple menu, for those like me who can't tell left from right) is being buffered, while the other side isn't. Switch among the buffering choices to get a sense of the difference that flicker or its absence makes in how you experience the animation.

What causes flicker? In our case, the shapes on the right are being erased and then redrawn over and over again as they move across the screen. And although the rendering of the shapes is very fast (your mileage may vary according to CPU speed), the act of constantly drawing and erasing them makes the whole thing look like an old silent movie. In places where circles overlap, pixels are made to take on different colors as each shape is drawn. In the resulting blur of colors, it's hard to see which shape is in front.

The key to avoiding flicker is to avoid erasing pixels on the screen needlessly between two stages of a drawing and to change only the color of those pixels that need to change. The left side of our sample application window is being double buffered, meaning that each circle is drawn into an offscreen buffer and then the whole scene is transferred onto the screen. Because at each step in the animation only the pixels that need to change color do, the movement of the circles is rendered flicker free. With double buffering there's no problem telling which circles are in front. Shapes move neatly past each other.

Figure 2 shows two frame-by-frame drawing sequences illustrating the difference between an update full of flicker and a flicker-free update.

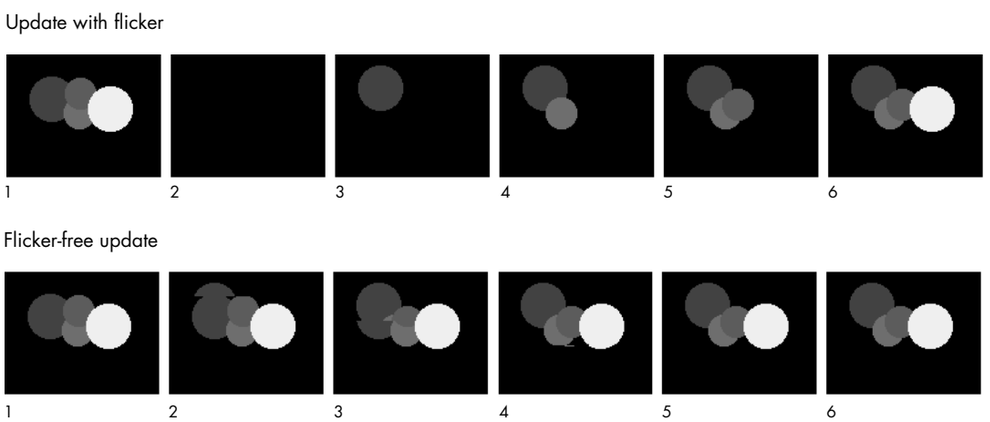


Figure 2. An update full of flicker vs. a flicker-free update

The upper set of frames in Figure 2 shows what happens without double buffering. The screen is erased (in frame 2 and then again, out of view, in frame 7) and then each circle is added to the screen in its new position. The whole assembly of circles appears on the screen only briefly before they're erased and the process is started again. The lower set of frames in the figure shows the update process during double buffering. The offscreen image is transferred to the screen in a sweep replacing the previous image. You can see the sweep line as a very subtle horizontal break in the frame.

The sample application gives a dramatic demonstration of how flicker affects animation. But even if your QuickDraw GX application isn't an animation package, it probably suffers from some form of flicker when update events are serviced. The most common and most annoying flicker occurs when applications engage in some form of user interaction — for example, dragging marquees, manipulating shapes, and editing text.

BUFFERING TRADEOFFS

When you're considering using screen buffering, it's important to understand the tradeoff with drawing speed. In the sample application, the speed at which the circles travel is a function of the number of circles in the window, the size of the window, and your choice of screen buffering. Given the same window size and number of shapes, *drawing with screen buffering is always slower than with no screen buffering.* Screen buffering involves the same amount of work as screen drawing plus the additional step of transferring the offscreen image onto the screen.

When the window contains one circle, the unbuffered performance is at least three times faster than that of the buffered case (again, your mileage may vary depending on your CPU speed). As more shapes are added, the performance in both cases goes down, but so does the performance gap between the two: the unbuffered performance doesn't have as much of an advantage over the buffered performance. This is because the speed at which the offscreen buffer is transferred to the screen is independent of the complexity of the shape it contains; it's purely a function of its size. As the complexity of the shape being buffered increases, the relative cost of shape buffering decreases.

Now, this doesn't mean that you should buffer only complex shapes that take a long time to draw. What it means is that when you add screen buffering to your application, you have to be mindful of what constitutes a reasonable tradeoff between buffering and drawing performance. You should try things out and see if screen buffering is the technique best suited to your needs. Alternatives to screen buffering that enable flicker-free drawing include the use of transfer modes and geometric operations. I hope to discuss these in a future *develop* article.

Meanwhile, we'll take a look at the screen buffering library that accompanies this article, which is ready for you to incorporate into your QuickDraw GX application. I wrote the library with performance issues in mind. Thus, it takes advantage of the fact that in the QuickDraw GX graphics object model, information that's needed to render a shape can be computed once, stored in a drawing cache, and reused every time that shape is drawn. The library is very careful to check before making calls that invalidate drawing caches, so the overhead of offscreen drawing is kept to a minimum.

A LOOK AT THE SCREEN BUFFERING LIBRARY

Everything you need in order to use the screen buffering library is defined in the interface file. The library consists of four major routines: the routine that creates the view port buffer object, the one that disposes of it, the one that updates it, and the one that uses it to actually buffer screen drawing. The four corresponding calls should parallel the drawing loop inside your application.

The include file defines only one data type:

```
typedef struct viewPortBufferRecord **viewPortBuffer;
```

The internals of the data type are private to the "screen buffering.c" file and are as follows:

```
struct viewPortBufferRecord  {
   gxViewGroup    group;      /* The offscreen's view group. */
   gxViewDevice   device;     /* The offscreen's view device. */
   gxViewPort     view;       /* The offscreen's view port. */
   gxShape        buffer;     /* The bitmap of the offscreen's view device. */
   gxBitmap       bits;       /* Source structure for the buffer shape. */
   Handle         storage;    /* A temp handle to the bits of the bitmap. */

   gxTransform    offxform;   /* This draws into the offscreen. */
   gxTransform    on_xform;   /* This draws onscreen. */
   gxShape        eraser;     /* Erases offscreen to background color. */
   gxShape        marker;     /* Used to draw into the offscreen. */
   gxShape        updatearea; /* Represents the area to update. */
```

```
    short          usehalftone;   /* True if screen has a halftone. */
    WindowPtr      window;        /* The window of the view port. */
    gxViewPort     parent;        /* The parent's view port. */
    gxViewPort     screenview;    /* The view port to buffer. */
    gxShape        page;          /* The shape that we're asked to draw. */

    gxRectangle    bounds;        /* The offscreen's bounds. */
    gxMapping      invmap;        /* The inv offscreen view port map. */
    gxPoint        viewdelta;     /* The last delta for the offscreen. */
};

typedef struct viewPortBufferRecord viewPortBufferRecord;
```

You don't need to understand all of the fields in the viewPortBufferRecord data structure to use the library. However, if you start having problems getting things to work inside your application and find that you need to modify the screen buffering library, see "The viewPortBufferRecord Data Structure" for some additional helpful information.

In general terms, the code works by allocating a number of QuickDraw GX objects and reusing them as required. Memory for the offscreen buffer is allocated from the MultiFinder temporary memory heap (Temp Mem). Allocation of the storage block is postponed until the last possible moment, and the block is kept locked and nonpurgeable only during the drawing operation. That is, after the resulting image has been transferred to the screen, the block is unlocked and marked purgeable but *not disposed of*. This permits the same block to be reused in case the memory for the buffer isn't purged.

While most users will keep their windows entirely within the bounds of one screen, it's important to handle the case where a window spans more than one device. Each time the DrawShapeBuffered routine is called (as described later), the code walks the device list checking to see if the area that needs to be buffered intersects a given screen. If it does, the code creates a buffer with the right settings and draws into that device. The process is repeated for each screen.

CREATING AN OFFSCREEN BUFFER

You'll need one view port buffer for each window in your program. To create a view port buffer, use the NewViewPortWBuffer routine.

```
viewPortBuffer NewViewPortWBuffer(WindowPtr window, gxViewPort view,
                                  const gxColor *backgroundColor);
```

Look at the Initialize routine in the file "flicker free.c" for an example of how to use NewViewPortWBuffer. Here's a description of the parameters:

window	The window that the buffering code should draw into.
view	The view port created by your application to draw into the given window. Note that this is different from the object obtained by calling GXNewWindowViewPort, in that this view port should have the window view port set to be its parent.
backgroundColor	A pointer to a gxColor data structure indicating which color should be drawn to erase the offscreen buffer. Passing nil is equivalent to specifying white as the background color.

THE VIEWPORTBUFFERRECORD DATA STRUCTURE

The following is an accounting of all of the fields of the viewPortBufferRecord data structure.

- **group**, **device**, **view** — These are the three elements of an offscreen drawing environment in QuickDraw GX. We need one of each to draw offscreen.
- **buffer**, **bits**, **storage** — These objects represent the bits of the offscreen device in decreasing order of abstraction. The field **buffer** is a bitmap shape that represents the "screen" of the view device. The field **bits** parallels the contents of **buffer** and is used to keep information about the offscreen bitmap around between invocations of DrawShapeBuffered, the routine that draws the buffered shape. Finally, the **storage** field is the handle in Temp Mem (the MultiFinder temporary memory heap) that contains the offscreen data.
- **offxform**, **on_xform** — These are QuickDraw GX transform objects. **offxform** has a view port list that contains just the view port for the offscreen device. **on_xform** has a view port list for the parent of the view port that's being buffered. You may expect that the view port list would be the view port being buffered and not its parent, but when drawing onscreen we've already taken into account all of the transformation and clips of the view port we're buffering, and we just need to copy the end result to the screen. This is why we use the view port's parent and not the view port proper.
- **eraser**, **marker** — These are the auxiliary shapes used to erase the offscreen buffer and to draw the incoming shapes. The shape **eraser** is of type gxFullType (a "full" shape) and is the color of the background. The **marker** (so named to complement **eraser**) is a picture shape that will be used to draw into the offscreen buffer. The reason for using the **marker** rather than swapping in the transform of the incoming shape to be **offxform** (thereby causing the shape to draw offscreen) is that the swapping operation would invalidate the caches for the incoming shape. Instead we use the property of picture shapes of redirecting any drawing to their view port list instead of the shape's own in order to cause incoming shapes to render offscreen. Furthermore, if the incoming shape is the same for every invocation of DrawShapeBuffered, we can test for it and not change the contents of the **marker**.
- **updatearea** — This is a rectangle shape used to compute the size of the offscreen buffer that is to be generated and what devices it falls on.
- **usehalftone** — This is a Boolean indicating whether to use a halftone in the offscreen buffer.
- **window**, **parent**, **screenview**, **page** — These fields hold incoming parameters to the library. The **window** field is the window in which drawing will occur. The **parent** field is a cache for the parent of the view port being buffered (see page 7-18 of *Inside Macintosh: QuickDraw GX Objects* to learn more about view port hierarchies). The **screenview** field indicates the view port that will be buffered. The **page** field is a reference to the last shape passed to DrawShapeBuffered.
- **bounds** — This field indicates the visible area of the **screenview** in the coordinate space of that view port.
- **invmap** — This is a mapping for translating between the coordinate system of the shapes being drawn in the window and the space of the window itself. If your view is zoomed in at 2x magnification, this mapping will be at 1/2 scale.
- **viewdelta** — This is the position of the upper left corner of the area being buffered, in the local coordinate system of the window. This parameter is used to adjust the drawing in the offscreen buffer so that only the correct part of the shape being buffered is drawn, and to position the content of the offscreen buffer when it's being transferred onto the screen.

Let's look at what it takes to create an offscreen buffer in the NewViewPortWBuffer routine (Listing 1). In QuickDraw GX, the place where drawing occurs (for example, the screen or an offscreen buffer) is described by a view device, so the primary purpose of the routine is to create a view device and store it in the **device** field of the viewPortBufferRecord data structure. Because we want the offscreen device that we specify to be as close as possible to the one into which we will eventually be drawing, you might think that we would go ahead and set all of the attributes of the view device now. But in fact, all that we want to concern ourselves with right now is allocating the gxViewDevice object. Later, when we get to the drawing part, we'll check the screen and our offscreen device and update the gxViewDevice object accordingly.

Listing 1. NewViewPortWBuffer

```
viewPortBuffer NewViewPortWBuffer(WindowPtr window, gxViewPort view,
      const gxColor *backgroundColor)
{
   Handle   sbHdl;

   if (sbHdl = NewHandleClear(sizeof(viewPortBufferRecord))) {
      gxInk                  background;
      gxHalftone             halftone;
      viewPortBufferRecord   *sbPtr;

      HLock(sbHdl);
      sbPtr = * (viewPortBufferRecord **) sbHdl;
      sbPtr->window = window;
      sbPtr->screenview = view;
      sbPtr->parent = GXGetViewPortParent(view);

      /* We don't allocate storage until we need it. */
      sbPtr->storage = nil;
      sbPtr->buffer = GXNewShape(gxBitmapType);
      sbPtr->group = GXNewViewGroup();
      sbPtr->view = GXNewViewPort(sbPtr->group);
      sbPtr->device = GXNewViewDevice(sbPtr->group, sbPtr->buffer);
      if (sbPtr->usehalftone = GXGetViewPortHalftone(view, &halftone))
         GXSetViewPortHalftone(sbPtr->view, &halftone);
      sbPtr->offxform = GXNewTransform();
      GXSetTransformViewPorts(sbPtr->offxform, 1, &sbPtr->view);
      sbPtr->on_xform = GXNewTransform();
      GXSetTransformViewPorts(sbPtr->on_xform, 1, &sbPtr->parent);

      background = GXNewInk();
      if (backgroundColor)
         GXSetInkColor(background, backgroundColor);
      else {
         gxColor  backcolor;

         backcolor.space = gxRGBSpace;
         backcolor.profile = nil;
         backcolor.element.rgb.red =
            backcolor.element.rgb.green =
            backcolor.element.rgb.blue = 0xFFFF;
         GXSetInkColor(background, &backcolor);
      }
      sbPtr->eraser = GXNewShape(gxFullType);
      GXSetShapeInk(sbPtr->eraser, background);
      GXDisposeInk(background);

      /* The initial bounds for the offscreen is the entire window. */
      sbPtr->bounds.left = ff(window->portRect.left);
      sbPtr->bounds.top = ff(window->portRect.top);
      sbPtr->bounds.right = ff(window->portRect.right);
      sbPtr->bounds.bottom = ff(window->portRect.bottom);
```

(continued on next page)

Listing 1. NewViewPortWBuffer *(continued)*

```
        sbPtr->updatearea = GXNewRectangle(&sbPtr->bounds);
        GXSetShapeViewPorts(sbPtr->updatearea, 1, &sbPtr->parent);
        sbPtr->marker = GXNewShape(gxPictureType);
        GXSetShapeTransform(sbPtr->eraser, sbPtr->offxform);
        GXSetShapeTransform(sbPtr->marker, sbPtr->offxform);
        GXSetShapeTransform(sbPtr->buffer, sbPtr->on_xform);
        ResetMapping(&sbPtr->invmap);

        /* The rest of the fields in the block are initialized to 0 */
        /* by the "Clear" in the NewHandleClear used to allocate this */
        /* block. */

        HUnlock(sbHdl);
    }
    return ((viewPortBuffer) sbHdl);
}
```

To create a view device we need a view group and a bitmap. Eventually we'll want to fill in all of the values of the gxBitmap object to match the screen, but for now the default values assigned to the bitmap by calling GXNewShape are sufficient.

The NewViewPortWBuffer routine also allocates a number of auxiliary objects that will be needed during the operation of the offscreen buffer. These include the following:

- a gxShape object to be used to erase the offscreen buffer
- a pair of gxTransform objects to direct drawing of incoming shapes to the offscreen buffer and of the content of the offscreen buffer to the screen

Because we'll use these objects throughout the life of the offscreen buffer, we'll do best by allocating them now and releasing them at the end. Whenever possible, you'll want to allocate objects that you'll use throughout the life of your application up front, work with them by changing their attributes, and dispose of them at the end.

DISPOSING OF THE BUFFER

When you've finished using the window and want to deallocate the memory being used by the view port buffer, you should call DisposeViewPortWBuffer.

```
void DisposeViewPortWBuffer(viewPortBuffer sb);
```

sb The object previously returned from NewViewPortWBuffer.

As shown in Listing 2, DisposeViewPortWBuffer just runs through the viewPortBufferRecord data structure and disposes of all of the objects allocated by NewViewPortWBuffer.

UPDATING THE BUFFER

When some aspect of the window in which you're drawing changes, you need to call UpdateViewPortWBuffer. In particular, if you change the clip shape or the mapping

Listing 2. DisposeViewPortWBuffer

```
void DisposeViewPortWBuffer(viewPortBuffer sb)
{
   viewPortBufferRecord   *sbPtr;

   HLock((Handle) sb);
   sbPtr = *sb;

   /* We need to dispose of all of the things that we allocated. */
   GXDisposeShape(sbPtr->marker);
   GXDisposeShape(sbPtr->eraser);
   GXDisposeTransform(sbPtr->on_xform);
   GXDisposeTransform(sbPtr->offxform);
   GXDisposeViewDevice(sbPtr->device);
   GXDisposeViewPort(sbPtr->view);
   GXDisposeViewGroup(sbPtr->group);
   GXDisposeShape(sbPtr->buffer);
   if (sbPtr->storage) DisposeHandle(sbPtr->storage);

   HUnlock((Handle) sb);
   DisposeHandle((Handle) sb);
}
```

of the viewPort object that you originally passed to NewViewPortWBuffer, you need to call UpdateViewPortWBuffer. Typically, you'll need to change the clip shape of the view port to keep QuickDraw GX from drawing shapes over the scroll bar area, and you'll need to change the mapping in order to zoom in or scroll.

```
void UpdateViewPortWBuffer(viewPortBuffer sb, gxShape clip,
                           gxMapping *displaymap);
```

sb — The object previously returned from NewViewPortWBuffer.

clip — The clip shape that should be applied when drawing into the window previously passed to NewViewPortWBuffer. Passing nil leaves the current clip shape untouched. The initial setting is for drawing to occur in the entire contents of the window (including the area typically assigned to scroll bars).

displaymap — The parameter used to update the view port buffer if you change the mapping of your window view port in order to zoom in or scroll. If nil, the current mapping is left untouched. The initial setting is the identity mapping.

DRAWING ON THE SCREEN

Now we get to the real substance of the library — the buffering routine and its supporting code.

When you want to draw on the screen, you'll call DrawShapeBuffered instead of GXDrawShape. If the memory is available to double buffer your drawing, DrawShapeBuffered will result in a flicker-free update; otherwise the routine will simply call GXDrawShape.

```
void DrawShapeBuffered(viewPortBuffer sb, gxShape page,
      const gxRectangle *updatebounds);
```

sb	The object previously returned from NewViewPortWBuffer.
page	The shape that you want to draw inside the window. This is typically a QuickDraw GX picture shape into which all of the shapes that make up a document have been collected.
updatebounds	A pointer to a QuickDraw GX rectangle indicating what area of the document is to be updated. The location of the rectangle is given in the coordinate system of the window's portRect. If nil, the code draws the area inside the clip shape passed to UpdateViewPortWBuffer.

As shown in Listing 3, the first thing that the buffering routine does is to compute the global bounds of the view port that's being buffered. Optionally, you could specify what area inside the view port you want to have buffered. Otherwise the routine attempts to draw all of the view port that's visible on the screen.

Listing 3. DrawShapeBuffered

```
void DrawShapeBuffered(viewPortBuffer sb, gxShape page,
      const gxRectangle *updatebounds)
{
   viewPortBufferRecord    *sbPtr;
   gxRectangle             bounds;

   HLock((Handle) sb);
   sbPtr = *sb;

   if (updatebounds) {
      gxMapping   map;

      GXGetViewPortMapping(sbPtr->screenview, &map);
      bounds = *updatebounds;
      bounds.left = bounds.left & 0xFFFF0000;
      bounds.right = (bounds.right + 0xFFFF) & 0xFFFF0000;
      bounds.top = bounds.top & 0xFFFF0000;
      bounds.bottom = (bounds.bottom + 0xFFFF) & 0xFFFF0000;
      MapPoints(&map, 2, (gxPoint *) &bounds);
      bounds.left = bounds.left & 0xFFFF0000;
      bounds.right = (bounds.right + 0xFFFF) & 0xFFFF0000;
      bounds.top = bounds.top & 0xFFFF0000;
      bounds.bottom = (bounds.bottom + 0xFFFF) & 0xFFFF0000;

      /* We remove the fractional part BEFORE the call to MapPoints */
      /* because we're rounding to enclose all pixels intersected */
      /* by the rectangle. Pixels are integers. Coordinates are */
      /* fractional. */
   }
   else
      bounds = sbPtr->bounds;
```

(continued on next page)

Listing 3. DrawShapeBuffered *(continued)*

```
   /* The above given bounds is in the window space -- just right. */
   GXSetRectangle(sbPtr->updatearea, &bounds);

   /* Check to see that the shape is actually visible on the screen */
   /* and then proceed to draw. */
   if (bounds.left < bounds.right && bounds.top < bounds.bottom) {
      GDHandle    screen;

      if (sbPtr->page != page) {
         GXSetPicture(sbPtr->marker, 1, &page, nil, nil, nil);
         sbPtr->page = page;
      }

      if (screen = GetDeviceList()) {
         do {
            gxViewDevice device = GXGetGDeviceViewDevice(screen);

            /* Note that we reuse the bounds in here. */
            if (GXGetShapeDeviceBounds(sbPtr->updatearea, sbPtr->parent,
                  device, &bounds))
               BufferDrawing(sbPtr, &bounds, device);
         } while (screen = GetNextDevice(screen));
      }
   }
}
```

If you haven't caught on to the fact that you can connect multiple screens to your Macintosh, the last part may be a little confusing. Once the routine has figured the global bounds of the visible part of the view port that it's buffering, it walks the device list checking to see if those bounds intersect each of the devices connected to the CPU and then calls the routine that performs the drawing (BufferDrawing, shown in Listing 4). Since most of the time a window will be completely contained within one screen, the BufferDrawing routine will be called only once per invocation of DrawShapeBuffered. The nice thing about breaking up the code this way is that the BufferDrawing routine can assume that it's drawing to a single device and therefore it's safe to make assumptions about the device's capabilities.

This approach of walking the device list is preferred to maintaining a buffer for each screen and having a routine to update the buffer list every time a window is moved. The latter approach would result in only minor performance improvements, and only when the window intersected more than one device. Since this is a rare case, the additional housekeeping isn't worth the trouble.

The key to understanding DrawShapeBuffered is the equivalence between the QuickDraw data type GDHandle and a QuickDraw GX view device. To walk the device list, the code uses the QuickDraw routines GetDeviceList and GetNextDevice. The GXGetShapeDeviceBounds routine converts a GDHandle to a view device. From the view device we can find out which area of the screen intersects the area that we're being asked to update.

The Display Manager can help you walk the device list, as discussed in the Graphical Truffles column in this issue of *develop*.•

Listing 4. BufferDrawing

```
static void BufferDrawing(viewPortBufferRecord *sbPtr,
      const gxRectangle *boundsPtr, gxViewDevice target)
{
   gxRectangle    bounds = *boundsPtr;
   long           depth, size, gxstatus;
   gxMapping      map, savemap;
   gxShape        devsh;
   gxBitmap       devbits;
   OSErr          status;
   gxPoint        viewloc;
   gxBitmap       oldbits = sbPtr->bits;

   /* Fill in all the values of sbPtr->bits. */
   ...

   viewloc.x = bounds.left;   /* These numbers are already in */
   viewloc.y = bounds.top;    /* local space. */

   /* Compute the onscreen location of the buffer. */
   ...

   /* This is the important part, allocating the actual bits. */
   size = sbPtr->bits.rowBytes * sbPtr->bits.height;
   check(size > 0);
   if (sbPtr->storage) {
      if ((* (sbPtr->storage)) != nil)
         SetHandleSize(sbPtr->storage, size);
      else {
         ReallocHandle(sbPtr->storage, size);
         nrequire(status = MemError(), TempBufferFailed);
      }
   }
   else
      require(sbPtr->storage = TempNewHandle(size, &status),
                  TempBufferFailed);
   HNoPurge(sbPtr->storage);
   HLock(sbPtr->storage);
   sbPtr->bits.image = * ((void **) sbPtr->storage);

   /* See if we need to invalidate all of the world when we do this. */
   if (oldbits.image != sbPtr->bits.image ||
       oldbits.width != sbPtr->bits.width ||
       oldbits.height != sbPtr->bits.height ||
       oldbits.rowBytes != sbPtr->bits.rowBytes ||
       oldbits.pixelSize != sbPtr->bits.pixelSize ||
       oldbits.space != sbPtr->bits.space ||
       (oldbits.set != sbPtr->bits.set && oldbits.set &&
         GXEqualColorSet(oldbits.set, sbPtr->bits.set) == false) ||
       (oldbits.profile != sbPtr->bits.profile && oldbits.profile &&
         GXEqualColorProfile(oldbits.profile, sbPtr->bits.profile)
                     == false)) {
```

(continued on next page)

Listing 4. BufferDrawing *(continued)*

```
      GXSetBitmap(sbPtr->buffer, &sbPtr->bits, nil);
      GXSetViewDeviceBitmap(sbPtr->device, sbPtr->buffer);
   }
   else {                                /* We test this one instead */
      sbPtr->bits.set = oldbits.set;     /* of the disposed one. */
      sbPtr->bits.profile = oldbits.profile;   /* Ditto */
   }

   /* Erase the offscreen bitmap, draw the shape into it, and then */
   /* copy it onscreen. */
   GXDrawShape(sbPtr->eraser);     /* Erase. */
   GXDrawShape(sbPtr->marker);     /* Buffer. */
   GXDrawShape(sbPtr->buffer);     /* Transfer -- done. */
   HUnlock(sbPtr->storage);
   HPurge(sbPtr->storage);
   if (devsh)
      GXDisposeShape(devsh);       /* Dispose of the device bitmap. */
   GXGetGraphicsError(&gxstatus);
   ncheck(gxstatus);
   if (gxstatus)
      goto DrawingFailed;
   return;

TempBufferFailed:
   GXDisposeShape(devsh);          /* Dispose of the device bitmap. */

DrawingFailed:
   GXDrawShape(sbPtr->updatearea);
   GXDrawShape(sbPtr->page);
}
```

In BufferDrawing, all of the parameters needed to create an offscreen bitmap as required by the given device are finally computed. Note that in the BufferDrawing routine there are no calls that create new objects; there are only calls that modify objects that were created when NewViewPortWBuffer was called. The modifications are done only if needed. For example, before calling GXSetTransformMapping, the library checks to see if the mapping has changed and merits updating. Without this check, the transform cache would be needlessly invalidated some of the time. Similarly, the code checks to see if any of the parameters of the bitmap for the offscreen view device have changed before calling GXSetBitmap and GXSetViewDeviceBitmap.

Changing the bitmap for the view device is one of the most expensive operations in QuickDraw GX because it invalidates most of the drawing caches. Fortunately, the check to see if the bitmap needs to be updated executes very quickly in spite of its length, and the cost of rebuilding all of the shape caches is avoided if possible.

The most confusing thing in the BufferDrawing routine is the call to the GXGetDeviceBitmap routine (omitted from Listing 4; see the full code on the CD for details) and the subsequent call to GXDisposeShape for the same object. This routine obtains a *copy* of the read-only object in QuickDraw GX that represents the bitmap for a given screen. There are two important points about this. The first is that

since we're being given a copy and not the object itself, we have to dispose of the object after we're finished with it. You may think that it would be more efficient to get the object during the initialization routine and then dispose of it when we're all done. But that's the other important point. Since the object that we have is a *copy* of the original, our copy would not be updated if the depth of the monitor was changed or the color table for the device had been updated. As a result of these two points, we're forced to allocate an object every time through our drawing loop, something that should be avoided in general.

THE REST OF THE ROUTINES

The rest of the routines in the offscreen buffering library provide support and access to some of the internal fields of the viewPortBufferRecord data structure. If you need more information on how to use these, look at the sample code included on this issue's CD.

I'll mention one other routine here. Because the internal view port created by the library is inaccessible from the outside, the routine SetViewPortWBufferDither is provided to change the dither level of the view port. If you need to change other attributes of the offscreen view port, use the SetViewPortWBufferDither routine as a template.

```
void SetViewPortWBufferDither(viewPortBuffer sb, const long ditherlevel);
```

sb	The object previously returned from NewViewPortWBuffer.
ditherlevel	The dithering level to set the offscreen view port to.

THINGS TO TRY IN THE CODE

If the code presented so far doesn't meet your particular needs, you may want to try changing or fine tuning it. Here are some suggestions and observations about things that you may want to try.

BITMAP ALLOCATION

Currently the code looks for memory in the MultiFinder temporary memory heap (Temp Mem) and will back down in case it can't obtain the memory for the offscreen buffer. If you need to guarantee that your drawing will be screen buffered, you'll need to change the memory allocation code inside the BufferDrawing routine.

INTEGRATED ERROR HANDLING

There are two places where memory allocation can trip the screen buffering library. If the library fails to allocate enough memory to hold its data structures, it will return nil from NewViewPortWBuffer. You may need to change this to better fit in with your application's error model.

The library will handle a failure to allocate the offscreen bitmap by resorting to drawing with GXDrawShape. If you want something different, see "Bitmap Allocation" above.

DEEP POCKETS

If the original data that you'll be working with requires more bits than are on the display that you're running on, you may want to create an offscreen buffer that's deeper than the screen and then take advantage of the dithering or halftoning mechanisms in QuickDraw GX to allow user manipulation. The code that checks

for changes in the screen view port's halftone should give you a good idea of how to do that.

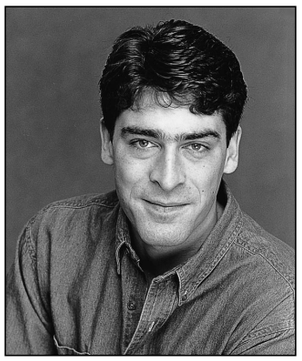
STEADY, NOW

Now you understand how to use double buffering to prevent flicker in your QuickDraw GX application. You may need to do some fine tuning of the screen buffering library to fit your purposes, but the result will be worth it. Users will appreciate the more professional look of your application and their eyes won't tire as quickly as they peer at a flicker-free screen.

RELATED READING

- *"Getting Started With QuickDraw GX"* by Pete "Luke" Alexander, *develop* Issue 15.
- *Inside Macintosh: QuickDraw GX Objects* and *Inside Macintosh: QuickDraw GX Graphics* (Addison-Wesley, 1994).

Thanks to our technical reviewers Dave Bice, Brian Chrisman, Tom Dowdy, David Hayward, and Ingrid Kelly.•



GRAPHICAL TRUFFLES

The Display Manager

MIKE MARINKOVICH

A major change is taking place on the screen, which your application might not even know about! With the help of the Display Manager, the user can use the Monitors control panel to rearrange displays, make resolution switches, add or remove a display, and move the menu bar from one display to another — all without rebooting. However, the ease of changing a display for the user poses new challenges for the developer if an application relies on a graphics device's bounding rectangle to position, zoom, and grow its windows.

To meet this challenge, the Display Manager provides several new functions that make it easier to gather information about the display environment and implement changes. I'll describe some of the more commonly used functions in this column. I'll also discuss how to use a notification event to find out when a display has changed (an example is included on this issue's CD).

Two versions of the Display Manager are currently implemented in the system software. The information in this column applies to both versions. Display Manager version 1.0 is available on all PowerPC™ processor–based Macintosh computers and Color QuickDraw–capable Macintosh computers running System 7.5. Display Manager 2.0 is available on PCI-based computers running System 7.5.2. To determine whether the Display Manager is available, call Gestalt with the selector gestaltDisplayMgrAttr and check the gestaltDisplayMgrPresent bit of the response. To determine which version you have, call Gestalt with the selector gestaltDisplayMgrVers.

MORE FUNCTIONS, LESS CODE

The Display Manager includes several new functions that greatly simplify tasks that used to take a lot of code. For example, many applications need to query screen devices for bounding rectangles, pixel depths, and a variety of other things. Prior to the Display Manager, an application could use the GetDeviceList function to retrieve the first graphics device record in the device list and call GetNextDevice for subsequent devices in the list. The application would then need to use the Device Manager to determine whether the device was a screen device and whether it was active. With the Display Manager, you can do all this with two functions: DMGetFirstScreenDevice and DMGetNextScreenDevice.

```
GDHandle     aDevice;

aDevice =
   DMGetFirstScreenDevice(dmOnlyActiveDisplays);
while (aDevice != nil) {
   // Do something with the device.
   ...
   // Get the next device in the list.
   aDevice = DMGetNextScreenDevice(aDevice,
                          dmOnlyActiveDisplays);
}
```

The Display Manager also introduces two functions that make it easier to retrieve information about the attached displays and to change their characteristics: DMCheckDisplayMode and DMSetDisplayMode.

DMCheckDisplayMode determines whether a specific display mode and pixel depth are supported by the supplied graphics device. (A display mode is a combination of several interrelated display characteristics, such as resolution and scan timing.) This function has two output parameters: modeOk and switchFlags. If the Boolean modeOk parameter is true, the screen device supports the requested display mode. The switchFlags parameter contains two flag bits that should be checked with the constants kNoSwitchConfirmBit and kDepthNotAvailableBit.

- If kNoSwitchConfirmBit isn't set, the requested mode is an optional mode and is only shown in the mode list of the Monitors control panel when the Option key is pressed (an optional mode requires confirmation from the user before it's allowed).

MIKE MARINKOVICH (marink@apple.com) is a member of the Printing, Imaging, and Graphics (PIGS) group in Developer Technical Support at Apple. He's been whiling away his days (and many of his evenings) coming to grips with the Display Manager and other QuickDraw-related esoterica. When not indulging in his hobby, which also happens to be playing around with the Toolbox and programming his Macintosh, Mike spends his time exploring the San Francisco Bay Area in his trusty Subaru. Mike's from Seattle and misses the rain.

- kDepthNotAvailableBit indicates whether the requested pixel depth is available with the requested display mode.

Once your application knows that the requested display mode and pixel depth are available, you can use the DMSetDisplayMode function to reconfigure the video display. If you pass 0 for the mode parameter, the Display Manager uses the device's current display mode.

If you like to change the display mode and pixel depth often, you can save the configuration and retrieve it at startup with the DMSaveScreenPrefs function. This function requires three parameters, which all take the value of NULL since they're private to the Display Manager. (Go figure.)

Identifying displays. Many of the Display Manager functions require a display ID (type DisplayIDType) as a parameter. A display ID is a long integer that uniquely identifies a screen display. Affiliating a display ID with a graphics device can be useful in cases where the graphics device might change or isn't available. You can obtain a display ID with the function DMGetDisplayIDByGDevice, which requires a graphics device as a parameter. Or you can retrieve the graphics device corresponding to a given display ID by calling DMGetGDeviceByDisplayID. Both functions require the Boolean parameter failToMain.

- If you set failToMain to true and the routine can't find what it's looking for (either the graphics device or the display ID), the routine returns information about the main graphics device rather than returning an error.
- If you set failToMain to false and the routine can't find what it's looking for, it will return kDMDisplayNotFoundErr. (For example, when a PowerBook goes to sleep, the display might be removed.)

KEEPING UP WITH THE CHANGES

Now that the user is able to change a screen display without restarting, your application may want to reposition and resize its windows, update internal display-related data structures, or update nonstandard window definitions on the fly.

If desired, the Display Manager can automatically adjust the positions of the windows that were onscreen before the change to keep them onscreen after the change, but it may not put them in the best possible positions. However, if you want to reposition and resize your windows yourself, you need to set the isDisplayManagerAware flag in your application's SIZE resource and install a callback procedure or an Apple event handler in your application so that you'll know when a display has changed.

Your application registers a callback procedure with the Display Manager function DMRegisterNotifyProc. The display notification procedure takes a Display Notice Apple event parameter describing the changes that were made to the display. The notification callback is especially useful for control panels and other instances where high-level event handling in an event loop isn't possible. Another benefit of the notification callback is that your application is informed on a more timely basis than through a high-level event, thus giving the appearance of seamless integration with the Display Manager.

If you're using Display Manager 1.0, you're not notified about depth changes, and A5 isn't restored when you receive the notification callback.•

You can also receive and process Display Notice events through an Apple event handler. Display Notice event handlers are installed like any other Apple event handlers, with the AEInstallEventHandler function:

```
err = AEInstallEventHandler(kCoreEventClass,
   kAESystemConfigNotice,
   NewAEEventHandlerProc(DoAEDisplayConfigChange),
   0, false);
```

To enable high-level events in your application, you need to set the isHighLevelEventAware flag in the SIZE resource. (You'll also need to support the required Apple events described in *Inside Macintosh: Interapplication Communication.*)

Whether your application uses a notification callback or a high-level event handler, a Display Notice Apple event is passed to your routine. You can obtain a list of descriptor records (an AEDescList) from the Display Notice event with the AEGetParamDesc function. Each descriptor record holds two additional keyword-specific descriptor records:

- keyDisplayOldConfig, which is a record of the display's previous state
- keyDisplayNewConfig, which is a record of the display's current state

You can obtain these records one at a time with the function AEGetNthDesc.

To move and resize your application's windows, you need to know which graphics device was affected, the old and new bounding rectangles of the device, and

Listing 1. Handling the Display Notice event

```
OSErr HandleNotification(AppleEvent *event)
{
   OSErr           err;
   GrafPtr         oldPort;
   AEDescList      displayList, aDisplay;
   AERecord        oldConfig, newConfig;
   AEKeyword       tempWord;
   DisplayIDType   displayID;
   unsigned long   returnType;
   long            count;
   Rect            oldRect, newRect;

   GetPort(&oldPort);

   // Get a list of the displays from the Display Notice Apple event.
   err = AEGetParamDesc(event, kAEDisplayNotice, typeWildCard, &DisplayList);

   // How many items in the list?
   err = AECountItems(&displayList, &count);

   while (count > 0) {
      // Loop through the list.
      err = AEGetNthDesc(&displayList, count, typeWildCard, &tempWord, &aDisplay);

      // Get the old rect.
      err = AEGetNthDesc(&aDisplay, 1, typeWildCard, &tempWord, &oldConfig);
      err = AEGetKeyPtr(&oldConfig, keyDeviceRect, typeWildCard, &returnType, &oldRect, 8, nil);

      // Get the display ID so that we can get the GDevice later.
      err = AEGetKeyPtr(&oldConfig, keyDisplayID, typeWildCard, &returnType, &displayID, 8, nil);

      // Get the new rect.
      err = AEGetNthDesc(&aDisplay, 2, typeWildCard, &tempWord, &newConfig);
      err = AEGetKeyPtr(&newConfig, keyDeviceRect, typeWildCard, &returnType, &newRect, 8, nil);

      // If the new and old rects are not the same, we can assume that the GDevice has changed,
      // and the windows need to be rearranged.
      if (err == noErr && !EqualRect(&newRect, &oldRect))
         HandleDeviceChange(displayID, &newRect);

      count--;
      err = AEDisposeDesc(&aDisplay);
      err = AEDisposeDesc(&oldConfig);
      err = AEDisposeDesc(&newConfig);
   }

   err = AEDisposeDesc(&displayList);
   SetPort(oldPort);

   return err;
}
```

possibly the pixel depth. All the information about the affected graphics device can be obtained from the descriptor list with keyword-specific descriptor constants, which are defined in the Displays.h universal header file. You call AEGetKeyPtr with the various descriptor constants to extract the information you need. In particular, the constant keyDeviceRect extracts the bounding rectangle, and keyDisplayID extracts the display ID. As previously mentioned, you can convert a display ID to a graphics device with the function DMGetGDeviceByDisplayID.

Listing 1 shows an example of what to do after receiving a Display Notice event from a notification callback or a high-level event handler.

WHAT TO DO NOW

The sample code on this issue's CD should provide a starting point for how to handle display notification events in your application. Additional documentation and sample code for the Display Manager are provided in the Display Manager Development Kit, which is also on the CD.

The Mac OS Software Developer's Kit incudes the Display Manager Development Kit along with a lot of other development software. The Mac OS SDK is now part of the Developer CD Series (included in the Apple Developer Mailing, which is available through the Apple Developer Catalog).•

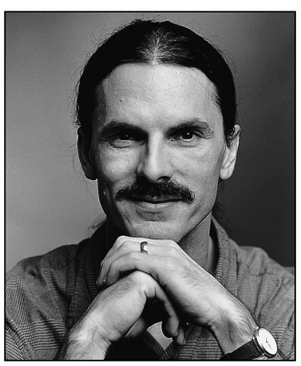
To learn more about what the Display Manager can do for you, you should also take a look at the Displays.h universal header file.

Now there's no excuse for your application to be in the dark about changes taking place on the screen. So why not keep your users happy and take advantage of the help that the Display Manager can give you?

Thanks to Eric Anderson, David Hayward, and Ian Hendry for reviewing this column.•

NURB Curves: A Guide for the Uninitiated

*QuickDraw 3D supports a mathematical model for arbitrary curves and surfaces known as NURB (**n**on**u**niform **r**ational **B**-splines). NURB curves are flexible and powerful, but using them effectively requires some understanding of the underlying mathematical theory. This article presents an intuitive introduction to the mathematical concepts of the NURB model and how to use them in your QuickDraw 3D programs.*



PHILIP J. SCHNEIDER

One of the more powerful features of QuickDraw 3D is its ability to work with curves and surfaces of arbitrary shape. The mathematical model it uses to represent them is known as *NURB*, for **n**on**u**niform **r**ational **B**-splines. The NURB model is flexible and powerful, but for those unfamiliar with the mathematics, it can appear dauntingly complex. The existing books and articles on the subject tend to be rigorous, lengthy, and theoretical, and often seem to require that you already understand the subject in order to follow the explanations.

The mathematics really aren't so frightening, though, once you understand them. The aim of this article is to give you an intuitive understanding of how NURB curves work. Later in the article, we'll look at some code to show you how you can start using NURB curves in your own programs — but you really do need to understand the theory before you can start putting it to practical use. So please be patient while we slog through the mathematical concepts: I promise we'll get around to some actual programming before we're through. Note also that this article is only about NURB *curves;* perhaps a future article will cover NURB *surfaces* and how to use curves and surfaces together.

Some writers also use the *s* from "spline," resulting in the acronym *NURBS* — but most avoid this usage because phrases like "a NURBS curve" sound awkward, and "a NURBS surface" sounds perfectly hideous.•

WHY NURB CURVES?

Like any graphics package, QuickDraw 3D offers low-level geometric primitives for objects such as lines, points, and triangles. Because the representations of these

PHILIP J. SCHNEIDER (pjs@apple.com) is the longest-surviving member of the QuickDraw 3D team. He lives with his wife Suzanne and son Dakota out in the middle of a redwood forest in the Santa Cruz mountains, pretending he does so because "it's more affordable." People who are taken in by that malarkey probably also believe he doesn't like driving a two-lane country highway to work every day, and would rather be stuck in traffic jams on the interstate freeway with flatlanders. His current projects include trying to single-handedly bring up the worldwide level of computer technology to what he finds in the science fiction novels he reads voraciously, and teaching his 18-month-old son to change his own wet diapers in the middle of the night.•

objects are mathematically exact — lines being defined by their two endpoints, triangles by their three vertices, and so forth — they're resolution independent and unaffected by changes in position, scale, or orientation.

The low-level primitives can also be used to define arbitrarily shaped objects, such as a football or an automobile hood, but at the cost of these desirable mathematical properties; for example, a circle that's approximated by a sequence of line segments will change its shape when rotated. One of the advantages of NURB curves is that they offer a way to represent arbitrary shapes while maintaining mathematical exactness and resolution independence. Among their useful properties are the following:

- They can represent virtually any desired shape, from points, straight lines, and polylines to conic sections (circles, ellipses, parabolas, and hyperbolas) to free-form curves with arbitrary shapes.
- They give you great control over the shape of a curve. A set of *control points* and *knots*, which guide the curve's shape, can be directly manipulated to control its smoothness and curvature.
- They can represent very complex shapes with remarkably little data. For instance, approximating a circle three feet across with a sequence of line segments would require tens of thousands of segments to make it look like a circle instead of a polygon. Defining the same circle with a NURB representation takes only seven control points!

In addition to drawing NURB curves directly as graphical items, you can use them in various other ways that exploit their useful mathematical properties, such as for guiding animation paths or for interpolating or approximating data. You can also use them as a tool to design and control the shapes of three-dimensional surfaces, for purposes such as

- surfaces of revolution (rotating a two-dimensional curve around an axis in three-dimensional space)
- extruding (translating a curve along a curved path)
- trimming (cutting away part of a NURB surface, using NURB curves to specify the cut)

CURVES 101

Before we go into the specifics of NURB curves, let's review some of the basics of curve representation in general.

Although QuickDraw 3D supports three-dimensional NURB curves, we'll limit all of our examples and discussions here to two dimensions. But everything we say about two-dimensional curves applies in three dimensions as well — the two-dimensional versions are just easier to visualize and easier to draw.

A BIT OF HISTORY

Back in the days before computers, architects, engineers, and artists would draw their designs for buildings, roads, machine parts, and the like by using pencil, paper, and various drafting tools. These tools included rulers and T-squares for drawing straight lines, compasses for drawing circles and circular arcs, and triangles and protractors for making precise angles.

Of course, a lot of interesting-shaped objects couldn't be drawn with just these simple tools, because they had curved parts that weren't just circles or ellipses. Often, a curve

was needed that went smoothly through a number of predetermined points. This problem was particularly acute in shipbuilding: although a skilled artist or draftsman could reliably hand-draw such curves on a drafting table, shipbuilders often needed to make life-size (or nearly life-size) drawings, where the sheer size of the required curves made hand drawing impossible. Because of their great size, such drawings were often done in the loft area of a large building, by a specialist known as a loftsman. To aid in the task, the loftsman would employ long, thin, flexible strips of wood, plastic, or metal, called *splines*. The splines were held in place with lead weights, called *ducks* because of their resemblance to the feathered creature of the same name (see Figure 1).

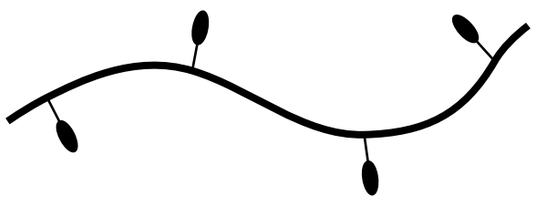


Figure 1. A draftsman's spline

The resulting curves were smooth, and varied in curvature depending on the position of the ducks. As computers were introduced into the design process, the physical properties of such splines were investigated so that they could be modeled mathematically on the computer.

DIRECT FUNCTIONS

Our goal is to represent curves in a mathematically precise fashion. One simple way is to think of the curve as the graph of a function:

$y = f(x)$

Take a simple one like the trigonometric sine function:

$y = \sin x$

By plotting the value of the function for various values of x and connecting them smoothly, we obtain the curve shown in Figure 2.

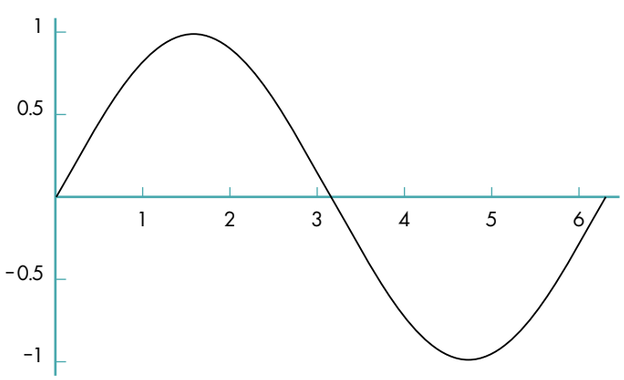


Figure 2. Plot of sine function values

In the case of curves drawn by the spline method, it turned out that with some reasonable simplifying assumptions, they could be mathematically represented by a series of cubic (third-degree) polynomials, each having the form

$$y = Ax^3 + Bx^2 + Cx + D$$

At this point, the standard references typically go into a long, involved development of this idea into what are known as *cubic spline curves*, eventually leading to the theory of NURB curves. Such explanations are interesting, but not terribly intuitive. If you're interested in pursuing this subject further, you'll find a good discussion in *Mathematical Elements for Computer Graphics*. (Complete information on this and other literature references in this article can be found in the bibliography at the end.)

PARAMETRIC FUNCTIONS

Using direct functions to represent a curve fits our criterion of being mathematically exact, but it has one serious drawback: since we can have only one value of y for each value of x, our curves can't loop back on themselves. Thus, although we can make some nice smooth curves this way, there are a lot of interesting curves we *can't* make — not even something as simple as a circle.

An alternative method, and the one we'll be using, is to define the curve with a *parametric function*. In general, such functions have the form

$$Q(t) = \{X(t), Y(t)\}$$

where $X(t)$ and $Y(t)$ are functions of the parameter t (hence *parametric*). Given a value of t, the function $X(t)$ gives the corresponding value of x, and $Y(t)$ the value of y. One way to understand such functions is to imagine a particle traveling across a sheet of paper, tracing out a curve. If you think of the parameter t as representing time, the parametric function $Q(t)$ gives the $\{x, y\}$ coordinates of the particle at time t. For example, defining the functions $X(t)$ and $Y(t)$ as

$$X(t) = \cos t$$
$$Y(t) = \sin t$$

produces a circle, as you can verify by plugging in some values of t between 0 and 2π and plotting the results.

SMOOTHNESS

One very important motivation for using NURB curves is the ability to control smoothness. The NURB model allows you to define curves with no kinks or sudden changes of direction (such as an airplane-wing cross section) or with precise control over where kinks and bends occur (sharp corners of machined objects, for instance).

We all know (or think we do) what a nice, smooth curve looks like: it has no kinks or corners. If we were to sit on that moving particle as it traces out a parametric curve, we would experience a nice smooth ride with no stopping, restarting, or sudden changes in speed or direction: we wouldn't be heading north, say, and then turn completely east in an instant. This intuitive notion can be expressed in precise mathematical terms: Imagine an arrow that always points in the direction in which our hypothetical particle is traveling as it moves along the curve. Mathematically, the direction arrow corresponds to the tangent of the curve, which can be computed as the derivative of the curve's defining function with respect to the time parameter *t:*

$$Q'(t)$$

In Figure 3, for example, the point on the curve corresponding to time t_A is labeled as $Q(t_A)$, and the direction vector (tangent) at that point as $Q'(t_A)$. If the tangent doesn't jump suddenly from one direction to another, the curve's function is said to have *first-derivative continuity*, denoted by C^1: this corresponds to our intuitive notion of smoothness.

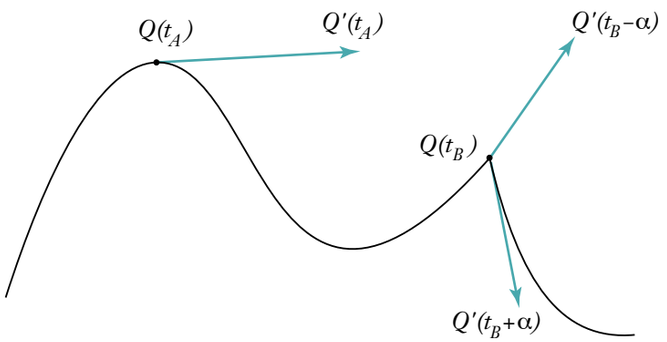


Figure 3. Tangent (derivative) of a curve

Now look at the point marked $Q(t_B)$, where there's a visible kink in the curve. The direction vector just a tiny bit to the left of that point, $Q'(t_B-\alpha)$, is wildly different from the one just a tiny bit to the right, $Q'(t_B+\alpha)$. In fact, the direction vector jumps instantaneously from one direction to another at point $Q(t_B)$. Mathematically, this is called a *discontinuity*.

Many of you will recall from your college calculus that the derivative of a function is also a function, whose degree is one less than that of the original function. For example, the derivative of a fourth-degree function is a third-degree function. The derivative of the derivative, called the second derivative, will then be of degree 2. This second derivative may or may not be continuous: if it is, we say that the original function has *second-derivative continuity*, or C^2. As the first derivative describes the direction of the curve, the second derivative describes how fast that direction is changing. The second derivative thus characterizes the curve's degree of curvature, and so a C^2-continuous curve is said to have *curvature continuity*. We'll come back to these important concepts after we've introduced NURB curves themselves.

NURB CURVES

Now that we know how parametric functions work, let's see how we can use them to build up a definition for NURB curves. If we call our function Q, the left side of our equation will look like this:

$Q(t) =$

where t is a parameter representing time. By evaluating this function at a number of values of t, we'll get a series of $\{x, y\}$ pairs that we can use to plot our curve, as shown in Figure 4. Now all we have to do is define the right-hand side.

CONTROL POINTS

One of the key characteristics of NURB curves is that their shape is determined by (among other things) the positions of a set of points called *control points*, like the ones labeled B_i in Figure 5. As in the figure, the control points are often joined with connecting lines to make them easier to see and to clarify their relationship to the curve. These connecting lines form what's known as a *control polygon*. (It would

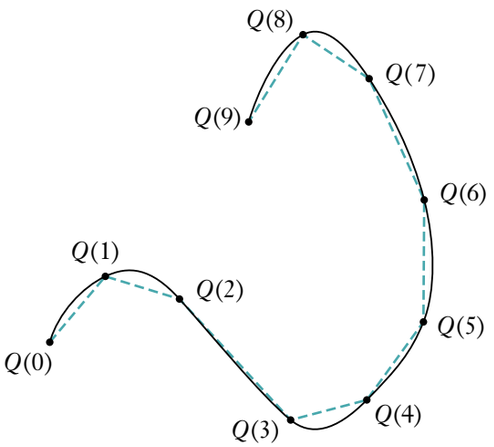


Figure 4. Plotting a parametric function

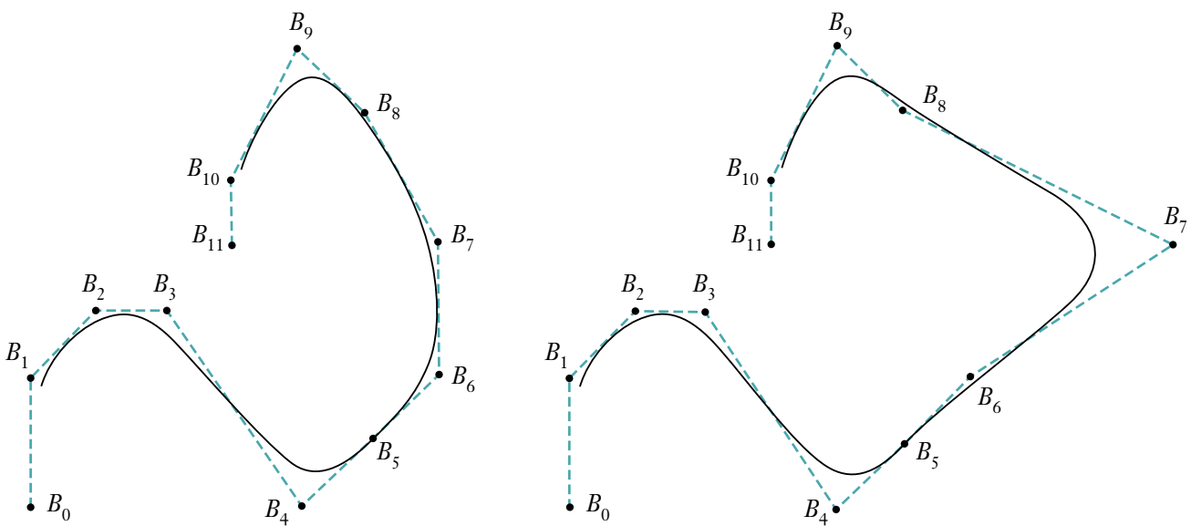


Figure 5. Defining a curve with control points

actually make more sense to call it a "control polyline," but the other is the conventional term.)

The second curve in Figure 5 is the same curve, but with one of the control points (B_7) moved a bit. Notice that the curve's shape isn't changed throughout its entire length, but only in a small neighborhood near the changed control point. This is a very desirable property, since it allows us to make localized changes by moving individual control points, without affecting the overall shape of the curve. Each control point influences the part of the curve nearest to it but has little or no effect on parts of the curve that are farther away.

One way to think about this is to consider how much influence each of the control points has over the path of our moving particle at each instant of time. At any time *t*, the particle's position will be a weighted average of all the control points, but with the points closer to the particle carrying more weight than those farther away. We can express this intuitive notion mathematically this way:

$$Q(t) = \sum_{i=0}^{n-1} B_i N_{i,k}(t)$$

In other words, to find the position of the moving particle at a particular time, add up the positions of all the control points (B_i) but vary the strength of each point's contribution over time ($N_{i,k}(t)$). We'll explain the meaning of the subscript k shortly.

The bounding volume returned by QuickDraw 3D for all other geometric primitives is a volume that encloses the primitive itself. For NURB curves, however, the returned bounding volume encloses the curve's control points, rather than the curve itself. This is done for historical reasons, and is the normal practice in 3D graphics packages.•

BASIS FUNCTIONS

The function $N_{i,k}(t)$, which determines how strongly control point B_i influences the curve at time t, is called the *basis function* for that control point. In fact, the B in "B-splines" stands for "basis." The value of this function is a real number such as 0.5, so that a particular point $Q(t)$ can be defined as, say, 25% of one control point's position, plus 50% of another's, plus 25% of yet a third's. To complete our NURB equation, we have to specify the basis function for each control point.

So how do we go about defining the basis functions? Remember that we want each region of the NURB curve to be a *local* average of some small number of control points close to that region. When the moving particle is far away from a given control point, that control point has little influence on it; as the particle gets closer, the control point affects it more and more. Then the effect diminishes again as the particle recedes past the control point.

Up to now, we've been using the words "near" and "far" in a rather vague way, but the time has come to pin them down more rigorously. Because we've defined our curve parametrically with respect to time, we can regard what we've been calling a "part" or "region" of the curve as a portion of the time interval the curve covers. For example, if our curve goes from time $t = 0.0$ to $t = 10.0$, we can specify a particular region as extending from, say, $t = 3.3$ to $t = 7.5$. So we can say, for instance, that a control point B_i is centered at time $t = 5.0$ and has an effect in the range from $t = 3.3$ to $t = 7.5$.

Figure 6 shows a typical example of what a basis function might look like: it has its maximum effect at some definite point in time and tapers off smoothly as it gets farther away from that point. If you were awake during your college statistics course, you might recognize this as the familiar "bell curve" that we all learned to know and loathe. The curve $N_{i,k}(t)$ in the figure shows that control point B_i has its greatest effect (about 95%) at time $t = 3.0$ and tapers off to about 50% at $t = 1.7$ and $t = 4.3$.

Since each control point has its own basis function, a NURB curve with, say, five control points will have five such functions, each covering some region of the curve (that is, some interval of time). At time $t = 2.3$ in Figure 7, for example, control point B_0 has a weight of about 0.2, B_1 about 0.7, and B_2 about 0.05. As t goes from 0.0 to 7.0, each control point's effect on the shape of the curve is initially 0, increases gradually to a maximum, and then gradually tapers off again to 0 as we reach the end of its effective range.

KNOTS

Notice that all of the basis functions in Figure 7 have exactly the same shape and cover equal intervals of time. In general, we'd like to be able to vary the width of the

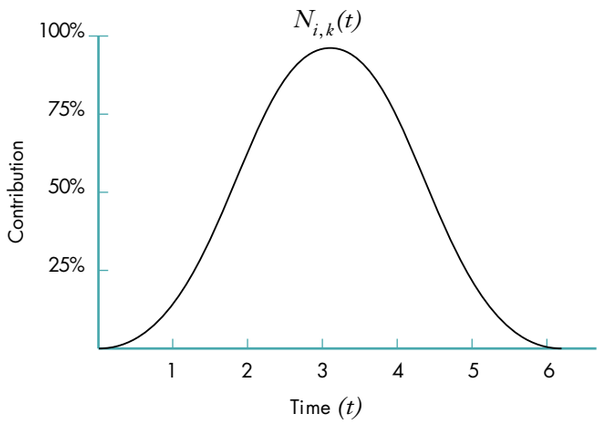


Figure 6. Basis function for a control point

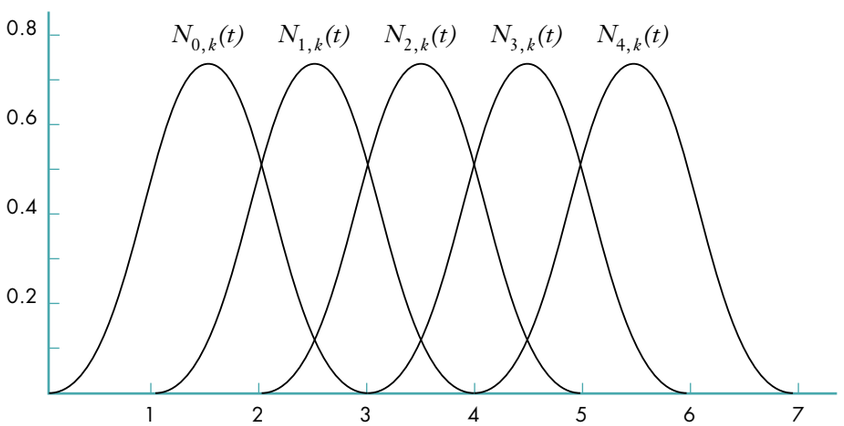


Figure 7. Uniform basis functions for a set of control points

intervals (so that some control points affect a larger region of the curve and others a smaller region) and the maximum height of the curves (so that some control points affect the shape of the curve more strongly than others). That's where the *NU* in *NURB* comes from: it stands for **n**on**u**niform.

The solution is to define a series of points that partition the time into intervals, which we can then use in the basis functions to achieve the desired effects. By varying the relative lengths of the intervals, we can vary the amount of time each control point affects the particle. The points demarcating the intervals are known as *knots*, and the ordered list of them is a *knot vector* (Figure 8). The knot vector for the basis functions shown in Figure 7 is {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0}. This is an example of a *uniform knot vector*, which is why all the functions in the figure cover equal intervals of time. Figure 9 shows an example of a curve created with such a knot vector.

Figure 8. A knot vector

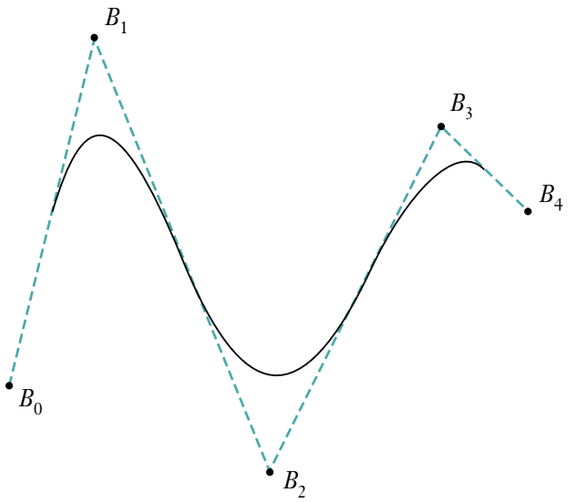


Figure 9. NURB curve with uniform knot vector

If we change the knot vector to {0.0, 1.0, 2.0, 3.75, 4.0, 4.25, 6.0, 7.0}, we get a set of nonuniform basis functions like the ones shown in Figure 10, and a curve that looks like Figure 11 (using the same set of control points as in Figure 9). Notice that the basis functions $N_{2,3}(t)$ and $N_{3,3}(t)$, associated with control points B_2 and B_3, respectively, are taller and narrower than the others. If you compare Figures 9 and 11, you'll see that the curve in Figure 11 is pulled more strongly toward control points B_2 and B_3 than the one in Figure 9. This is because the basis functions for these control points have a greater maximum value. Also, the curve rapidly approaches these control points and rapidly moves away: compare how tightly curved it is near these points, relative to the curve in Figure 9. This is a result of the narrower basis functions for these two control points: intuitively, our moving particle has to traverse more space in relatively less time. Looking at the knot vector, you can see that the knot intervals for these two control points are narrower than the others — {3.75, 4.0} and {4.0, 4.25} — meaning that their effects on the curve are concentrated in shorter time intervals.

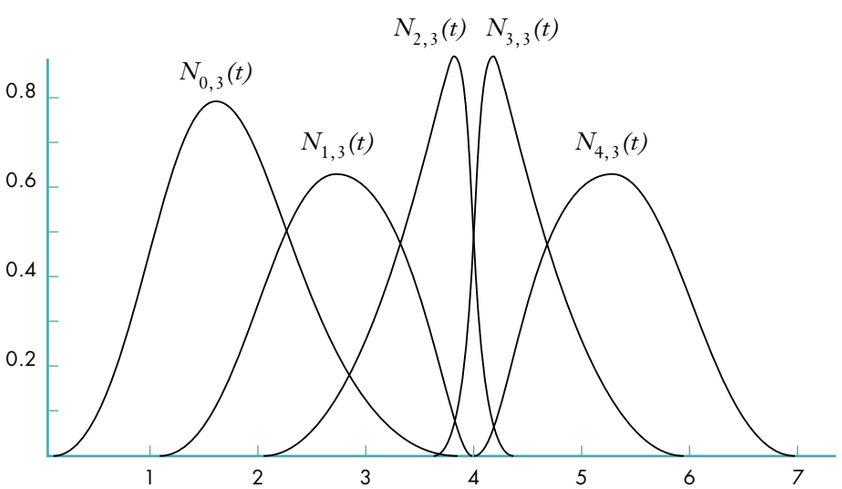


Figure 10. Nonuniform basis functions for a set of control points

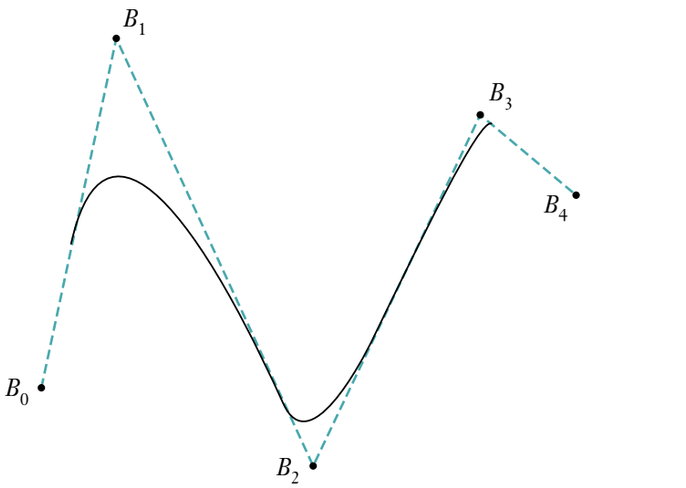


Figure 11. NURB curve with nonuniform knot vector

DEFINING THE BASIS FUNCTIONS

We're now ready to complete our definition of a NURB curve by giving an exact specification of the basis functions. In some respects, we're free to use any sort of functions we'd like, but by choosing them carefully, we can get certain desirable effects. The definitions we'll be using are as follows:

$$N_{i,1}(t) = \begin{cases} 1 & \text{if } x_i \le t < x_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

$$N_{i,k}(t) = \frac{(t - x_i)N_{i,k-1}(t)}{x_{i+k-1} - x_i} + \frac{(x_{i+k} - t)N_{i+1,k-1}(t)}{x_{i+k} - x_{i+1}}$$

where x_i is the conventional notation for the ith knot in the knot vector.

This definition has a lot of stuff in it, and lots of subscripts — we're getting into the real theoretical aspects of NURB curves here. Notice that the functions for higher values of the subscript k (called the *order* of the basis function) are built up recursively from those of lower orders. If k is the highest order of basis function we define, the resulting NURB curve is said to be of order k or of *degree* k–1. At the very bottom of the hierarchy, the functions of order 1 are simply 1 if t is between the ith and (i+1)st knots, and 0 otherwise.

The specifics of this particular set of basis functions, and how they came to be this way, are beyond the scope of this article; if you're interested in learning more, you'll find all the detail you could possibly want (and then some) in *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*. However, we can at least mention a number of important characteristics that this choice of basis functions exhibits:

- At any time t, the values of all the basis functions add up to exactly 1.
- If all control points have positive weights, the curve is contained within a bounding region known as the *convex hull*. (See the book cited above for details.)
- At any time t, no more than k basis functions affect the curve, where k is the order of the curve.
- A curve of order k is defined only where k of the basis functions are nonzero.

This last characteristic is of more than theoretical interest: a cubic (degree-3 or order-4) NURB curve with a knot vector of, say, {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0} only goes from t = 3.0 to t = 4.0! The rule is that the curve begins at the kth knot from the beginning of the knot vector and ends at the kth knot from the end.

KNOTS AND KINKS

I should point out here that nonuniform knot vectors aren't really very useful for controlling the shape of a curve. (In fact, moving control points around directly isn't that useful, either — but we'll get to that later.) Instead, nonuniform knot vectors have two important uses:

- You've probably noticed that all of our NURB curves so far have had their endpoints just "floating in space"; that is, the curve's endpoints don't coincide with any control point. In real life, though, we generally want to be able to control the exact placement of the endpoints, and most often we want them to coincide exactly with the first and last control points.
- You may also have noticed that the curves displayed so far are quite smooth. While this is usually a good thing, we sometimes need to create a curve with a kink or corner.

We can accomplish both of these goals by using a rather extreme case of nonuniformity: giving several consecutive knots the same value of t! For example, a knot vector like {0.0, 0.0, 0.0, 3.0, 4.0, 5.0, 6.0, 7.0} produces a set of basis functions like those in Figure 12 and a curve (using the same control points as before) that looks like Figure 13. Looking at Figure 12, you can see that at t = 0, the basis functions associated with all but the first control point have a 0 value — so basis function $N_{0,3}(t)$ (the one for control point B_0) has total control over the curve. Thus the curve at t = 0 coincides with the first control point.

If we bunch up some knots in the middle of the knot vector {0.0, 1.0, 2.0, 3.0, 3.0, 5.0, 6.0, 7.0}, we get the basis functions shown in Figure 14 and the curve in Figure 15. At t = 3.0, all the basis functions except $N_{2,3}(t)$ have a 0 value — so control point B_2 is the only one to affect the curve at that instant, and thus the curve coincides with that control point.

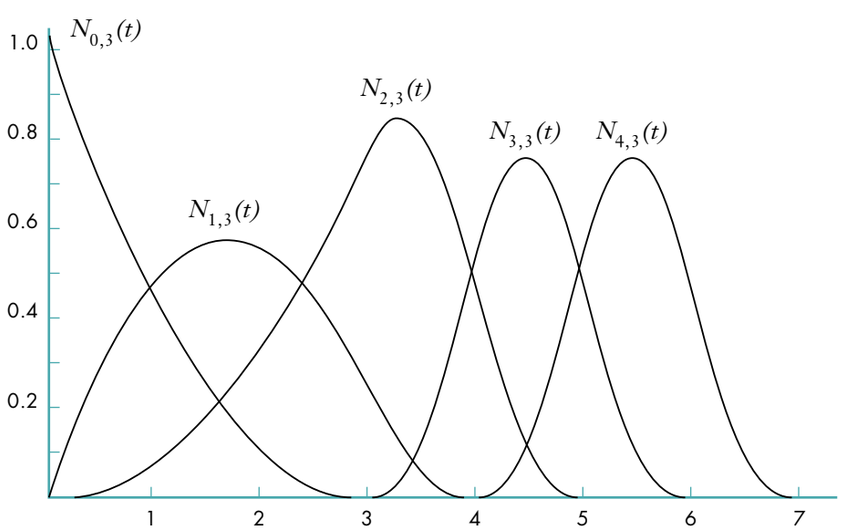


Figure 12. Basis functions for a curve with multiple identical knots at the beginning

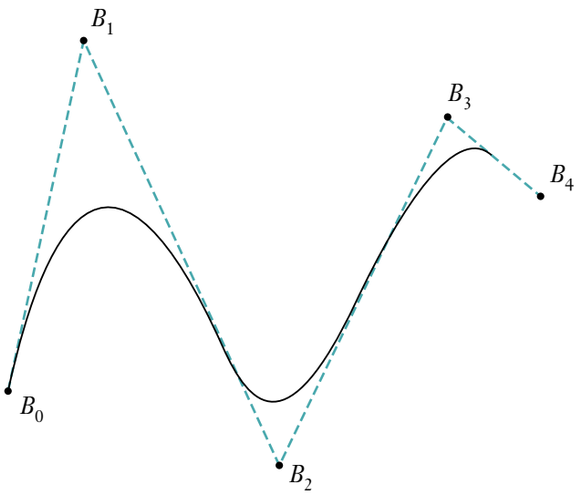


Figure 13. NURB curve with multiple identical knots at the beginning

In mathematical terms, continuity (smoothness) is an issue only at the joints defined by the curve's knots, where two segments of the curve meet; between the joints, the curve is perfectly smooth and continuous. A typical curve, in which each joint corresponds to a single knot, has continuity C^{n-1} where n is the degree of the curve. So a cubic (degree-3 or order-4) curve has second-derivative continuity (C^2) at each joint if all the knots are distinct. If two knots coincide, the continuity at that joint goes down by one degree; if three coincide, the continuity goes down another degree; and so on.

This means you can put a kink in the curve at a particular point by adding knots to the knot vector at that point. Later, we'll look at some code that shows how to do this. We'll also see how you can use this same technique of knot insertion to convert a curve from NURB to Bézier representation.

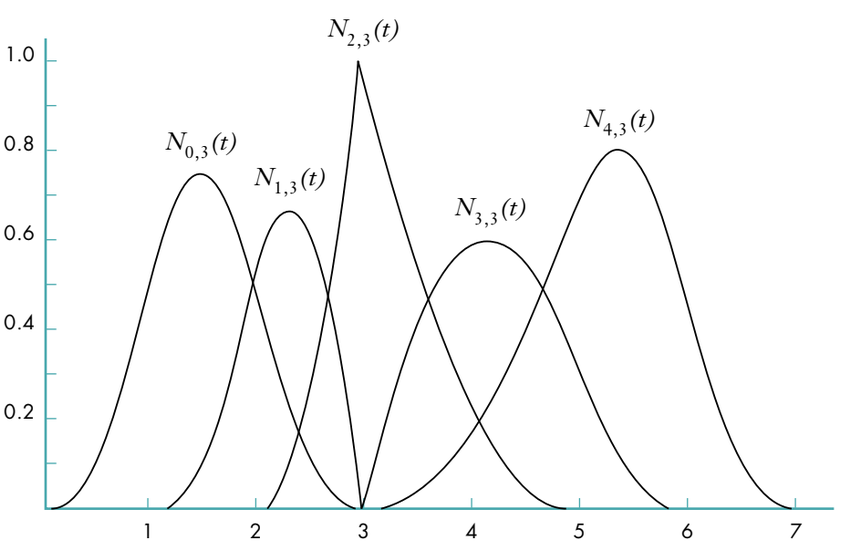


Figure 14. Basis functions for a curve with multiple identical knots in the middle

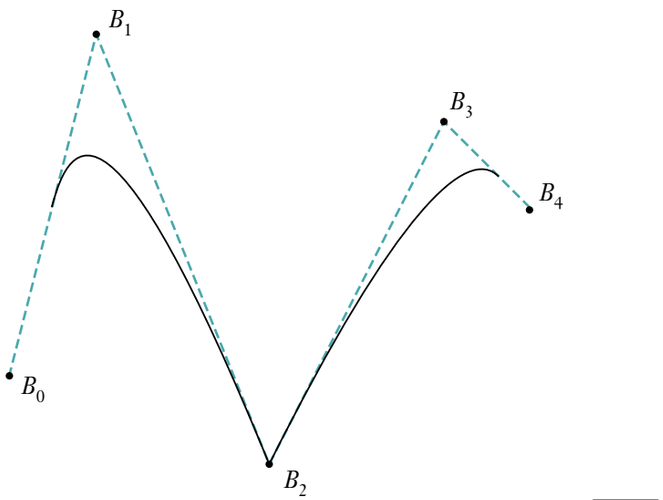


Figure 15. NURB curve with multiple identical knots in the middle

RATIONAL CURVES

Now that we've learned all about control points and knots and basis functions, we understand NUB (**n**on**u**niform **B**-spline) curves. But what about the rest of the acronym? We're still missing the *R* in *NURB*. It's time to talk about rational curves.

If you've sneaked a peek at QuickDraw 3D's NURB definitions, you may have wondered why it uses a four-dimensional representation for three-dimensional control points: {*x*, *y*, *z*, *w*} instead of just {*x*, *y*, *z*}. The reason for the extra coordinate is that it allows us to exactly represent conic curves (circles, ellipses, parabolas, and hyperbolas), as well as giving us more control over the shape of other curves. The fourth coordinate, *w*, is customarily referred to as the *weight* of the control point. Ordinarily, each control point carries a weight of 1.0, meaning that they all have equal influence on the shape of the curve. Increasing the weight of an individual control point gives it more influence and has the effect of "pulling" the curve toward that point (see Figure 16).

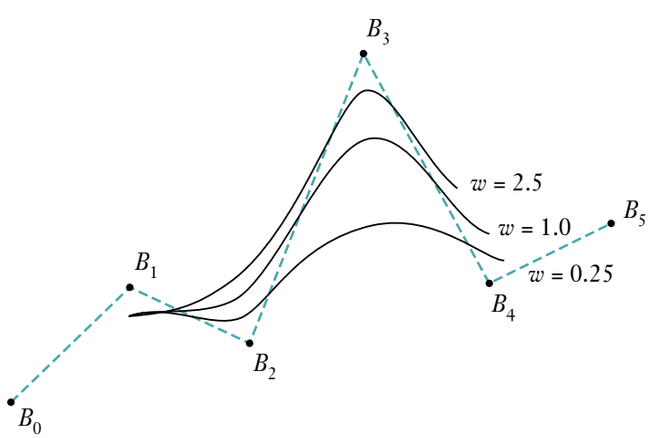


Figure 16. Increasing the weight of a control point

Curves that are defined in this way, with a weight w for each control point, are called *rational curves*. Mathematically, such curves are defined in four-dimensional space (since the control points have four components) and are projected down into three-dimensional space. Visualizing objects in four dimensions is a bit difficult (let alone drawing them in a diagram), but we can understand the basic idea by considering rational *two*-dimensional curves: that is, curves defined in three-dimensional space and projected onto a plane, as shown in Figure 17.

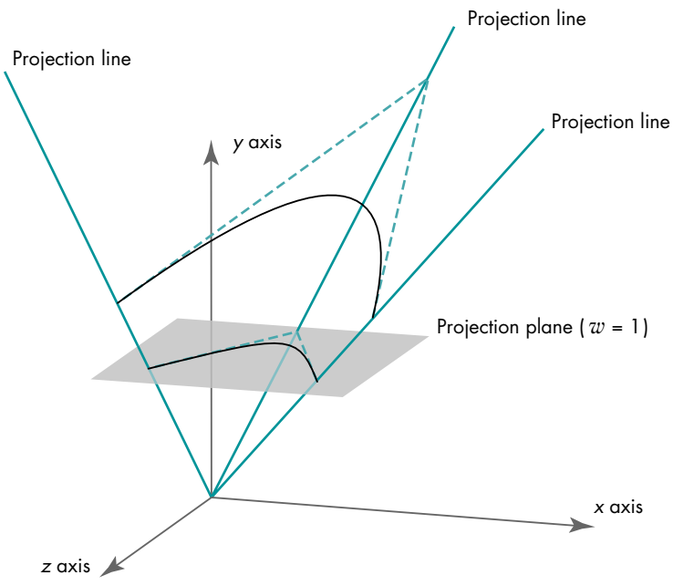


Figure 17. Projecting a three-dimensional curve into two dimensions

This is essentially the same process as projecting a three-dimensional model onto a two-dimensional screen with a perspective camera. The basic method for such perspective projection is to divide by the homogeneous component of the vertex (that is, w); we use an analogous approach to project our four-dimensional rational curve into three-dimensional space. Mathematically, then, we must incorporate this division into our earlier definition for a B-spline curve:

$$Q(t) = \frac{\sum_{i=0}^{n-1} B_i w_i N_{i,k}(t)}{\sum_{i=0}^{n-1} w_i N_{i,k}(t)}$$

The B_i are the projections of the four-dimensional control points and the w_i are their weights.

There are two different conventions for representing the control points in terms of their four-dimensional coordinates $\{x, y, z, w\}$:

- *Homogeneous*, in which the coordinates represent the point's position in four-dimensional space. To project it into three dimensions, the components must all be divided through by w. Thus the point's three-dimensional position is actually $\{x/w, y/w, z/w\}$. (Note that w/w is always 1.)

- *Weighted Euclidean*, in which the coordinates are already considered to have been divided through. Thus the first three components {*x*, *y*, *z*} directly represent the point's position in three-dimensional space and the fourth (*w*) represents its weight.

QuickDraw 3D uses homogeneous representation, as do most technical papers and other graphics libraries.

CONIC SECTIONS

I said earlier that we could use the rational aspect of NURB curves to create conic sections (such as circles and ellipses). Conic sections are so called because they're the curves we get by intersecting a cone with a plane; the angle at which the plane intersects the cone determines whether the resulting curve is a circle, an ellipse, a parabola, or a hyperbola. Strictly speaking, hyperbolas and parabolas are of infinite extent — but infinite curves are generally not useful in graphics applications (besides being very hard to compute a bounding box for). So we'll restrict our discussion to conic arcs.

Since conic curves are quadratic, we can represent them by quadratic (degree-2 or order-3) NURB curves. The practical question, of course, is which NURB curve. Although the proof is beyond the scope of this article, the following method (illustrated in Figure 18) can be used to generate conic arcs:

- The curve is defined by three control points. The first and last are the endpoints of the conic arc, while the placement of the inner control point helps determine the shape of the curve.
- The weights of the first and last control points are 1.0.
- A weight less than 1.0 for the inner control point generates an ellipse; a weight equal to 1.0 generates a parabola; a weight greater than 1.0 generates a hyperbola.
- The knot vector is {0.0, 0.0, 0.0, 1.0, 1.0, 1.0}.

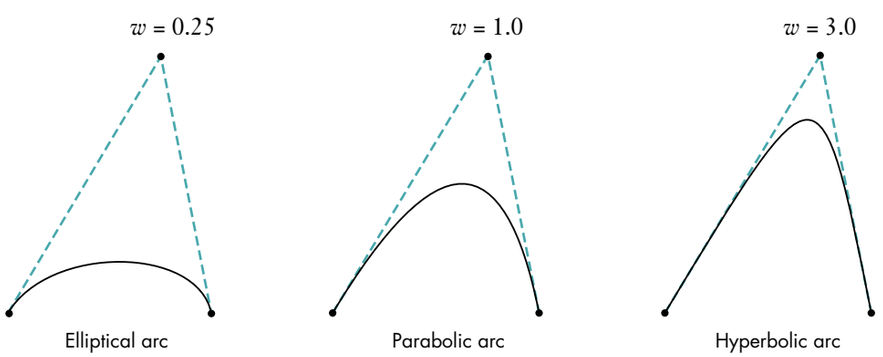


Figure 18. Constructing conic arcs

Probably the most common form of conic arc, particularly in modeling and design applications, is a circular arc. Since a circle is simply a special case of an ellipse, the method for constructing a circular arc is a special case of the general method for elliptical arcs:

- The legs of the control polygon are of equal length (that is, the control triangle is isosceles).

- The chord connecting the first and last control points meets each leg at an angle θ equal to half the angular extent of the desired arc (for instance, 30° for a 60° arc).
- The weight of the inner control point is equal to the cosine of θ.
- The knot vector is {0.0, 0.0, 0.0, 1.0, 1.0, 1.0}, just as before.

Figure 19 illustrates this construction. (In this case, the control triangle is equilateral, so the angle θ is 60° and the resulting arc is 120°, or one-third of a circle.)

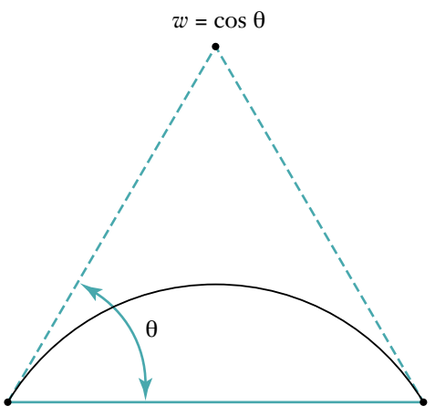


Figure 19. Constructing a circular arc

Note that the foregoing method can only produce circular arcs less than 180°; for larger arcs, we have to piece together several NURB curves. So to draw a complete circle we could combine three 120° arcs, or four 90° arcs. However, it's possible to represent these three or four separate arcs as a single curve and to make a circle with only one NURB curve. Figures 20 and 21 show how to do it with three and four arcs, respectively.

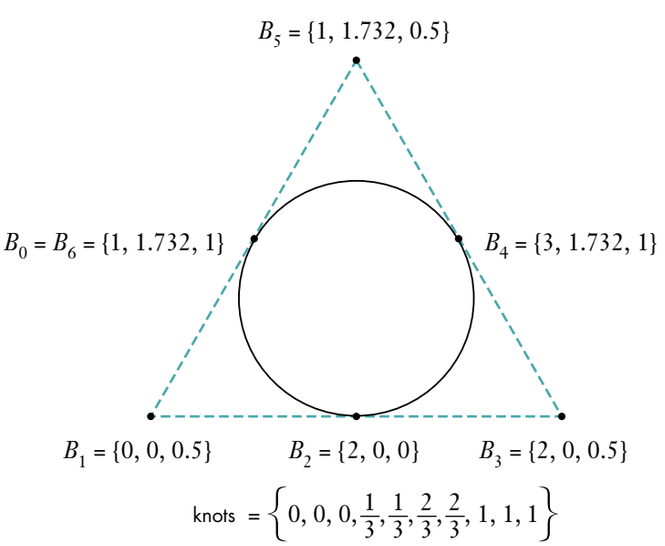


Figure 20. Constructing a circle with three arcs

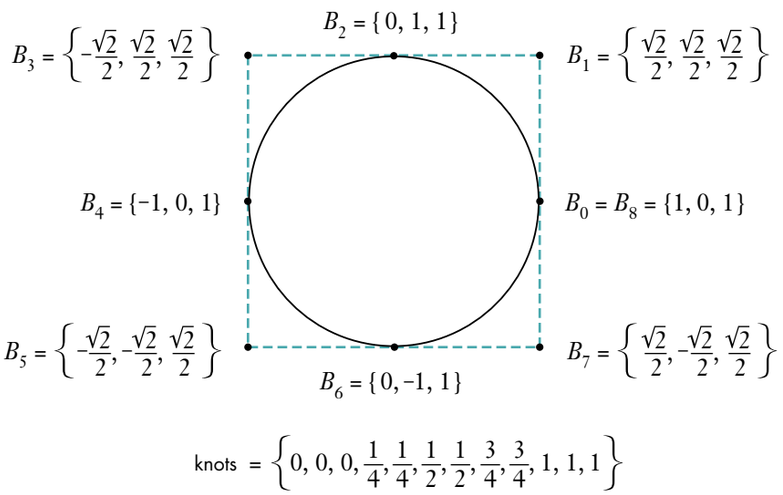


Figure 21. Constructing a circle with four arcs

NURB CURVES IN QUICKDRAW 3D

By now you're probably saying, "Enough theory already — how does all this relate to Macintosh programming?" So let's finally look at QuickDraw 3D's data structures and routines for working with NURB curves.

DATA STRUCTURES

If you've been following the discussion so far, you can probably guess the contents of the data structure representing a NURB curve: the order of the curve, its control points, and its knots. There's also the usual QuickDraw 3D attribute set, so you can draw your curves in, say, fuchsia or vermilion. Here are the definitions:

```
typedef struct TQ3RationalPoint4D {
   float    x;
   float    y;
   float    z;
   float    w;
} TQ3RationalPoint4D;

typedef struct TQ3NURBCurveData {
   unsigned long       order;              // Order of the curve
   unsigned long       numPoints;          // Number of control points
   TQ3RationalPoint4D  *controlPoints;     // Array of control points
   float               *knots;             // Array of knots
   TQ3AttributeSet     curveAttributeSet;  // QuickDraw 3D attributes
} TQ3NURBCurveData;
```

Most of this is pretty straightforward, but here are a few things to keep in mind:

- The order of the curve must be between 2 and 16, inclusive. Order 2 gives you a polyline effect; the most common orders are 3 (quadratic) and 4 (cubic).
- The control points are represented in homogeneous form, meaning that you have to divide the *x*, *y*, and *z* components by the *w* component to find the point's actual position in three-dimensional space.
- The *w* component of each control point must be positive.

- The number of control points must be equal to or greater than the order.
- The number of knots must be equal to the number of control points plus the order of the curve.
- The knots must be specified in nondecreasing order.
- If *k* is the order of the curve, there can't be more than *k*–1 knots with the same value (except at the beginning or end of the sequence, where *k* consecutive equal knots are allowed).
- The attribute set should contain only attributes that make sense for a curve. Most often, the attribute set will either be NULL or simply contain a color.

RENDERING A NURB CURVE

If you're familiar with QuickDraw 3D, you know that there are two ways to render a graphical entity (called a *geometry* in QuickDraw 3D terminology): *retained mode* and *immediate mode*. In retained mode, you first create an object representing the figure you want to draw, then use this *retained object* to do your drawing. (See the article "The Basics of QuickDraw 3D Geometries" in *develop* Issue 23 for more on this.) Listing 1 shows how this works for a NURB curve. First we initialize a TQ3NURBCurveData structure describing the curve to be drawn; we use this structure to create a retained

Listing 1. Rendering a NURB curve in retained mode

```
TQ3GeometryObject       curveObject;
TQ3NURBCurveData        curveData;

static TQ3RationalPoint4D    controlPoints[4] = {
                                { 0, 0, 0, 1 },
                                { 1, 1, 0, 1 },
                                { 2, 0, 0, 1 },
                                { 3, 1, 0, 1 }
                             };
static float                 knots[8] = {
                                0, 0, 0, 0, 1, 1, 1, 1
                             };

// Initialize the data structure.
curveData.order              = 4;
curveData.numPoints          = 4;
curveData.controlPoints      = controlPoints;
curveData.knots              = knots;
curveData.curveAttributeSet  = NULL;

// Make a retained object.
curveObject = Q3NURBCurve_New(&curveData);

// Use the retained object to render the curve.
Q3View_StartRendering(view);
do {
   Q3Geometry_Submit(curveObject, view);
} while (Q3View_EndRendering(view) == kQ3ViewStatusRetraverse);

// Dispose of the curve object.
Q3Object_Dispose(curveObject);
```

object with the QuickDraw 3D function Q3NURBCurve_New, and then we pass the resulting object to Q3Geometry_Submit to render the curve. Finally, we dispose of the retained object we created.

The equivalent drawing operation in immediate mode uses exactly the same code up to the point where the object is created. Instead of creating the retained object, we simply pass the TQ3NURBCurveData structure directly to the QuickDraw 3D function Q3NURBCurve_Submit to be rendered immediately:

```
// Render the curve directly.
Q3View_StartRendering(view);
do {
   Q3NURBCurve_Submit(&curveData, view);
} while (Q3View_EndRendering(view) == kQ3ViewStatusRetraverse);
```

CONTROLLING SUBDIVISION

QuickDraw 3D doesn't render NURB curves directly — as it does, say, lines or triangles. To draw a NURB curve, the renderer has to break it up into a sequence of lines or polylines. The more lines it's broken up into, the smoother it looks, but of course the longer it takes to render. Before rendering a curve, you have to tell the renderer how finely you want it subdivided.

There are three ways of doing this, denoted by the values of an enumerated data type:

```
typedef enum TQ3SubdivisionMethod {
   kQ3SubdivisionMethodConstant,
   kQ3SubdivisionMethodWorldSpace,
   kQ3SubdivisionMethodScreenSpace
} TQ3SubdivisionMethod;
```

- The first method, kQ3SubdivisionMethodConstant, says to subdivide the curve into a polyline with a specified number of segments between each pair of joints.
- The second method, kQ3SubdivisionMethodWorldSpace, says to subdivide the curve so that the length of each line segment is no longer than a specified value, measured in world space.
- The third method, kQ3SubdivisionMethodScreenSpace, is similar to the second, but the measurement is done in screen space.

The following data structure specifies the subdivision method to use and the relevant parameter values:

```
typedef struct TQ3SubdivisionStyleData {
   TQ3SubdivisionMethod   method;
   float                  c1;
   float                  c2;
} TQ3SubdivisionStyleData;
```

NURB curves use only the c1 component; the other is for NURB surfaces. A couple of things to note:

- You should set both c1 and c2 to legitimate values. QuickDraw 3D doesn't know whether a curve or a surface is coming up, so it always checks both parameters for validity. If you're only drawing a curve, you may as well set c2 to the same value as c1.

- If you specify an unreasonable value for either parameter, QuickDraw 3D will substitute a more reasonable one and issue a warning. It won't let you subdivide a curve at a million positions!
- For method kQ3SubdivisionMethodConstant, c1 should be a whole number greater than 0; fractional values will be truncated.
- If you don't specify a subdivision style, the default value will be used.

Expanding on our example of immediate mode rendering, the following code will render our NURB curve with a five-segment polyline between each pair of knots:

```
TQ3SubdivisionStyleData      subdivData;
...
subdivData.method = kQ3SubdivisionMethodConstant;
subdivData.c1 = subdivData.c2 = 5;
...
Q3View_StartRendering(view);
do {
   Q3SubdivisionStyle_Submit(&subdivData, view);
   Q3NURBCurve_Submit(&curveData, view);
} while (Q3View_EndRendering(view) == kQ3ViewStatusRetraverse);
```

EDITING NURB CURVES

If you're rendering your curve in immediate mode, you can edit the curve by simply modifying its control points, weights, and knot vectors directly in the TQ3NURBCurveData structure. If you're using retained mode, QuickDraw 3D provides calls to retrieve and set individual control points and knots:

```
TQ3Status Q3NURBCurve_GetControlPoint(TQ3GeometryObject curve,
            unsigned long pointIndex, TQ3RationalPoint4D *point4D);

TQ3Status Q3NURBCurve_SetControlPoint(TQ3GeometryObject curve,
            unsigned long pointIndex, const TQ3RationalPoint4D *point4D);

TQ3Status Q3NURBCurve_GetKnot(TQ3GeometryObject curve,
            unsigned long knotIndex, float *knotValue);

TQ3Status Q3NURBCurve_SetKnot(TQ3GeometryObject curve,
            unsigned long knotIndex, float knotValue);
```

Because we're not interacting with items that are objects themselves, there are no reference counts involved and no need to dispose of any data structures. Note, however, that if you edit a knot, the resulting knot vector must remain nondecreasing and follow the limitations described earlier for multiple knots.

You may have noticed that there are no calls to add, delete, or reorder control points or knots. Instead, QuickDraw 3D provides calls for retrieving and replacing the entire TQ3NURBCurveData structure from the retained object:

```
TQ3Status Q3NURBCurve_GetData(TQ3GeometryObject curve,
                              TQ3NURBCurveData *nurbCurveData);

TQ3Status Q3NURBCurve_SetData(TQ3GeometryObject curve,
                              const TQ3NURBCurveData *nurbCurveData);
```

If you want to change the number of control points and knots in a curve, you have to make a local copy of the data structure you obtain from Q3NURBCurve_GetData (making sure to allocate extra space for the new knots and control points), modify the arrays in the local copy, and store it back into the object with Q3NURBCurve_SetData. You must then call the following routine to dispose of the data you received from Q3NURBCurve_GetData:

```
TQ3Status Q3NURBCurve_EmptyData(TQ3NURBCurveData *nurbCurveData);
```

However, if you're going to be modifying the NURB curve frequently, you should probably be working in immediate mode and not using a retained object at all.

KNOT INSERTION

In general, the more control points we define for a NURB curve, the more control we have over its shape. It would seem reasonable that we could add more control points without changing the shape of the curve, and in fact this turns out to be true. Remember, though, that there's a fundamental relationship among the knots, the control points, and the order of the curve: the number of knots is equal to the number of control points plus the order. For example, a cubic curve (order 4) with 9 control points will require 13 knots. So every time we add a control point, we also have to add an extra knot — and make sure all the control points are in the correct locations to keep the curve's shape the same as before.

In practice, we actually take the reverse approach: we decide where to add a new knot, then compute the location of the corresponding new control point (as well as the new locations of some of the existing ones). For example, if we take the curve depicted earlier in Figure 9 and insert a new knot at $t = 3.6$, we get a new curve with exactly the same shape but with a new set of control points (Figure 22).

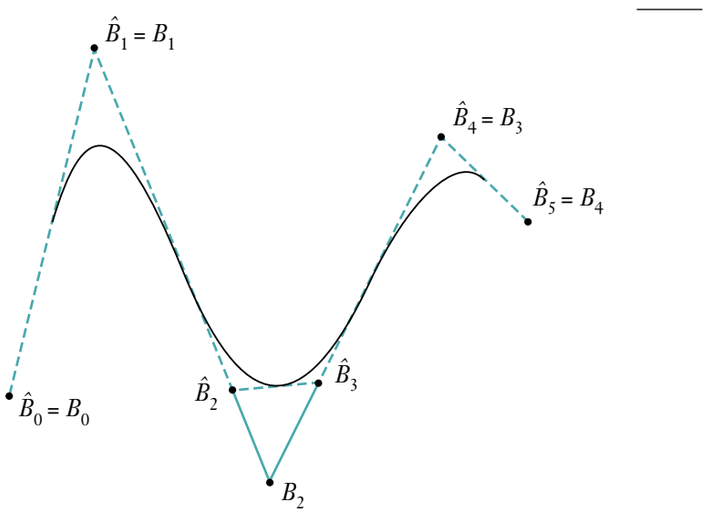


Figure 22. Inserting a knot

This operation of *knot insertion* is a fundamental one in working with NURB curves. It's directly useful in both modifying (editing) and rendering curves, and can also be used to convert a NURB curve to Bézier representation. After a brief discussion of the mathematical algorithm for inserting a knot, we'll look at some example C code for implementing it.

THE ALGORITHM

We start with a NURB curve represented by

$$Q(t) = \sum_{i=0}^{n-1} B_i N_{i,k}(t)$$

with a knot vector $\{x_0, x_1, \ldots, x_{n+k-1}\}$. Suppose we want to add a new knot x_{new}, where $x_i < x_{\text{new}} \le x_{i+1}$. The new knot vector $\hat{x}$ is simply the old knot vector with x_{new} inserted between x_i and x_{i+1}. The new curve will be defined by

$$\hat{Q}(t) = \sum_{i=0}^{n-1} \hat{B}_i N_{i,k}(t)$$

with knot vector $\hat{x}$.

Now we have to figure out not only where the new control point is located and where it goes in the ordered vector of control points, but also how to adjust some of the existing control points to keep the shape of the curve unchanged; this process yields the new control point vector, $\hat{B}$. It turns out that the relationship between the old and new control points is

$$\hat{B}_j = (1 - \alpha_j) B_{j-1} + \alpha_j B_j$$

where α is defined by

$$\alpha_j = \begin{cases} 1 & j \le i-k+1 \\ \dfrac{x_{\text{new}} - x_j}{x_{j+k-1} - x_j} & i-k+2 \le j \le i \\ 0 & j \ge i+1 \end{cases}$$

The proof of this is relatively simple, but we don't have the time or space to go into it here. For a full discussion, see *Curves and Surfaces in Computer Aided Geometric Design*.

THE IMPLEMENTATION

Listing 2 shows a function to implement this basic algorithm. The function, which is included on this issue's CD, accepts a QuickDraw 3D NURB-curve data structure as an argument, along with the value of the new knot to insert, and returns a new data structure representing the same curve with the new knot inserted. For brevity, the function performs no range checking on the inserted knot, but simply assumes that it falls within the legal range and that the resulting knot vector obeys the usual limitations on multiple knots. Note also that the code shown here does no checking on the results of memory allocation requests, though of course you should always perform such checks in real life.

EVALUATING NURB CURVES

Recall from our earlier discussion that if we have two knots at the same location, we lose one degree of continuity; with three identical knots, we lose two degrees of continuity; and so on. This process can be repeated until, when we reach k–1 identical knots (where k is the order of the curve), we have no continuity at all at the given point. In this case, the curve at that point coincides directly with a control point, as we saw in Figure 15.

Listing 2. Inserting a knot

```
static TQ3NURBCurveData *InsertKnot
   (TQ3NURBCurveData    *oldCurveData,    // Old curve
    float               tNew)             // Knot to insert

{
   TQ3NURBCurveData     *newCurveData;    // New curve after adding knot
   unsigned long        k;                // Order of curve
   unsigned long        n;                // Number of control points
   TQ3RationalPoint4D   *b;               // Old control point vector
   TQ3RationalPoint4D   *bHat;            // New control point vector
   float                *x;               // Old knot vector
   float                *xHat;            // New knot vector
   float                alpha;            // Interpolation ratio
   unsigned long        i;                // Knot to insert after
   unsigned long        j;                // Knot index for search
   TQ3Boolean           foundIndex;       // Insertion index found?

   // Set up local variables for readability.
   k = oldCurveData->order;
   n = oldCurveData->numPoints;
   x = oldCurveData->knots;
   b = oldCurveData->controlPoints;

   // Allocate space for new control points and knot vector.
   bHat = malloc((n + 1) * sizeof(TQ3RationalPoint4D));
   xHat = malloc((n + k + 1) * sizeof(float));

   // Allocate data structure for new curve.
   newCurveData = malloc(sizeof(TQ3NURBCurveData));
   newCurveData->order = k;
   newCurveData->numPoints = n + 1;
   newCurveData->controlPoints = bHat;
   newCurveData->knots = xHat;
   newCurveData->curveAttributeSet =
      (oldCurveData->curveAttributeSet == NULL)
                  ? NULL
                  : Q3Object_Duplicate(oldCurveData->curveAttributeSet);

   // Find where to insert the new knot.
   for (j = 0, foundIndex = kQ3False; j < n + k; j++) {
      if (tNew > x[j] && tNew <= x[j + 1]) {
         i = j;
         foundIndex = kQ3True;
         break;
      }
   }

   // Return if not found.
   if (!foundIndex) {
      return (NULL);
   }
```

(continued on next page)

Listing 2. Inserting a knot *(continued)*

```
   // Copy knots to new vector.
   for (j = 0; j < n + k + 1; j++) {
      if (j <= i) {
         xHat[j] = x[j];
      } else if (j == i + 1) {
         xHat[j] = tNew;
      } else {
         xHat[j] = x[j - 1];
      }
   }

   // Compute position of new control point and new positions of
   // existing ones.
   for (j = 0; j < n + 1; j++) {
      if (j <= i - k + 1) {
         alpha = 1;
      } else if (i - k + 2 <= j && j <= i) {
         if (x[j + k - 1] - x[j] == 0) {
            alpha = 0;
         } else {
            alpha = (tNew - x[j]) / (x[j + k - 1] - x[j]);
         }
      } else {
         alpha = 0;
      }

      if (alpha == 0) {
         bHat[j] = b[j - 1];
      } else if (alpha == 1) {
         bHat[j] = b[j];
      } else {
         bHat[j].x = (1 - alpha) * b[j - 1].x + alpha * b[j].x;
         bHat[j].y = (1 - alpha) * b[j - 1].y + alpha * b[j].y;
         bHat[j].z = (1 - alpha) * b[j - 1].z + alpha * b[j].z;
         bHat[j].w = (1 - alpha) * b[j - 1].w + alpha * b[j].w;
      }
   }

   return (newCurveData);
}
```

We've just seen that we can add a knot x_{new} and calculate the new control points. If we take this "new" curve (really just the old one with more knots) and add in that same knot again and again, until we have k–1 knots in the same place, we'll end up with a control point that lies exactly at $\hat{Q}(x_{new})$. We can use this technique to calculate the location of a particular point on the NURB curve: simply keep inserting knots at the point of interest until there are k–1 of them, at which time the newest control point created will lie at the desired point on the curve.

We can also use this approach to render a curve: by adding enough knots at some number of successive points in time t, we'll end up with a list of evaluated points on the curve, which we can then render as a polyline. The greater the number of

evaluation points, the more segments the polyline will have, and the more closely the resulting image will approximate the curve.

This isn't the most efficient algorithm; a number of better alternatives are available. For example, see *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling* for a description of the Oslo algorithm, which is significantly more efficient if you're adding more than a few knots.•

NURB CURVES AND BÉZIER CURVES

If you're familiar with Bézier curves, you may be wondering how they relate to NURB curves. In particular, if your application currently uses Bézier curves, how can you draw them when QuickDraw 3D currently only supports NURB curves? Although a thorough treatment of the subject is beyond the scope of this article, you'll be happy to learn that Bézier curves can actually be viewed as a subset of NURB curves. As a result, converting from Bézier to NURB representation turns out to be trivial.

CONVERTING BÉZIER TO NURB CURVES

Here's all it takes to convert a Bézier curve to NURB format:

1. Use the Bézier control points as the NURB control points. If the Bézier control points are rational (that is, if they have four components {*x*, *y*, *z*, *w*}), make sure they're in homogeneous rather than weighted Euclidean form. If they're nonrational (have no *w* component), simply set *w* = 1.0 for each NURB control point.
2. Set the order of the NURB curve to the number of control points. Bézier curves typically have three or four control points, corresponding to quadratic (order-3) or cubic (order-4) NURB curves, respectively.
3. Create a knot vector with 2*k* elements, where *k* is the order of the curve. Set the first *k* knots to 0.0 and the last *k* to 1.0.

Listing 3 shows a function to perform the conversion (it's included on this issue's CD). The Bézier curve is assumed to be represented by a data structure of the form

```
typedef struct BezierCurve {
   unsigned int   order;
   Point3D        *controlPoints;
} BezierCurve;
```

where the number of control points is equal to the order of the curve. The function returns a TQ3NURBCurveData structure representing the equivalent NURB curve. Once again, we've saved code space by leaving out the necessary checks on the results of memory allocation requests.

CONVERTING NURB TO BÉZIER CURVES

Converting a NURB curve to Bézier format is more complicated than the other way around. As we've just seen, any Bézier curve can be represented by a particular type of NURB curve, having half its knots at one end and half at the other. The converse, however, isn't true: an arbitrary NURB curve can't, in general, be represented by a single Bézier curve. In fact, it generally requires several Béziers to represent a single NURB curve: one for each distinct segment of the curve, as defined by its knot vector.

Recall that each segment of a NURB curve is affected by some subset of the control points. If we take each segment and add knots to both ends, generating a new set of control points each time, until each end has a number of knots equal to the order of

Listing 3. Converting a Bézier curve to NURB format

```
TQ3NURBCurveData *BezierToNURBCurve(BezierCurve *bezCurve)
{
   TQ3NURBCurveData  *nurbCurveData;   // NURB curve data structure
   unsigned long     k;                // Order of curve
   Point3D           *b;               // Bezier control point vector
   unsigned long     i;                // Control point or knot index

   // Set up local variables for readability.
   k = bezCurve->order;
   b = bezCurve->controlPoints;

   // Allocate data structure for new curve.
   nurbCurveData = malloc(sizeof(TQ3NURBCurveData));
   nurbCurveData->order = k;
   nurbCurveData->numPoints = k;
   nurbCurveData->controlPoints = malloc(k*sizeof(TQ3RationalPoint4D));
   nurbCurveData->knots = malloc(2*k*sizeof(float);

   // Create the control points.
   for (i = 0; i < k; i++) {
      TQ3RationalPoint4D_Set(&nurbCurveData->controlPoints[i],
                             b[i].x, b[i].y, b[i].z, 1.0);
   }

   // Create the knots.
   for (i = 0; i < k; i++) {
      nurbCurveData->knots[i] = 0.0;
      nurbCurveData->knots[i + k] = 1.0;
   }

   // Set attributes here, if desired.
   nurbCurveData->nurbCurveAttributes = NULL;

   return (nurbCurveData);
}
```

the curve, the result will be a Bézier representation of that particular segment. Do this for each segment, and we'll end up with a series of Bézier curves that, taken together, look exactly like the original NURB curve.

DESIGNING WITH NURB CURVES

The topic of how to use NURB curves in design could easily fill a book; we'll have to be content with just a brief discussion, along with some pointers for further reading.

The most obvious capabilities an application program can offer for creating and modifying NURB curves are

- interactive placement and movement of control points
- interactive placement and movement of knots
- interactive setting and modification of control-point weights

These capabilities can be moderately effective, but actually using them to model a desired shape turns out to be difficult and awkward. In addition, modifying a control point, knot, or weight will generally affect parts of the curve that the user wants to remain unchanged.

One problem that has been explored extensively is that of automatically creating a curve that goes through (interpolates) a given set of points, which may have been interactively placed by the user or perhaps obtained by some sort of data sampling. Indeed, it might be said that this was one of the original motivations for the mathematical development of spline curves. The first straightforward attempts yielded less than satisfactory results, but later efforts weren't too bad and may be useful if the curve must pass exactly through the given points. Often, however, we only need to *approximate* the given set of points with a spline curve. The points may have been obtained by sampling the user's freehand drawing with a mouse or tablet, or perhaps by measuring a physical object or extracting edge information from a glyph in a bitmapped font. In these cases, we probably want to preserve features such as endpoints and corners, but the remaining data samples may be noisy or nonsmooth and need not be fitted exactly. Techniques for both exact and approximate fitting can be found in *Phoenix: An Interactive Curve Design System Based on the Automatic Fitting of Hand-Sketched Curves* and *A User Interface Model and Tools for Geometric Design*. These techniques can also be adapted for use in modifying an existing curve, whether it was generated in the usual way or via one of these fitting algorithms.

CURVING ON

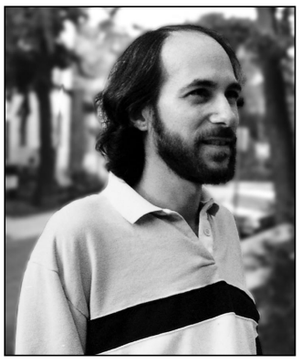
Well, there you have it: more than you probably wanted to know about NURB curves, plus some free code to boot. Look for a possible upcoming article on NURB surfaces, and how NURB curves and surfaces can be used together for designing objects and controlling motion.

BIBLIOGRAPHY AND RECOMMENDED READING

This article only scratches the surface of the theory underlying NURB curves. The following is a list of books and articles referred to in this article, as well as others you may want to investigate for further information.

- *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide* by Gerald Farin (Academic Press, 1990).
- *Curves and Surfaces in Computer Aided Geometric Design* by Fujio Yamaguchi (Springer-Verlag, 1988).
- *Fundamentals of Computer Aided Geometric Design* by Josef Hoschek and Dieter Lasser (A. K. Peters, 1993).
- *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling* by Richard H. Bartels, John C. Beatty, and Brian A. Barsky (Morgan Kaufman Publishers, 1987).
- *Mathematical Elements for Computer Graphics* by David F. Rogers and J. Alan Adams (McGraw-Hill, 1976).
- *NURB Curves and Surfaces from Projective Geometry to Practical Use* by Gerald Farin (A. K. Peters, 1995).
- *Phoenix: An Interactive Curve Design System Based on the Automatic Fitting of Hand-Sketched Curves* by Philip J. Schneider (master's thesis, University of Washington, 1988).
- "A Survey of Curve and Surface Methods in CAGD" by Wolfgang Böhm, Gerald Farin, and Jürgen Kahman, in *Computer Aided Geometric Design,* volume 1 (1984).
- *A User Interface Model and Tools for Geometric Design* by Michael J. Banks (master's thesis, University of Utah, 1989).

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



ACCORDING TO SCRIPT

Properties and Preferences

CAL SIMONE

On the way to implementing scripting support in your applications, you're bound to confront a variety of issues. In this column, I'll give you some pointers for devising and testing property names and discuss the techniques for handling preferences through scripting.

PROPERTIES

In an application's scripting vocabulary, a property is an attribute of an object. Properties can replace variables in **if** and **repeat** statements, as well as in expressions, and a script writer normally uses the AppleScript verbs **set** and **get** with them. Here I'll give some guidelines for coming up with human-language names for properties and testing the viability of those names within the overall natural style of the AppleScript language.

It's important that properties have names that users can easily become familiar with. Ideally, users should be able to refer to properties in a script the way they think or speak about them.

Don't start property names with verbs. Starting property names with verbs leads to confusion when the property appears in the middle of a sentence. For example, naming a property **disable call waiting** leads to commands that don't read smoothly:

```
set disable call waiting to true
if disable call waiting then ...
```

This is somewhat clearer:

```
set call waiting enabled to false
if not call waiting enabled ...
```

In fact, in the above case, it would be even better to name the property **call waiting** and use an enumeration as its value type (for a discussion of enumerations, see my article "Designing a Scripting Implementation" in *develop* Issue 21). The choices **enabled** and **disabled** allow grammatically correct sentences, as in the following:

```
set call waiting to enabled
if call waiting is disabled ...
```

A little creative thinking goes a long way in making it easy for users to work with the language.

The "the" test. AppleScript allows you to add or remove the word **the** almost anywhere in a script without changing the meaning of the script. Many script writers precede object and property names with the word **the** to make their scripts easier to read. Writing your test scripts in this way helps you determine the degree to which your property names facilitate forming natural sentences.

```
set the service to "America Online"
if the priority is high then ...
```

Don't confuse attributes and actions. Sometimes setting a property can cause an immediate change on the screen. In deciding whether to use a property in this situation, a helpful rule is: When an *action is initiated*, use a verb; when an *attribute changes* (even if it produces immediate visible results), use a property. Another way of looking at this is if a visible change is immediate, it's OK to use a property, but if an action has a duration, use a verb.

As an example, the following command causes an immediate change on the screen:

```
set the font of the third paragraph to "Courier"
```

Even though setting the **font** property creates a visible change, the font is still an attribute of the text, not an action. On the other hand, naming a property or enumerator **playing**, as shown in the next two commands, is a poor choice, because **playing** actually initiates an action:

```
set playing to true
set [the] status to playing
```

CAL SIMONE (AppleLink MAIN.EVENT, Internet mainevent@his.com) wants *your* dictionary for the Webster database, which will be used to help resolve human-name conflicts between different applications and scripting additions. He'll be analyzing the terms in your vocabulary against others in search of similarities and differences. Send your 'aete' resources to Cal via AppleLink or the Internet.•

The **playing** enumerator value in the second command is fine for obtaining state information, but a **status** property should be read-only. Instead of creating a property to control an action, use a verb. Verbs such as **play** or **start playing** are better suited for actions, as shown here:

```
play the movie "Wowie Zowie"
start playing the movie "Wowie Zowie"
```

Note that the commands are **play** and **start playing**, not **play movie** or **start playing movie**. In an application based on the object model, **movie** would be an object class.

The properties property. A **properties** property enables script writers to obtain all the properties for a given object in the form of a record by using a **get properties** construct. (I first suggested using records in this column in *develop* Issue 22.) The **properties** property can also be set with the **set** command. The sample **properties** property shown in Listing 1 can be included as a property of any object for which you allow the setting of more than one property at a time.

Listing 1. A sample **properties** property

```
{  /* array Properties: 5 elements */
   /* [5] */
   "properties",
   'Prop',
   'reco',
   "Property that allows setting of a list
      of properties.",
   reserved, singleItem, notEnumerated,
   readWrite, reserved,
   ...
},
```

Don't require the user to supply *all* the properties when setting the **properties** property — allow the setting of just one or a few properties.

```
get the properties of the fourth paragraph
   -- returns font, size, style, and so on
set the properties of the fourth paragraph to ¬
   {font:"Helvetica", size:14}
```

PREFERENCES

Developers use a variety of techniques to allow users to set preferences through scripts. I'll describe three common and easily implemented approaches for dealing with preference properties in your application class. (These same approaches can be used to implement document settings or group properties for individual objects within your application.)

Separate properties for each preference. Implementing preferences as individual properties works well when you have only a few preferences. For example:

```
set the connect sound to "Shriek"
set the receive folder to alias "HD:Drop Folder"
```

If you have many preferences, it's inefficient for the user to have to set each property individually. To solve this, you can implement your preferences as individual properties (usually in your vocabulary's application class definition) and also include a **preferences** property, described next.

A property that includes all the preferences. You can make a single **preferences** or **settings** property, which is a record that's defined elsewhere in your vocabulary. To define the elements of the record, create a fake "class" in your vocabulary, preferably in your Type Definitions Suite, to serve as the definition of the element labels in a record definition. In the comment field for your "class," be sure to document clearly that this is a record definition, not an object class. Listing 2 illustrates this technique; for more information, see the section "Define Record Labels in a Record Definition" in "Designing a Scripting Implementation" in *develop* Issue 21.

Lists and records are the two principal constructs in AppleScript that don't lend themselves to human sentence structure. They are, however, an integral part of the language and can occasionally help to make the script writer's life easier. When you use a record to create a **preferences** property, it's OK to stray a little from strict natural-language style. Of course, when referring to elements of a list or record, you should use natural-language style.

As with the **properties** property described earlier, don't require the user to set all the individual preferences at once. Allow the setting of just one or a few preferences at a time:

```
set the preferences to ¬
   {connect sound:"Shriek", ¬
    receive folder:alias "HD:Drop folder"}
```

A user can address individual preferences as if they were defined as separate application properties. To allow for varying user experience with AppleScript, your application should always accept property

specifications for individual preferences using the technique described above, regardless of whether the user includes the qualifying phrase **of the preferences**.

Listing 2. A sample **preferences** property

```
/* First, define this application property. */
{  /* array Properties: 5 elements */
   /* [5] */
   "preferences",
   'Pref',
   'cprf,    /* for "preferences class" */
   "Property that allows setting some or all
      of your preferences.",
   reserved, singleItem, notEnumerated,
   readWrite, reserved,
   ...   /* more reserved items */
},
... /* more property definitions */

/* Later, in your Type Definitions Suite, */
/* create a fake class. */
{  /* array Classes: 1 element */
/* [1] */
"preferences record",
'cprf',
"A record containing individual preferences",
{  /* array Properties: 10 elements */
   /* [1] */
   "connect sound", 'CSND', 'itxt',
   "the name of the sound to use when
      connected",
   reserved, singleItem, notEnumerated,
   ...
   /* [2] */
   "receive folder", 'RFLD', 'alis',
   "the folder to place files when received",
   reserved, singleItem, notEnumerated,
   ...
},
{  /* array Elements: 0 elements */
}
```

For example, both of the following statements should be allowed:

```
set the receive folder of the preferences to ¬
   alias "HD:Drop Folder"
set the receive folder to alias "HD:Drop Folder"
```

Multiple "group" properties for grouping preferences. If you have many preferences or want to group the preferences according to similar functionality, such as those often found in multipaneled dialog boxes, you can create separate properties for groups of preferences or settings (using the record definition technique just described). The properties can reflect the groupings you've set up in your graphical interface:

```
set the compiler preferences to ¬
   {warnings included:true, ¬
   default integer size:short integer}

set the drawing settings to ¬
   {pen size:{1,2}, shape:circle}
```

A user addresses an individual preference by including in the object property specification the record that the preference is an element of, as follows:

```
the pen size of the drawing settings
set the shape of the drawing settings to ¬
   rectangle
set the default integer size of the compiler ¬
   preferences to short integer
```

PARTING WORDS

Following these guidelines in implementing scriptability in your applications makes it easier for users to write scripts. Although they may seem like small points, it's the details that mean the difference between frustration and smooth sailing for the script writer. Remember to think about the way a user would write or speak about accomplishing what they want to do. Until next time, I remain your obedient servant on the AppleScript front. I'll see you on applescript-implementors@abs.apple.com, the mailing list for scriptability.

Thanks to Eric Gundrum, Jon Pugh, and Derrick Schneider for reviewing this column.•

Using C++ Exceptions in C

Exception handling in C++ offers many advantages over error handling in C. Using the techniques outlined here, you can implement C++ exceptions in your C code without a lot of effort. The payback is streamlined debugging that can result in more error-free code. When your program encounters errors, it jumps to the appropriate error-handling section, rather than dealing with the error locally. This simplifies your design and helps you concentrate on the normal flow of control. Centralized error handling also makes it easier to improve your reporting and feedback mechanisms incrementally.

AVI RAPPOPORT

I wrote a few little XCMDs in C and after the fifteenth crash of the day, I decided that I'd better add some error handling. So I looked at Dartmouth XCMDs, but I wasn't impressed. Each check for an error meant another indentation in the code, and I was worried about disposing of handles correctly as I passed errors up the call chain. Since I'd been looking at a lot of C++ lately, I wondered whether I couldn't use part of the C++ exception-handling mechanism to avoid problems in my code. It worked pretty well, so I thought I'd share my results.

For part of my solution, I used some Metrowerks macros. Metrowerks has graciously allowed these helpful exception and debugging source, header, and resource files to be included on this issue's CD, so you can use them without purchasing its CodeWarrior CD. The files contain macros that provide convenient tools for implementing exceptions and debugging signals, as well as an alert resource that can provide information during debugging.

Although I've used C++ exception handling in my C code with great results, I'd like to offer you one word of caution before you use them. Realize that C++ is not strictly an extension of C; as a result, in some cases it's possible that the program may not behave as you think it should.

BASIC ERROR-HANDLING REQUIREMENTS

All programs must respond to system and subroutine failures somehow. For example, many Macintosh Toolbox routines return a variable of type OSErr, while others

AVI RAPPOPORT has degrees in medieval studies and library/information studies, so she feels well qualified to work in the Macintosh software industry. In her job as user advocate and publications coordinator at Metrowerks, she spent her time documenting PowerPlant, making conference calls, and frantically trying to check CodeWarrior CDs before they were burned. Avi now works at StarNine as product manager for messaging products. She lives in Berkeley, California, with her Mac/Web scripter husband and their four-year-old son — all BMUG members.

require that you call Toolbox routines (such as MemError and ResError) to retrieve the error. If you ignore system and subroutine failures, your program is practically guaranteed to crash.

Good error handling allows you to cope with many kinds of problems. Your checks can trigger other code that deals with the problem (for example, by freeing memory). During debugging, error checking should notify you that something has gone wrong. And since you can't, unfortunately, catch all the bugs during testing, you must also set up an error-reporting mechanism to notify your users when something has gone wrong. In the worst case, your error handling should at least ensure that your program exits gracefully, without losing or corrupting user data.

THROWING EXCEPTIONS

The American National Standards Institute (ANSI) has defined a mechanism for C++ compilers that allows code to "throw" exceptions. When the compiler encounters a **throw** statement, it jumps to the nearest **catch** statement. (The "nearest" **catch** statement is the one associated with the current **try** statement, whether it's in the current routine or farther up the call chain.) The **catch** statement can deal with the error, pass it up the call chain, or both. A **throw** statement should appear only within a **try** or **catch** statement or in code called from within a **try** statement. Listing 1 shows these basic components.

Listing 1. Throwing exceptions

```
OSErr theErr = noErr;

// Try block.
try {
   // Do something.
   ...
   // If error, throw an exception.
   if (theErr != noErr)
      throw (theErr);
}
// Catch blocks.
catch (OSErr theErr) {
   // Do something with the error.
   ...
}
catch (...) {
   // Catch anything else.
   ...
}
```

As shown in Listing 1, exceptions are dealt with in *catch blocks*, which take an appropriate action depending on the error. For serious errors, this means cleaning up and terminating the program. For less serious errors, the catch block could continue without making a fuss, or make changes based on the error and again call the routine that threw the error; sometimes you might want to throw a more generic error, which is caught and interpreted in a higher-level catch block. I also recommend using the Metrowerks signal macros (described later) within your catch blocks to help you locate errors during debugging.

The three dots in catch (...) are actually in the code; the other such dots that appear in these listings are ellipses representing code that isn't shown.•

When carefully designed, C++ exception handling in your program can deal with problems at an appropriate level. As you may already have guessed, this feature is both powerful and dangerous. The advantage is that you don't have to mess around with returning errors for every routine or indenting deeply. However, if you allocate memory, you must be careful to dispose of it at the right time or it will cause a leak.

ADDING C++ EXCEPTIONS TO YOUR CODE

To add C++ exceptions to your code, you must do the following:

- Force the use of the C++ compiler.
- Create a top-level exception handler in your main routine.
- Define try blocks and catch blocks, and call **throw** at appropriate times.
- Add the C++ library (CPlusPlus.lib, CPlusPlusA4.lib, or MWCRuntime.Lib) to your project.

The Metrowerks macros that you'll see in the code that follows make implementing exception handling much easier than it would be otherwise. I'll talk about them later.

USING C++

To use C++ exceptions, you have to force the use of the C++ compiler. In Metrowerks CodeWarrior, the easiest way is to select the Activate C++ Compiler checkbox in the C/C++ Language panel. You should also make sure that the Enable C++ Exceptions checkbox is selected, because it enables throwing exceptions rather than direct destruction (one of those weird C++ things). An alternative way to invoke the compiler is to change the extension on your source code files to ".cp" or by changing the Target panel preferences; however, the checkbox method is the easiest.

C++ is stricter about automatic parameter conversion than C, so selecting the MPW Pointer Type Rules checkbox in the C/C++ Language panel avoids a bunch of errors (it forces the compiler to allow some implicit **char*** casts). But you'll get errors for other parameters and return values, so you have to clean them up as indicated by the compiler. For example, the following is an error message returned by a C++ compiler:

```
HC2RTF.c line 224  textLen = strlen(textString);
Error   : cannot convert
'unsigned char *' to
'char *'
```

To fix this problem, you can change the code to

```
textLen = strlen((char *) textString)
```

The CodeWarrior C++ compiler puts special C++ information into function names (this is called *name mangling*). C doesn't do this, so header files for C functions should be surrounded by **#extern "C"** statements to tell the compiler not to mangle these names (see Listing 2). The Macintosh Toolbox header files take care of this already.

CREATING A TOP-LEVEL EXCEPTION HANDLER IN MAIN

In your main loop or function, you should specify the top-level exception handler. This should catch serious errors, report them, and exit gracefully. Listing 3 shows the simplest possible exception handler (which you'll understand better as you read on).

Listing 2. Preventing name mangling

```
#ifdef __cplusplus
extern "C" {
#endif

long  FindBreak(char* buffer, short len);
// More declarations here
...

#ifdef __cplusplus
}
#endif
```

Listing 3. Simple top-level exception handler

```
pascal void main(XCmdPtr paramPtr)
{
   long  oldA4 = SetCurrentA4();

   try {
      CreateFile(paramPtr);
      WriteFile(paramPtr);
   }
   catch (...) {
      ReportError("\pSerious error occurred.")
      // XCMDs do not have to use ExitToShell.
   }
   SetA4(oldA4);
}
```

DEFINING TRY BLOCKS

When you use a **try** statement, it tells the compiler that the following code might have exceptions thrown in it. All functions that throw exceptions must be within a try block, either in the current function or in a calling function. It's pretty easy to set up try blocks before catch blocks. This is good, because you do have to do it: any throws that aren't caught will automatically abort the program.

DEFINING CATCH BLOCKS

You should have catch blocks for each error type. So, for example, you might define **catch (OSErr theErr)**, **catch (errStruct errRecord)**, and **catch (Str255 theErr)**. You should also have a generic catch, **catch (...)**, which doesn't have any parameters, to catch exceptions of all other types. Although it's better to use typed catches that handle specific errors, always add at least one generic catch and have it signal an error with an alert or break to the debugger. This will help you catch exception mistakes during your debugging and testing phase. Listing 4 shows examples of these types of catch blocks.

The compiler automatically routes the error to the appropriate **catch** statement, depending on the parameter passed to the **throw** statement. In Listing 4, both the StringPtr and OSErr types are caught specifically, after which they're reported. The OSErr catch rethrows the error as well. Any other types of errors are caught by the

generic catch, which calls a signal macro to display a message and then exits the program.

Listing 4. Specific and generic catch blocks

```
catch (StringPtr errString) {
   // If HandleError throws, it will be caught above this catch.
   HandleError(errString);
}
catch (OSErr theErr) {
   Str255 errString;
   ConvertErrToString(theErr, errString);
   ReportError(errString);
   throw (theErr);   // Rethrow to handle error.
}
// Forces the application to quit after the message.
catch (...) {
   SignalPStr_("\pUntyped error occurred in prefs.")
   ExitToShell();
}
```

You can, and often should, continue after catching an error. For example, after a disk full error, you should allow the user to choose a different volume. Note that the program will continue after the catch block, rather than in the location where the exception was thrown.

MOVING DEEPER — HANDLING EXCEPTIONS IN THE CALL CHAIN

Many of your low-level routines may call the Macintosh Toolbox or otherwise interact with the Mac OS. They should throw an exception if there's an error, as shown in Listing 5.

So where do you catch these exceptions? Remember, they percolate up the call chain until they find a **catch** statement, so you don't have to take care of them in the immediate calling function (unless you've allocated memory or done other things that need undoing). When you catch them, you can, and sometimes should, throw the error again. You can either report errors in mid-level routines or rethrow them up to a higher-level error reporting mechanism.

In addition to these **catch** statements, be sure to add a **catch** statement in circumstances where you need to do any of the following:

- Dispose of handles and otherwise deallocate memory.
- Shut down something you started in the try block, such as opening a file.
- Change the error thrown.

For your own functions, you should throw errors in situations that can cause serious problems or crash the machine. For instance, if you're providing a function that accesses a variable-length array that contains 16 members and the caller asks for the 17th member, you can throw a range error. There's no hard-and-fast rule about when to put the error checking into a function and when to require it before calling — it

Listing 5. Throwing exceptions for Macintosh Toolbox errors

```
void MakeMyResFile(Str32 fileName)
{
   CreateResFile(fileName);
   // Could also use the Metrowerks ThrowIfResError_ macro.
   err = ResError();
   if (err <> noErr)
      throw (err);
   // Continue with execution.
   ...
}

// Call the function.
MakeThisFile()
{
   ...
   try {
      MakeMyResFile(thisFile);
   }
   catch (OSErr theErr) {
      if (theErr == dupFNErr) {
         // Do something; file already exists.
         ...
      }
      else
         throw (theErr);   // Rethrow the error.
   }  // End catch statement.
   ...
}
```

depends on the situation. For example, if you're calling a function inside a tight graphics loop only and you want speed, you can probably check the parameters sufficiently in the calling function. However, if you have a utility routine that's called from several sections of your code, adding error checking will help you remember its requirements, such as parameters, memory, and other system states, to avoid problems later on.

Handling exceptions in libraries is tricky because you don't know much about the calling program. Think carefully about what you should report to the user and what you should simply return to calling functions.

As your programs become more sophisticated, you can start working around certain errors — for example, by using temporary memory when the application's heap is full. You'll also need to design interactive error reporting, allowing your users to take action (such as unlocking a locked disk) when they can. Then your application can continue properly.

EXCEPTIONS AND DEBUGGING WITH THE METROWERKS MACROS

The Metrowerks PowerPlant UDebugging and UException files, included on this issue's CD, provide convenient tools for throwing common exceptions and alerting you during debugging. To use them, put the folder in your project folder, add the

sources and the “PP DebugAlerts.rsrc” resource file to your project, and include the headers in your source files.

The UException.h file includes macros that automate common exception conditions. The UException.cp file includes an abort function. The UDebugging.h file defines some macros that make locating problems easier by allowing you to specify a *signal*, a debugging string displayed when the macro is invoked.

If your project includes an ANSI library you don’t need to add UException.cp. The abort function will conflict.•

SETTING GLOBAL VARIABLES FOR DEBUGGING

You need to set the global variables gDebugThrow and gDebugSignal in UDebugging.h to specify the debugging actions for throws and signals. By default, they’re set to do nothing at all. Other options include displaying a dialog, dropping into the source-level debugger, or dropping into the low-level debugger.

To activate the macros, be sure to define Debug_Signal in your precompiled header or UDebugging.h.

The following are the global variable options:

- debugAction_Nothing — Do nothing.
- debugAction_Alert — Display an alert box with an exception code (described later), filename, and line number where the throw or signal was made. For this to work, you must include the file “PP DebugAlerts.rsrc” in your project.
- debugAction_SourceDebugger — Break into the source-level debugger. For the Metrowerks source-level debugger, execution will stop with the arrow pointing to the line containing the **throw** statement. The exception code isn’t displayed. You can check the display of variable values in the source-level debugger for that information. (I’ve tested this with the Metrowerks debugger only.) If you aren’t running under the source-level debugger, debugAction_SourceDebugger will break into the low-level debugger on PowerPC processor–based machines, but might crash on 680x0 systems.
- debugAction_LowLevelDebugger — Break into MacsBug and display the exception code as a string. In MacsBug, the console will display two lines:

  ```
  User Break at routine + offset
  exception code
  ```

  Note that if you don’t have a low-level debugger installed, your program will crash with an unimplemented trap error if it tries to break into the low-level debugger.

THE THROW MACROS

UException.h defines several useful macros that automatically perform tests and throw exceptions if a test failed. It also defines a type, ExceptionCode (a long), and two standard exceptions, err_AssertFailed ('asrt') and err_NilPointer ('nilP'), which are treated as type ExceptionCode. Here are the throw macros:

- ThrowIf_(*test*) — Throws an exception if *test* is true, where *test* is a Boolean or the result of a Boolean condition. The exception code will be err_AssertFailed.
- ThrowIfNot_(*test*) — Throws an exception if *test* is false. The exception code will be err_AssertFailed.

- ThrowIfOSErr_(*err*) — Throws an exception if *err* isn't equal to noErr.
- ThrowOSErr_ (*err*), FailOSErr_ (*err*) — Throws an exception with *err* as the exception code.
- ThrowIfNULL_(*ptr*), ThrowIfNil_(*ptr*), FailNil_(*ptr*) — If *ptr* is NULL (or nil), throws an exception with err_NilPointer as the exception code.
- ThrowIfMemError_() — Calls the Toolbox routine MemError and throws an exception if it returns a result that's not equal to noErr; the MemError return becomes the exception code.
- ThrowIfMemFail_(*p*) — Throws an exception if *p* (a pointer or a handle) is nil. The MemError routine is used to check the success or failure of the last Memory Manager call. If MemError returns a result that's not equal to noErr, the exception code is set to the return value of the MemError call. If MemError returns noErr, the exception code is set to memFullErr, a constant defined by Apple.
- ThrowIfResError_() — Calls the Toolbox routine ResError and throws an exception if it returns a result that's not equal to noErr; the result becomes the exception code. ResError is used to check the success or failure of the last Resource Manager call.
- ThrowIfResFail_(*h*) — Throws an exception if *h* (a handle to a resource) is nil. If ResError returns a result that's not equal to noErr, the exception code is set to that result. If ResError returns noErr, the exception code is set to resNotFound, a constant defined by Apple.

You can use all of the macros within **if-else** clauses, as they're designed to be self-contained. For example:

```
if (err != fnfErr)
   ThrowIfOSErr_(err);
```

THE SIGNAL MACROS

UDebugging.h defines macros for raising signals, also known as *asserts*. These will stop the execution of the program and report errors. You can use them to check for nil pointers, out-of-range offsets, excess length, division by zero, and other problems. If you remove the definition of Debug_Signal, the entire set of macros is converted to white space and takes no runtime overhead whatsoever.

The macros are defined to check gDebugSignal for the action to take on execution, as described previously.

The following are the signal macros:

- SignalPStr_(*pstr*) takes a Pascal string argument. The string can be a literal Pascal string (in double quotes beginning with **\p**) or a StringPtr variable (and its variants, such as Str255).
- SignalCStr_(*cstr*) takes a literal C string argument. The string must be a literal (text within double quotes) and can't be a **char***. Because the underlying Toolbox routines take Pascal strings, the SignalPStr_ macro is more efficient.
- SignalIf_(*test*), SignalIfNot_(*test*) each take a Boolean condition as an argument and raise a signal depending on whether the condition is true or false.
- Assert_(*test*) is a synonym for SignalIfNot_(*test*).

STRESS REDUCTION WITH EXCEPTION HANDLING

C++ exceptions and these Metrowerks macros make error handling reasonably easy to add to most programs. With a little thought, you can design a clean structure for dealing with Mac OS errors and internal errors — a structure that's easily extensible to new code. You can avoid stress during testing by adding signal macro calls for common errors throughout your code. They're much easier to debug than system crashes. And yes, thank you, my XCMDs are much better now!

RELATED READING

- For a more in-depth examination of exceptions in C++, consult the article "Try C++ Exception Handling" by Kent Sandvik (*MacTech Magazine*, October 1995). For another view of C exceptions, see "Living in an Exceptional World" by Sean Parent in *develop* Issue 11.
- For information on the return values of Macintosh Toolbox routines and the error codes, see the *Inside Macintosh* series, *Macintosh Programmer's Toolbox Assistant*, and *THINK Reference*. You can also look at the header file Errors.h.

Because C has no objects, when you read these publications, you can ignore all discussions of object throwing, exception objects, construction, and destruction.

Thanks to Greg Dow, Pete Gontier, Tom Lippincott, and Jon Wätte for their C++ wizardry and personal patience, and to Pete and Tom for reviewing this article.

MPW TIPS AND TRICKS

Using ToolServer From CodeWarrior

TIM MARONEY

Last issue's column discussed various ways of using ToolServer. I looked forward to deeper integration of MPW scripts into other development systems, in a stirring plea that must have brought tears to the eyes of many an overly sensitive reader. In this column, I'll show just how self-fulfilling a prophecy can be: I'll explain how to use a generic ToolServer plug-in compiler with the popular Metrowerks CodeWarrior development system.

BEYOND THE WORKSHEET

CodeWarrior already comes with ToolServer support in the form of an integrated Worksheet window, similar to the MPW Shell Worksheet. Simply choose Start ToolServer from the Tools menu and you can issue all kinds of shell commands. It's like a miniature MPW Shell inside CodeWarrior. What more could you ask for?

Ten bonus points for reading skills if you could tell that that wasn't really a rhetorical question. The Worksheet is useful but it falls short of full integration. True, you can execute a Make command from CodeWarrior's Worksheet, but it would be even better to integrate MPW tools and scripts into the default CodeWarrior build sequence.

Let's say you have a SOM build — that is, you're using IBM's System Object Model as implemented on the Macintosh in the form of "SOMObjects for MacOS™". Before too long, it's likely that the current preprocessing approach to SOM will be just a fading (though still traumatic) memory, and that CodeWarrior and MPW will have direct-to-SOM compilers. For now, though, building with SOM requires running MPW tools and scripts to generate include files which are then processed by the C or C++ compiler.

Error prevention is one of the basic principles of user friendliness. You can invoke the SOM compiler from a makefile and run it in the CodeWarrior worksheet before you build, but if you're like me, short-term memory loss from a misspent youth will cause you to forget to run the makefile from time to time, leading to bizarre errors and gratuitous hair-tearing behaviors. MPW makefiles provide another rich source of errors by requiring you to track your own include files.

It's more convenient to simply give the single menu command Make than to bring up the Worksheet, enter "BuildProgram MyBuildFile", wait for the build to finish, and then give the Make menu command. One could only wish that CodeWarrior had a built-in SOM compiler.

A BUILT-IN SOM COMPILER (AND MORE)

Thanks to CodeWarrior's new plug-in compiler architecture (available starting with CW7), you can add build rules that invoke ToolServer scripts automatically to compile ".idl" files, or any other type of file. I've created a generic ToolServer plug-in for CodeWarrior (found on this issue's CD) that allows you to set up different command lines for different filename extensions. It will automatically track include files as well, if you want it to. It should be powerful enough for most applications, but if you need something different, you can take the source code and hack it endlessly to your own nefarious purposes.

To install the plug-in compiler, put the compiler file ToolFrontEnd into the Compilers folder of the CodeWarrior Plugins folder of your CodeWarrior application folder, and the preferences file ToolFrontEnd Panel into the Preferences folder of CodeWarrior Plugins. To set it up, first decide which filename extensions you want to run through ToolServer; in this example, we'll be doing the ".idl" files used by SOM. Give the Preferences menu command in CodeWarrior, go to the Targets panel, and attach the ToolFrontEnd compiler to source files of the appropriate type and extension.

Finally, go to the new ToolFrontEnd panel in Preferences and enter the command line you want to

TIM MARONEY depends on calcium for his structural integrity and potassium for the generation of axonic spikes in his nervous system. His recent reading includes *Mysticism and Philosophy* by W. T. Stace, *Popper Selections* edited by David Miller, *Seth, God of Confusion* by H. Te Velde, *Abrahadabra* by Rodney Orpheus, *Hathor and Thoth* by Dr. C. J. Bleeker, *Making Monsters* by Rochard Ofshe and Ethan Watters, and *Soul Music* by Terry Pratchett. A thoroughgoing nominalist, Tim doesn't believe in either tables or natural laws, but his contract work at Apple remains stubbornly limited by his desk and by the flow of time.•

execute for files of this extension. The ToolFrontEnd panel is shown in Figure 1. Like all software, this is a work in progress, so it may look slightly different by the time it reaches you.

The pop-up menu allows you to enter different commands for different filename extensions. Bind each extension to the ToolFrontEnd compiler from the Target panel.

The Script Include File will be executed before your command line. This lets you set up variables and aliases that may be useful for your scripts and tools. The same include file is used for all filename extensions. The include file can be anywhere in the project's access paths. Here I'm using an include file named "cwtsinclude," which sets up a few handy variables. You don't need to specify any include file if you don't want one.

Your source file can be preprocessed to find include files. I've provided a default preprocessor that deals with **#include** specifications. You can add other preprocessors — see the documentation and sample code that come with the software. Each include file will be added to CodeWarrior's internal list of dependencies for the source file, so the source file will automatically be rebuilt when an include file changes. If you don't want to scan for include files, choose None from the pop-up menu.

All include files should be in the CodeWarrior project's access paths. The project's access paths will be combined into the IncludeFiles variable, prefixed with the Path Parameter shown in the panel. This variable is available to all scripts executed from the plug-in.

All commands will be executed in the ToolServer context, so they'll use any startup scripts you've installed. See the notes from last issue's column about minimizing dependencies, though; all your requirements should be fulfilled by files you explicitly execute or by the Script Include File specified in the panel. Otherwise you'll run into configuration synchronization problems when restoring archived builds or sharing sources with your team members.

When the plug-in compiler executes scripts, ToolServer's current directory will have been set to the folder containing the project file. The following variables will be set up:

- {IncludeFiles} — the parameterized include path list
- {ProjectFolder} — the full pathname of the folder containing the current project file
- {SourceFile} — the full pathname of the source file being compiled
- {SourceFileStem} — the root or stem of the name of the source file (for instance, the stem of "MyPanels.idl" would be "MyPanels")

Generally speaking, you'll probably want to create a front-end script for the plug-in compiler, rather than enter a raw MPW command line in the panel. This allows you to specify any number of parameters,

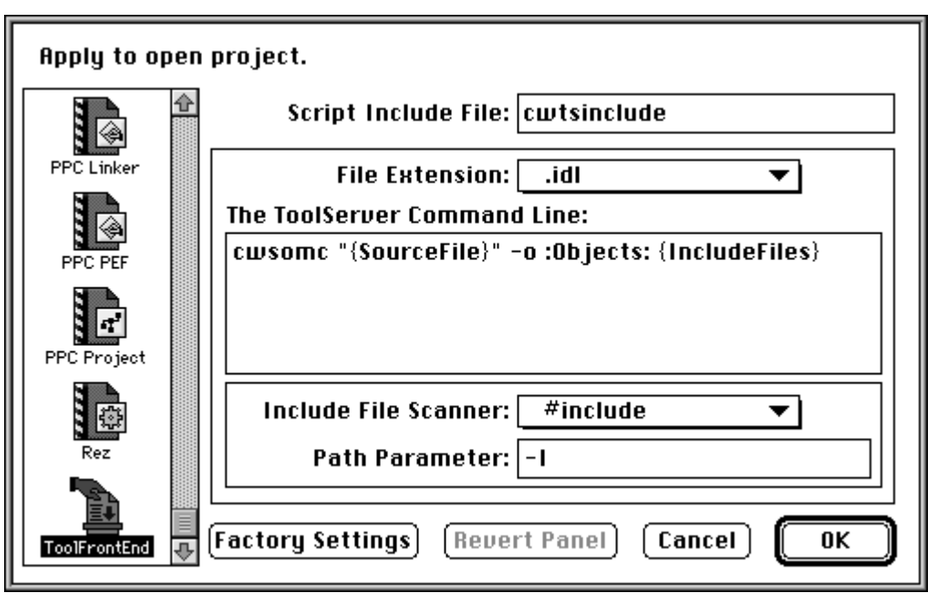


Figure 1. The ToolFrontEnd preferences panel

redirections, and so on in a script file without worrying about the text editing limits in the modal dialog. The command line you specify can be up to 255 characters long. The SOM compiler has a lot of options, so I've put all the ones I use into a script file named "cwsomc."

All diagnostic output will appear in the CodeWarrior error window. All standard output will be ignored. Internal errors in the plug-in will appear as alerts.

Due to a limitation in the current plug-in API, CodeWarrior doesn't know about dependencies involving compilers that put out source files. While the SOM compiler will emit, for instance, a ".xh" include file that will be included by a ".cp" file later in the build process, there is currently no way for CodeWarrior to know that the ".cp" file depends on the ".idl" file from which the ".xh" file was generated. This will be fixed in a future version of the API, and I'll add functionality to the ToolFrontEnd plug-in to support this feature when it becomes available. For now, since CodeWarrior compiles files in the order they appear in the project file, just put ".idl" files above ".cp" files.

UNDER THE HOOD

Source code is provided with ToolFrontEnd, so you can get a detailed peek at its insides and mutate it to your own needs. A quick overview may be useful here, though.

ToolFrontEnd sends commands to ToolServer in the form of Apple events, as described in last issue's column. It builds a command in memory that is a short multiline script with semicolons separating the commands. The last command of this script is the one you typed in the preferences panel. At the start of the script are commands that set the four variables described above. The diagnostic output is extracted from the 'diag' parameter of the reply Apple event returned from ToolServer, and the error code is extracted from the 'stat' parameter. All this is done using a slightly modified version of the sample code for communicating with SourceServer that I provided in this column in Issue 23; the Apple event conventions of SourceServer and ToolServer are much the same.

The plug-in was built by starting with the sample code provided with CodeWarrior. I didn't have to make any large-scale changes to the structure — Metrowerks deserves kudos for the quality of their sample code and their clean API. There are two code modules to be built, one for the preferences panel and one for the compiler itself. Library routines are provided for common operations like registering a dependency and getting a stored preferences record. The sample compiler already contained an include file parser, which I broke out into a separate module to allow customization for different file types.

THE BEST OF TWO WORLDS

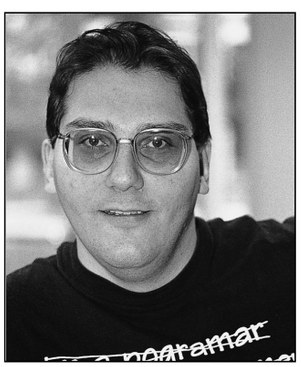
Though I've shown just one example, many different things can be done with the ToolFrontEnd plug-in. One of my friends is using a third-party version of the UNIX™ tool **yacc** (Yet Another Compiler-Compiler), which is delivered as an MPW tool. Thanks to this plug-in, he no longer has to switch between the MPW Shell and CodeWarrior constantly. You could also define your own macro language to preprocess your source files, or add your own original compilers. And of course, using ToolServer scripts from the compiler is fully compatible with using the CodeWarrior ToolServer Worksheet window for other tasks such as installing software. The Worksheet is also useful for testing and debugging scripts you'll use with ToolFrontEnd. I haven't provided a linker plug-in, but the API is similar and the compiler plug-in could easily be adapted to this purpose.

With ToolFrontEnd, the friendliness of CodeWarrior and the power of MPW celebrate a *hieros gamos* (or sacred union, if that's Greek to you). The fruit of this union is an all-in-one development environment from which you can execute your entire build process, no matter how complicated, without changing contexts or inviting errors. Those whose souls are devoid of romance may prefer to contemplate the consequences of increased productivity for their next performance review — in either case, enjoy in good humor and good health!

Thanks to Rick Mann and Greg Robbins for reviewing this column.•

Country Stringing: Localized Strings for the Newton

Newton products are currently available localized for English, French, German, and Swedish. Thus, to take full advantage of the market, Newton applications must be developed for four languages. As of Newton Toolkit version 1.5, there's a mechanism for localizing strings at compile time but no built-in support for organizing all the categories of strings across the different languages (unlike on the Macintosh, where you can use resources). This article presents a couple of ways to organize localized strings in your Newton application.



MAURICE SHARP

Until Newton Toolkit 1.5, developing an application for English, French, German, and Swedish required four different application projects or many skanky contortions. This was tedious, to say the least, but necessary for those who wanted to take full advantage of the worldwide market for Newton products.

Newton Toolkit 1.5 provides support (with the SetLocalizationFrame and LocObj calls) for localizing your applications from just one project. But this is useful only at compile time, and it doesn't provide an infrastructure for organizing and categorizing the localized objects. In other words, you can have different strings for four locales, but how you keep track of what strings you have and which ones need localizing is up to you. Macintosh developers don't have this problem because all strings can reside in resources; changing the strings in the resources changes them in the application.

This article presents two ways to organize your localized strings. Both methods are meant to be used at compile time, but there's also information on changing strings at run time. Before reading this article, you should be familiar with the information in the *Newton Programmer's Guide* on localizing Newton applications.

STRINGING YOU ALONG WITHOUT RESOURCES

In a Macintosh application you can keep localized strings in the 'STR#' resource of the resource fork. This isn't an option in a Newton application for two reasons: ResEdit doesn't directly support Unicode strings, and, more important, a Newton application doesn't have a resource fork. All your strings have to reside somewhere in your application package.

MAURICE SHARP is a truly multinational person. He was born in England, naturalized to Canada, and now lives in California. He hopes to visit the United States someday as well. His multinational background makes him a bit psychotic when it comes to beer. He's never sure if he should order it warm or cold, or just have water. This is why he prefers sake. Maurice is one of the original members of Newton Developer Technical Support and is still there (remember, we said he was a bit psychotic).•

A first cut at a solution to the problem of how to organize localized strings in your Newton application would be to have a viewSetupFormScript or TextSetup method (where applicable) that sets a particular string based on some application-global setting. This solution has several disadvantages, such as spreading localized strings throughout the code (resulting in multiple copies of strings) and requiring all strings for all countries to be included.

If you've programmed the Newton for a while, you might think of taking advantage of dead code stripping and using an **if** statement that switches on a compile-time constant. This would eliminate unused localized strings but is still awkward.

The best idea is a technique that lets you keep all your strings together. You can do this by defining a frame in your Project Data with one slot per string that you want to localize. You can even use nested frames. For example:

```
constant kUSStrings := '{
   AppName:    "World Ready!",
   ExtrasName: "World!",
   HelloWorld: "Hello World",
   Dialogs: {
      OK:      "OK",
      Cancel:  "Cancel",
      Yes:     "Yes",
      No:      "No",
   },
};
constant kFrenchStrings := ...
```

In Newton Toolkit 1.5 and later, you can use this frame with SetLocalizationFrame. Unfortunately, there's no specification for how to build up the frame, which is essential to organizing your strings in a sane way. Also, SetLocalizationFrame is meant only for compile-time localizations. With some extra effort you can organize the strings in a way that allows them to be localized at run time as well. As the next section shows, the key is using the Load command in combination with a few constant functions.

LINGUA FRAMA — CREATING THE LANGUAGES FRAME

In the previous section, we defined a frame that can be used for each target language. Each of those target language frames can be nested into an outer frame, called the *languages frame*. Each target language subframe contains the localized strings in that language. These subframes can in turn contain other subframes, enabling you to group strings into logical categories such as strings used in filing, strings used in searching, and so on. Each of the frames at the top level of the languages frame must have the same structure. If you have a path in the USEnglish frame of Entries.Names.Phones.Home, that path will also need to exist in French, German, and any other languages your application supports.

The overall structure of the languages frame is as follows:

```
{USEnglish: {
   AppName: "World Ready!",
   Dialogs: {
      Cancel:  "Cancel",
      OK:      "OK",
      // ... and so on
   },
```

```
 French: {
   AppName: "Prêt pour le Monde!",
   Dialogs: {
      OK:      "OK",
      Cancel:  "Annuler",
      // ... and so on
   },
 German: {
   AppName: "Welt Ready!",
   Dialogs: {
      OK:      "OK",
      Cancel:  "Absagen",
      // ... and so on
   },
 // ... and so on
}
```

This is the format of the frame you would pass to SetLocalizationFrame as well as of a constant that can be used in runtime localization. Typically, the languages frame would be kept in a text file or in your Project Data. The problem with this is that the frame is rather large, and adding or changing an entry in a language subframe can be difficult. Also, several entries are identical (such as the string for OK).

A better solution is to separate the localized strings by category. This article uses the target languages as the categories, though you could also employ similar techniques with other categories. Once the strings are split, you can use the Load command to assemble the languages frame.

There are two main schemes for organizing the strings. One uses simple text files and works on both the Mac OS and Windows platforms. The other uses compile-time functions to read the strings from some other format; on the Macintosh platform, this method can be used to construct the languages frame from a resource file. We'll look at each of these methods in turn.

LOADING FROM TEXT FILES

In the first scheme, you separate each language into a different text file. Remember that Load will return the result of the last statement it executes in the specified file. This means that each text file will specify one frame. For example, the contents of your French text file might look like this:

```
{
   AppName: "Prêt pour le Monde!",
   Dialogs: {
      OK:      "OK",
      Cancel:  "Annuler",
      // ... and so on
   }
};
```

You could then modify your Project Data to build the localization frame:

```
SetLocalizationFrame({French: Load(HOME & "FrenchStrings.f"), ...
```

It's also helpful to have some string constants that can be used in multiple places. A good example is the string for OK, which is the same in some languages. To do this, you should load some general constants before constructing the individual languages

that make up the languages frame. So the overall process for building the languages frame would be as follows:

1. Load a file of string constants.
2. Construct an empty languages frame.
3. For each language, build the individual target language frame and add it to the languages frame.

You only need predefined constants if you aren't using object combination. Object combination, a feature that exists as of Newton Toolkit version 1.6, would solve the problem of multiple instances of a single string (such as "OK").•

The above description smells of an algorithm. Since you can run NewtonScript at compile time, you can call a function to load a languages frame from text files (see Listing 1). The main trick of this function is that it uses the language symbol to create a pathname for Load.

Listing 1. CreateLanguagesFrameFromText

```
global CreateLanguagesFrameFromText(GlobalsFilePath, LanguagesSymArray)
begin
   if GlobalsFilePath then
      Load(GlobalsFilePath);

   local langFrame := {};

   foreach sym in LanguagesSymArray do
      langFrame.(sym) := Load(HOME & sym & "Strings.f");
   langFrame;
end;
```

You can define this function in a text file (say, WorldStrings.f) that you add to your project. Note that you must compile this file before you load your international strings.

You could use the languages frame directly as the argument to SetLocalizationFrame; however, as we'll see later in this article, there are better ways to use the frame.

LOADING FROM RESOURCES

The second scheme creates the languages frame from a resource file. You can apply the methodology to other non–text file sources as well. To take advantage of the code below, you'll need Newton Toolkit 1.6 or later. One important point: all of this code works only for Roman-based languages.

To make life easier, we'll define a template in ResEdit that shows all the localized versions of a particular string. The template defines a resource of type 'LOC#', which is loosely based on the 'STR#' resource (see Table 1). Because we're using a template, the number of languages must be defined in advance; we'll choose 5 as a nice arbitrary number. You can find the 'LOC#' template in the sample code on this issue's CD.

You can now use the 'LOC#' resource to enter all of your strings, grouped into categories that make sense to you. The advantage of this resource is that the path expression in the languages frame and all localized strings for that path expression are grouped together.

Table 1. The 'LOC#' template

Item Number	Label	Type
1	NumStrings	OCNT
2	*****	LSTC
3	path	CSTR
4	English	CSTR
5	French	CSTR
6	German	CSTR
7	Other1	CSTR
8	Other2	CSTR
9	*****	LSTE

You may be wondering why the 'LOC#' template contains an English string. If you use LocObj, the first argument is a string that's taken as the English localization. For the case where you're only localizing at compile time, the English string is redundant. But if you want to localize at run time, you'll need the English string around.

If you're familiar with the resource calls in the Newton Toolkit, you will have spotted a potential problem: there's no way to query for the available resource IDs of a particular resource. The basic solution to this problem is to try reading a resource and to catch the exception that the Newton Toolkit throws if the resource isn't present. Unfortunately, iterating through all possible resource IDs while catching exceptions takes several minutes.

So we impose these restrictions: there can be any number of 'LOC#' resources but they must be numbered consecutively, and the first resource ID must be either 0 (because programmatically generated resources are likely to start with 0) or 128 (because those created in ResEdit will start with 128). The code in Listing 2 generates an array of resources of a given type based on these criteria.

Once you have an array of 'LOC#' resources, you need to parse these resources into NewtonScript path expressions and strings. The code in Listing 3 gets all the 'LOC#' resources and generates a languages frame.

Unlike the text method, the resource method has to assume a certain number of base languages. The first thing the code does is to check that there are exactly five language symbols. If not, the code throws an exception. The result is a typical Newton Toolkit error dialog with the string specified in the code.

In reality, we could be a bit more forgiving. The code won't create entries in the languages array for items that are empty strings. So if a developer were careful not to fill out entries for particular languages, the restriction could be relaxed to *no more than* five languages. You could also make the code a bit more complex and just not add strings for undefined languages. This is left as an exercise for the masochistic reader.

An even better approach would be to create some other resource (say 'LOCi') that contains information on how many languages are defined by the 'LOC#' template and the language symbols. It would require slightly more complex code for CreateLanguagesFrameFromRsrc, but it would provide more flexibility later on. The CD contains modified code that uses an 'LOCi' resource.

As you can see, this is considerably more complex than the function used for text files. Also note that this methodology can't use constants for common strings. There are ways to massage the data to use constants, but that's left as another exercise for the reader.

Listing 2. GetAllResources

```
global GetAllResources(ResType, NewtType)
begin
   local result := [];
   local atID := 0;

   // See if we can read in resource ID 0. If so, increment the
   // next resource ID; if not, set the ID to 128.
   try
      AddArraySlot(result, GetResource(ResType, atID, NewtType));
      atID := 1;
   onexception |evt.ex.msg| do
      atID := 128;

   // Start at the current resource ID (either 1 or 128) and
   // continue reading in resources until an exception occurs.
   loop
   begin
      try
         AddArraySlot(result, GetResource(ResType, atID, NewtType));
         atID := atID + 1;
      onexception |evt.ex.msg| do
         break;
   end;
   result;
end;
```

Listing 3. CreateLanguagesFrameFromRsrc

```
global CreateLanguagesFrameFromRsrc(ResFilePath, LanguagesSymArray)
begin
   // Throw if there aren't exactly 5 languages.
   if Length(LanguagesSymArray) <> 5 then
      Throw('|evt.ex.msg|,
         "The LanguagesSymArray must be exactly 5 elements long.");

   // The languages frame array that will be returned
   local langFrame := {};
   foreach sym in LanguagesSymArray do
      langFrame.(sym) := {};

   // Could use a constant since currently must be exactly 5 languages.
   local numLanguages := Length(LanguagesSymArray);
   local r := OpenResFileX(ResFilePath);
   local locResourceArray := GetAllResources("LOC#", 'binaryObject);

   /* Process the LOC# resources. The format of the resource is:
      16-bit count of number of string sets
      string set 1
      string set 2...
      string set n
```

(continued on next page)

Listing 3. CreateLanguagesFrameFromRsrc *(continued)*

```
      string set:
         pathexpression as C string
         English as C string
         French as C string
         German as C string
         other1 as C string
         other2 as C string
   */

   local numStringSets;
   local pathExpr;
   local tempString;
   local atIndex;

   foreach locResource in locResourceArray do
   begin
      // Get the number of string sets.
      numStringSets := ExtractWord(locResource, 0);

      atIndex := 2;

      // Grab each string set.
      for stringSet := 1 to numStringSets do
      begin
         // Grab the C string that is the path.
         pathExpr := ExtractCString(locResource, atIndex);

         // Update index counter.
         atIndex := atIndex + StrLen(pathExpr) + 1;

         // Create path expression for following strings.
         pathExpr := call Compile("'" & pathExpr) with ();

         // Get the language strings and jam them.
         // WARNING: This code will ignore zero-length strings.
         // There are rare cases where you actually want an empty
         // string for a particular translation; in this case, you
         // could modify the code to throw an evt.ex.msg with the
         // appropriate error.
         foreach langSym in LanguagesSymArray do
         begin
            tempString := ExtractCString(locResource, atIndex);
            if StrLen(tempString) > 0 then
               langFrame.(langSym).(pathExpr) := tempString;
            atIndex := atIndex + Length(tempString) + 1;
         end;
      end;
   end;

   CloseResFileX(r);
   langFrame;
end;
```

PUTTING IT ALL TOGETHER

Once you've created the languages frame, you can use SetLocalizationFrame and LocObj in your project to localize your strings. The sample on this issue's CD (Compile Time Strings) uses the code shown in Listing 4. This code is more general than you may need, in that it creates the frame from either text files or resources. The last line sets up a constant for the English (that is, the default) language frame. You can use the constant English strings as part of the first argument to LocObj.

The LocObj mechanism can be used with any object, not just strings. This article looks only at strings, though the text-based method will work for most types of objects.•

Listing 4. Calling SetLocalizationFrame

```
// Create the languages frame, either by text or by resource.
constant kFromText := nil;

// Create the kLanguagesArray constant for the languages.
// The text method requires only as many languages as there are
// text files; the resource method requires a 5-element array.
DefConst('kLanguagesArray,
   call func(isText)
      if isText then
         '[English, French, German];
      else
         '[English, French, German, Other1, Other2]
      with (kFromText));

if kFromText then
   DefConst('kLangFrame,
      CreateLanguagesFrameFromText(
         HOME & "StringsCommon.f", kLanguagesArray));
else
   DefConst('kLangFrame,
      CreateLanguagesFrameFromRsrc(
         HOME & "strings.rsrc", kLanguagesArray));

SetLocalizationFrame(kLangFrame);

// Define a constant for the English language frame.
constant kStrings := kLangFrame.English;
```

You're probably wondering why we don't create a wrapper function to generate the correct LocObj call. Unfortunately, LocObj is a special type of call in the Newton Toolkit; it's evaluated as soon as the compiler hits it and it must return a constant value.

CHANGING STRINGS AT RUN TIME

The LocObj mechanism is designed for compile-time customization of your application. In other words, the LocObj function exists only in the compile-time environment of the Newton Toolkit; you can use it only in places that will be evaluated at compile time. In some circumstances you may want to change localized strings at run time. One example would be a language translator application where you want the interface strings to be displayed in the current source language.

The raw data for the runtime strings exists in the languages frame. The frame can be included in your package so that you have access to all the localized strings. This will add a significant amount of space to your package; at worst, it will take up two bytes per character in the unique strings, plus the storage occupied by the symbols and frame structure.

You'll need to add some runtime support for switching language elements of the interface. The main task is to decide what views need to be updated when a language is switched. The simplest way to do this is to recursively propagate a conditional message send through the application's view children:

```
// In application base view ...
myApp.PropagateLanguageChange := func()
begin
   // ... conditionally recur through all the kids.
   foreach child in :ChildViewFrames() do
      // "x.y exists" only checks for y using proto inheritance.
      if child.PropagateLanguageChange exists then
         child:PropagateLanguageChange();
end;
```

This code won't send to all children. To do that you would remove the **exists** test and just send the message, which will always be found since the top-level parent defines it. If you make this change, you should add some sort of conditional check for a message that does the real work of updating (like "if child.DoLanguageChange exists then ...").

An alternative is to keep track of which views need updates. How you do it depends on your application's structure. Typically, you would maintain an array of the declared views that need updating. If the views that need updating are well known, you're better off using the latter method.

Each view that requires an update will need to perform three tasks: change the text based on the source language; usually change the viewBounds based on the new text; and redraw or refresh based on the new viewBounds and text. Since it's very likely that the viewBounds will change, most of the work can be done in the viewSetupFormScript method of the view. Remember that redisplaying with a new viewBounds requires sending a SyncView, which has the side effect of sending all viewSetup messages.

This means that you can use the SyncView call as your message to indicate that the source language has changed. When a view opens by normal means it will also use the correct source language. Note that in some cases you may want to use RedoChildren, which has the same basic effect as SyncView sent to all children.

One caveat is that both SyncView and RedoChildren are expensive calls. You should limit the places where the language can change. An example of runtime customization (Run Time Strings) is provided on the CD.

READY TO ROCK AND ROLL

With the code from this article, you can now make all your applications world ready. If you're just starting an application, take the time and use LocObj where you should. If you already have a project, retrofit it. Then take the code samples, customize them to your heart's content, and code away. Today English, tomorrow the world.

Thanks to our technical reviewers Bob Ebert, Mike Engber, David Fedor, and Martin Gannholm.•

Newton Q & A: Ask the Llama

Q *Now that Newton 2.0 is shipping, what has changed?*

A A fair question, and one that's been much on my mind. Newton 2.0 solves some of the problems previously presented in this column in much better ways. So I've gone back over old questions to see what has changed. I'll start out this time by revisiting those questions that have new answers. (Questions that dealt with subsystems whose APIs on the Newton 2.0 OS are drastically different will not be covered; most of these have to do with routing, which has undergone a significant change for the better, while some have to do with communications.)

Q *How do I create my own class of binary object? (Issue 18)*

A In the Newton 1.x OS you had to use SetClass on a string object to make some other binary object. In 2.0 you can just call the new function MakeBinary. So the line of code to define the canonical CharID object (see the original answer) changes to

```
DefConst('kDefaultCharIDOBj, MakeBinary(4, 'CharID));
```

Q *I would like to add a* [*button*|*view*|*Llama*] *to* [*Notes*|*Dates*|*Names*|*etc.*]. *How can I do that safely? (Issue 19)*

A In the Newton 1.x OS there was no supported way to add items to the built-in applications. In 2.0 there are a few ways you can do this.

For general changes, you can add new stationery to Notes, Dates, and Names. For example, you could add graph paper to Notes. You could also define new card styles or views of a person in Names.

Names and Dates also let you add whole new classes of things. For instance, you could add a Pet type of names entry that would appear in the New pop-up menu along with Person, Company, and Group.

The Dates application has an API to add new types of meetings. It also lets you add items to its Info button.

In addition, there's a general API to register buttons that can show up in the "blessed" application's status bar. It's up to each application to decide whether and how it will display registered buttons. You should no longer use the unsupported keyboardChicken hack.

Note that on the Newton 2.0 platform there's still no support for adding buttons to built-in slips. For example, if you wanted to add something to the alarm picker for a meeting, you would need to add a new type of stationery that's a superset of the alarm picker.

Q *I've written my own IsASCIIAlpha, IsASCIINumeric, etc. functions. They seem to be really slow. Why is that? Here's my IsASCIIAlpha: [code not repeated here; all the functions work on strings] (Issue 20)*

The llama is the unofficial mascot of the Developer Technical Support group in Apple's Newton Systems Group. Send your Newton-related questions to NewtonMail or eWorld DRLLAMA or to AppleLink DR.LLAMA. The first time we use a question from you, we'll send you a T-shirt.•

A Most of the comments from the original answer still hold. However, in the Newton 2.0 OS the string could be a rich string; that is, there could be an ink character inside the string. That means the compare functions have to check whether a particular character was kInkChar.

Q *When I try to add an index to my soup I sometimes get an exception -48019, but not always. What's going on? (Issue 22)*

A In early versions of Newton, if you added an index on a slot, and an entry in that soup had a value of nil for that slot, you would get an error. As of the Newton 2.0 OS this is no longer a problem. You can add an index even if there are entries with nil values for the slot in the soup.

Q *I have an application that uses ADSP to connect to a server on the desktop. I want the server to handle multiple Newton devices connected simultaneously. Unfortunately, if a connection fails after it's opened, the server doesn't seem to be able to identify it as a new connection when the Newton device reconnects. This causes problems in the server's ability to handle multiple connections. Can you help? (Issue 23)*

A In the Newton 2.0 OS this no longer occurs. The Newton device will generate a new ID for the connection.

Q *Since there are changes between Newton 1.x and 2.0, what features in 2.0 can I rely on? What is the core set that defines Newton 2.0?*

A At this time there is no published core set of NewtonScript-level features that you can rely on. We're confident that you can rely on the features of the NewtonScript language and major components like the view system and communication endpoint interface. However, you can't count on individual protos or even the internal applications being there. Since we license Newton technology to other companies, they could produce a Newton device that doesn't include Names, Dates, or other built-in features. They may also produce Newton devices that have features that aren't present in Apple products.

The key is to test the features you rely on. If you find that some of the features you need are missing, you can either run in a less-featured mode or just not open your application. As a simple example, suppose your application runs only in a limited set of screen sizes and aspect ratios. You can give your base application view a viewSetupFormScript that looks something like this:

```
myBaseView.viewSetupFormScript := func()
begin
   local screenSize := GetAppParams();
   local aspectRatio := screenSize.appAreaWidth /
                        screenSize.appAreaHeight;

   // very simplistic test, no MINIMUM even!
   if aspectRatio > 1.0 then  // landscape
   begin
      local maxHeight := kMaxAppWidth;
      local maxWidth := kMaxAppHeight;
   end;
```

```
   else begin                 // portrait or square
      local maxHeight := kMaxAppHeight;
      local maxWidth := kMaxAppWidth;
   end;

   if screenSize.appAreaWidth <= maxWidth AND
      screenSize.appAreaHeight <= maxHeight then
   begin
      self.viewBounds := RelBounds(...);
      // other setup stuff
      ...
   end
   else begin
      // cannot operate at screen size
      :Notify(kNotifyAlert, EnsureInternal(kAppName),
         EnsureInternal(kErrorWrongScreenSize));
      AddDeferredSend(self, 'Close, nil);
   end;
end;
```

For global functions and variables, you can use the GlobalFnExists and GlobalVarExists utility functions. To find out whether a built-in application exists, you can check the root view with the appropriate symbol:

```
// check for Dates
if GetRoot().calendar then
   ...
// check for Names
if GetRoot().cardfile then
   ...
// check for Extras
if GetRoot().extrasDrawer then
   ...
```

For protos, you can try to access the proto and catch a frame reference exception. If the exception occurs, the proto is not present.

In general, it's a wise idea to do all your existence testing as your application is launching. Set flags in your base application so that you test for existing features only once.

Q *Is there a hardware-unique ID that I can access on a Newton device?*

A At this time there's no built-in hardware-unique ID, nor is there an API for accessing one if it existed. However, this doesn't rule out having such an API in future Newton devices.

Q *I'm using a Newton 2.0 protoSoupOverview and I want to change the font style. How do I do that?*

A This is one of those things that are obvious once you make the connection. You use the Abstract method of protoSoupOverview (and protoOverview, for that matter) to build the shape that's displayed for a particular soup entry. Notice that you're returning a shape, with all that entails. The chapter on drawing in the *Newton Programmer's Guide* says you can include a styles entry in a shape

array, allowing you to specify things like font style. See the DTS Sample Code Checkbook on the Newton Developer CD for an example.

Q *I noticed that some of the built-in applications have keyboards in their slips — for example, the new name editor in the Names file. Is this stationery based? Is there a magic slot I can set? Is there a proto?*

A Those keyboards are just views based on protoKeypad that are laid out as a child view of the slip. All you need to do is lay out your own protoKeypad and set up the definitions appropriately. There is no supported magic slot.

Q *I'm trying to use a protoListPicker to display a soup structure that has nested frame entries. I can't get the listPicker to work. Am I doing something wrong?*

A No. The default listPicker proto doesn't work with items that are accessed via path expressions. However, if you make the following three changes, your listPicker should work fine.

First, you have to specialize the GetObjSlot method of your pickerDef:

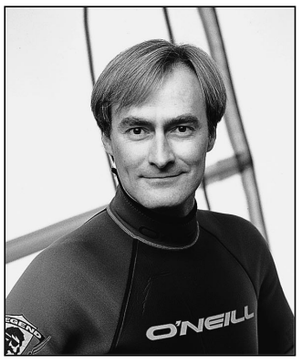
```
GetObjSlot: func(item, fieldPath)
begin
   if ClassOf(fieldPath) <> 'pathExpr then
      // if not a path expression, return the inherited value
      return inherited:GetObjSlot(item, fieldPath);

   // otherwise, if there is no item, return nil
   if not item then
      return nil;

   // there is an item, so get the real value, since the item
   // could be a NameRef or an Entry
   if IsNameRef(item) then
      local val := EntryFromObj(item);
   else
      val := item;

   // assuming we have a real thing, access the real data via the
   // path expression in fieldPath
   if val then
      val.(fieldPath);
end
```

Second, if you specify a validation frame in for your listPicker, the nesting of that frame must match the nesting of your soup entry.

Finally, modify your pickerDef so that the column that displays the data based on the index path uses the appropriate index path.

Thanks to our Newton Partners for the questions used in this column, and to jXopher Bell, Henry Cate, Bob Ebert, David Fedor, Jim Schram, Maurice Sharp, and Bruce Thompson for the answers.•

If you need more answers, check out http://dev.info.apple.com/newton on the World Wide Web or look at Newton Developer Info on AppleLink.•



THE VETERAN NEOPHYTE

Killing Time Killers

BO3B JOHNSON

So I'm sitting at my desk, mouse in hand, digging through the guts of my Mac, trying to track down yet another pathetic bug. The only trouble is, this is getting dull. I've done it a thousand times, and I always win; it's just a question of how much time the old ball and chain is going to eat up this time. Worse, the wind is blowing 20 knots right outside my window, and for the windsurfers in the crowd, you know the exquisite torture of good wind that you aren't allowed to transform into mind-blowing speed.

Maybe for you it's that you'd like to get home to see your insanely great mate. Or you've got kids whose names you can't pronounce. It's even likely that some of you just graduated from college and are still astonished that they pay you for something that's so much fun. But you're thinking, "If I can get this bug fixed quickly, I can get back to writing that rad Marathon hack to make the other net players slower than me." Perhaps your boss has started to notice that you spend a lot of time on the job but you don't really get very much done. Getting a little nervous? What if he's thinking of pulling the plug on your baby because you're too slow?

Or maybe you shipped the 1.0 version, but it had a few too many bugs, and *MacWEEK* was so incensed that they broke with the tradition of objective journalism and are calling for your head. People who bought your software with actual money have been calling every day, filling your answering machine with unveiled threats. It seems that you left a bug in there that just cost several thousand people each about two weeks of valuable time, reentering their data.

In particular, recognize that the wasted time from programming errors and bugs gets exponentially more expensive the further into the process you get. If I make a syntax error, I just fix it and recompile. If I toss buggy code over the fence to the testers, now I'm wasting *their* time. If I ship software with occasional crashing bugs that I just can't quite track down, I'm wasting thousands of people's time.

The point is, there are lots of ways to waste time while programming. I'm here today to offer some ideas on how to save time through better programming habits, so that you can take up windsurfing, or maybe the electric guitar, or learn how to pronounce your kids' names, or gain "the power to crush the other kids" while playing Marathon. Whenever I mention windsurfing, substitute your favorite quality-of-life enhancer.

I'll use some real life examples, and we'll see what sorts of lessons we can learn from them. I've categorized these ideas in three ways. First, there are some obvious time wasters that can be eradicated; these aren't really bug related, just daily time wasters. Considering how much time bugs cost, the second category consists of high-value rules that can find bugs quickly and painlessly. Finally, there are the super-value rules that prevent bugs from happening at all. Be sure to consider how these ideas might apply in your specific circumstances.

RIGOROUS, YET REUSABLE

OK, I'm in the midst of writing the MMU tables for Blat, and I realize that I've already got a similar table. Not being a fool, I know not to rewrite code I've already written, so I open that file and casually copy and paste the table into my current work. Oh no, not another copy and paste casualty! Apparently I missed changing those two table entries, and with MMU tables that means the machine hangs before MacsBug loads. Seems like every time I paste in code there's something I miss, making it not compile — or worse, causing a malfunction.

Even with small chunks of code, I've found it helps to review the pasted code line by line, carefully, and not to assume that since it ran before, it'll run now. Some subtle assumptions may have changed, and even though

BO3B JOHNSON (bo3b@rahul.net) is completely whacked out about windsurfing, and takes summers off in order to windsurf every day. But since it's winter, he's doing consulting so that he can pay for his next windsurf board and windsurf trip to Aruba. Bo3b prefers to be addressed as "Bob," since the 3 is silent.•

Where's Dave? That other Johnson, who usually writes this column, is probably at the public library researching his obsession du jour, taking his dogs for very long walks, or reclining on the couch reading a book. Since he's cut back his working hours, we're having guest Neophytes write this column. We can't promise they'll all be Johnsons, however.•

reusing code is certainly superior to writing it again, don't be misled into thinking this is risk free. A more powerful technique is to seriously modularize code, so that when I copy and paste I take an entire routine, not just a few lines. With a well-defined interface, the chances of blowing it are greatly reduced. This means adopting the habit of writing each routine with the idea that I'm going to reuse it later. This radically improves every routine I write.

New rule: Reuse code modules, not code fragments. If the code has to be altered, inspect it as if it were new (which it is).

VERBOSE, YET LUCID

While in the guts of Font/DA Mover, I ran across some very strange code that didn't make any sense to me at all — and it wasn't documented. It was never executed as far as I could determine, but I painstakingly figured out that it was looking for System file 3.2 and, if found, would patch the OS to fix a font bug. I would have saved a full day of effort had there been a comment in that funny little splat of code. Like I'm supposed to know what the bugs in System 3.2 are off the top of my head?

Comments really are necessary to make code reusable and maintainable. I always write "strategy" comments, which say what the routine is trying to do, and avoid writing "tactical" comments, like what it's doing line by line. Remember, sometimes the time savings occur in the future, not at the moment. I've found that skipping comments is being penny wise and pound foolish. Usually the strategy comments help clarify my thinking on the routine as well, so there actually is a short-term gain.

New rule: Always write strategy comments. It's possible to decipher intent from the code, but why not just explicitly say it?

PLUMP, YET HONED

I used to think it was important to save every line of assembly code that was possible. The first program I wrote for the Mac was Anaclock, an analog clock program, and I remember thinking that if I changed the order of some routines I could save code. Don't we all get into that mode sometimes? If I just change these two lines, I can save an assignment, and blah blah. It must come from the old 128K Macs and Apple IIs.

Well, guess what? These machines are so stuffed full of junk nowadays that saving just one or two lines is as meaningless an effort as trying to decide how many demons fit on the head of a transistor. Worse, I spent my own valuable time deciding something that has zero impact. Sorry, no can do anymore. My philosophy now is: write it straightforward, easy to read, vanilla. I want to save my windsurfing time, not pretend that I know up front what needs optimizing. In the Anaclock example, the computer had an entire second between screen updates. When I actually measured execution time, all the time was spent in CopyBits updating the screen, and waiting in the main event loop for the next second to arrive. There was zero measurable time in my entire clock calculation and offscreen drawing code.

New rule: No premature optimization. Measure with performance tools first. Then optimize only where it counts.

TEMPERAMENTAL, YET DISCRIMINATING

During System 7 development, we once tracked down a bug, taking seven hours in the middle of the night to find it, and it wound up being a bad parameter passed to a ROM routine. Incredibly, I could have found that bug in about 15 seconds if I'd used the Discipline tool. Nowadays, I never debug something by hand unless it has passed all the debugging tools that Fred Huxham and I talked about in our article in *develop* Issue 8.

There are lots and lots of tools available now, and I use all of them. I don't care how hard they are to use; if they can find a bug in seconds that might take me hours or days, then I win. This includes such notorious tools as Blat and Jasik's debugger. I know Blat's a pain, since it doesn't work on all machines, but hey, it's too valuable to skip. Same with Jasik's debugger. Sure it's confusing, but it's got features no one else provides. Before throwing the software over to the testers, I make sure it passes all the tools.

High-value rule: Use the best tools, all the time. Don't spend time in a debugger when a test tool will hand you the answer on a silver platter.

SPECULATIVE, YET REWARDING

As part of a contract, my job was to make a program to save, print, and display 300 dpi bitmaps that were scanned in from a fax machine through new hardware. This was to be a low-cost scanner, and my software would be the initial scan-and-display code. Nothing too fancy, but it still required basic functionality. I bid 15 hours for the entire program. Was I crazy? Well, of course, but not for this reason. I used MacApp to give me the application functionality, and the FracApp300 sample program was a good starting point for 300 dpi bitmap handling. All I really did was add an object to talk to the scanning hardware, and I came in under bid!

Sometimes learning those new tough coding tools can really pay off. I generally try to sample every new tool and coding advance that comes along to see if it can help me save time. MacApp was clearly a massive win, because it focused my programming onto teensy parts to be added instead of all the Toolbox calls of a typical application. In addition, it was fully debugged and very robust, giving me a more solid final application. I try not to be wedded to any given style or approach; I just want to use the best stuff currently available.

High-value rule: Try new things. New ideas, approaches, tools, and programming styles can be like winning the free-time lottery.

PAVLOVIAN, YET TRAINABLE

Sometimes it takes a while to recognize bad habits for what they are. While writing Bowser, which turned into Mouser and then MacBrowser, I wrote the source code parser by hand, to look for keywords. This was not a good strategy. It was quick and dirty, and stayed dirty, and was less quick all the time. It would be reasonable to expect that after modifying the parser for the eighth or ninth time to handle some stupid language exception, I would have gotten a clue that this was not the right approach. The right answer was to learn how the **lex** and **yacc** tools worked, since parsers for both Object Pascal and C++ already existed in that format.

After seeing similar bugs go by several times, it becomes clear that something must be done to stop that kind of bug. I don't want to spend time fixing the same problem over and over again, so now my goal is to permanently fix bugs so that they can't happen again. By this I mean changing how I do things, so that that specific bug will either be caught quickly or never happen again. It can be as simple as adding a test to a test suite to ensure that bugs of that form are caught immediately, or adding an assert to catch that error. Or it can be as hard as changing my programming habits to never use pointer math. Whatever it takes, I try to learn from each bug and make sure it can't happen again. Especially after I've done something twice, it's time to write a tool to fix that problem.

High-value rule: Learn from mistakes. If my dog gets bonked on the nose every time he gets near the door, he learns to avoid the door. I want to be at least as smart as my dog.

FASTIDIOUS, YET NOBLE

Another slant on the Bowser problem is that I wasn't really trying to make the parser right. If I'd been a little more quality conscious, I wouldn't have gone that route, because it was clear that the hand-built parser was clunky.

As noted before, the longer a bug survives, the more expensive it will be. Early bug extinction is my goal, so I consciously try to write with quality in mind. Examples are: using the strictest coding rules, not using any tricky features of the compiler, using type-suggestive variable names, insisting on type checking, not using raw pointer variables, avoiding type coercion, adopting a simple easy-to-read style, writing clear module interfaces, and using full warnings in the compiler.

Since I started noticing how much time bugs cost, I've changed my mindset on them. I no longer automatically accept that code will just have bugs. I hate 'em. I want to kill 'em. Better, I want to kill 'em before they hatch. Since they take up my personal time, I feel it's only proper to take it personally when they show up.

Super-value rule: Write with quality in mind. As they say, the inner game of programming is *so* important.

UGLY, YET EVOLVED

Once upon a time, I was asked to fix a couple of bugs in Font/DA Mover and make it work with TrueType fonts, as an interim solution before System 7. The program was so disgusting to me that I just had to go in and clean it up. Move this here, change these names, document some pieces, take out the redundant code, modularize some pieces — ah, how aesthetically pleasing. Oops . . . I just introduced a couple of bugs while I was "improving" the code. It felt like progress, but actually it was just motion. You know, like company reorgs.

What to do? Don't "improve" code, unless it's never been debugged. Any fully debugged code, no matter how shoddily written, is superior to newly written code, no matter how pristine. It went against my grain, but the right answer was to leave it gross. That heavily used Font/DA Mover code had thousands of hours of value in it, with literally millions of testers, that were all lost when I rewrote it. Rewriting it took time that I wanted to spend on something more valuable, like fixing the last few remaining bugs — and then getting outside and windsurfing! Once I rewrote the code, it was like a new program, and thus needed a full development/testing/debugging cycle. I backed off to an earlier "skanky" version and just debugged that.

Super-value rule: Never rewrite something that's been fully tested. It may be ugly, but evolution is on its side.

BORING, YET ELEGANT

We all know about the "cool" things that C can do, and some tricky ways of using it, but sometimes isn't it a bit like juggling live weasels? When I found that using a #define had added an extra unwanted character to each place I used it, it no longer seemed so clever, and felt more like I was playing tricks on myself. Or how about that favorite of putting an actual assignment in an **if** statement? It's cleverly camouflaged, but there aren't any natural predators here, so I'm not sure this is needed. These simple examples obviously don't do justice to the possible tricks that we've all seen, but they all cost time and rarely add value.

OK, so it's clear that being "clever" often winds up being a way to play tricks on myself. Is there anything wrong with doing it simply, in a straightforward, vanilla style? I know for sure I'll get it done sooner and, even better, the programmer who has to maintain this code won't have to waste a bunch of time understanding mindless tricks (remembering of course that that maintenance programmer might very well be me, two years after I forgot what tricks I was playing). And let's just forget the malarkey about it saving code. Is it really worth saving 10 whole bytes out of a 16 meg machine, at the expense of wasting my time? I want to count cycles and bytes only in places where it makes a measurable difference.

Super-value rule: Write vanilla code. Doing it simply, and the same way each time, also makes it more likely to be correct.

ASSERTIVE, YET FRIENDLY

Back in the deep dark Macintosh past, I wrote the driver for an external RAM disk called DASCH. This high-speed serial link required some different debugging tactics than I'd used previously, because I couldn't step through the code; it was time critical. Any slight perturbation in speed would overrun and cause an error, but I still needed to debug it. It was like a "look Mom, no hands" type of debugging. Code inspection is OK, but I wanted to be sure it worked as I read it. Have you ever read a piece of code that took a branch you didn't expect?

The answer, although I didn't use the name at the time, was to use asserts. These have been talked about a fair amount, and you've probably used primitive asserts under the name of DebugStr. Nowadays, the most powerful combination I've used is to hook together asserts with a failure handler like MacApp's catch/fail mechanism. Asserts make it easy to build a debug-only version that checks every stupid thing that can go wrong and lets me know right up front during testing, but doesn't compile into the final version. The catch/fail stuff makes it easy to handle every possible error in a graceful way. (See the article "Using C++ Exceptions in C" in this issue.)

If something absolutely positively cannot fail, I use a debugging-version assert to catch the occasional times when it does fail, so that I can surprise myself early and not spend hours tracking down the "impossible" error. One great thing to check with asserts is input parameters, to catch those inevitable times when some routine passes in rubbish.

Super-value rule: Use asserts along with a failure handler. Catching bugs as they happen is vastly superior to backtracking 15 miles after the program crashes.

ENDING, YET BEGINNING

I'm not going to pretend that this is all there is to the idea of saving time, but hopefully the idea seems worth pursuing. It has certainly helped me get better at my carving jibes and, not incidentally, better at programming at the same time. Higher-quality code, fewer bugs, earlier ship dates, happier customers, and more free time. Yup, I'd say it's been worth it.

If you've got some additional time-saving ideas, I'd naturally be interested in trying them too, so write me at bo3b@rahul.net.

RECOMMENDED READING

- *Writing Solid Code* by Steve Maguire (Microsoft Press, 1993).
- *Debugging the Development Process* by Steve Maguire (Microsoft Press, 1995).
- "Macintosh Debugging: A Weird Journey Into the Belly of the Beast" by Bo3b Johnson and Fred Huxham, *develop* Issue 8, and "Macintosh Debugging: The Belly of the Beast Revisited" by Fred Huxham and Greg Marriott, *develop* Issue 13.
- *Zen and the Art of Windsurfing* by Frank Fox (Amberco Press, 1988).

Thanks to Jeff Barbose, Jim Friedlander, Brian Hamlin, Fred Huxham, Dave Johnson, Jim Reekes, and Patty Walters for their terribly helpful review comments.•

Missing something?

Are there issues of *develop* that have passed you by? If you'd like to complete your *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 AppleLink and 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 Writing Localizable Applications; 3-D Rotation Using a 2-D Input Device; Video Digitizing Under QuickTime; Making Better QuickTime Movies

Issue 15 QuickDraw GX (Getting Started; Printing Extensions; PostScript); 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 Proto Templates on the Newton; Standalone Code on PowerPC; Debugging on PowerPC; 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; Dylan; Designing a Scripting Implementation; Object-Oriented Hierarchical Lists; Introducing PowerPC Assembly Language

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 Images Faster With Data Compression; The New Device Drivers and Memory

Macintosh Q & A

Q *I'm having problems with our caching drivers on the Power Macintosh 9500. Our drivers allocate a large amount of RAM (up to 4 MB) early in the boot process. If I set the driver's cache size to 4 MB, the computer locks up as soon as the driver is executed. If I set the cache size to 2 MB, the driver loads and executes properly, but the computer gets a bus error much later in the boot process (after MacsBug loads and after the Mac OS screen is displayed, but before the Finder executes). If I set the cache size to 1 MB, everything runs properly. What's going on here?*

A Because of Open Firmware requirements, the boot stack on the new Power Macintosh 9500 CPUs was changed to 4 MB. As a result, you can't grow the system heap past 4 MB or a system crash will occur. If possible, try to defer allocating memory until INIT time. The 'sysz' mechanism is supported by the enabler ('boot' 3) when INITs are being loaded.

Q *We want to maximize our throughput across the PCI bus between Macintosh memory and a block of static RAM on our card. This static RAM is also accessible from an on-card DSP, which constantly reads/modifies RAM. The DSP isn't directly on the PCI bus, so it can't easily participate in cache coherency schemes. What's the best way to get data across the PCI bus to and from this memory?*

A The PowerPC processor can only burst to and from a *cacheable* memory space. Your best option is to use BlockMoveDataUncached. This doesn't use burst transfers, but rather uses floating-point loads and stores. You may want to design your own algorithms, using the **double** declaration in C to get compilers to translate BlockMoveDataUncached into floating-point loads and stores. For more information, see Chapter 9 of *Designing PCI Cards and Drivers for Power Macintosh Computers*, and Technote 1008, "Understanding PCI Bus Performance."

Q *The Power Macintosh 9500 Developer Note says that the sound-in port for the 9500 has 4-conductor requirements. What is the 4-conductor pinout? Does the PlainTalk microphone work on the 9500?*

A The 9500 has stereo in (supporting left, right, power, and ground) and requires a mini stereo plug that you can buy today — no wacky new pinouts. The PlainTalk microphone works fine on the 9500.

Q *Is there any information on writing native SCSI disk drivers for the PCI-based Power Macintosh computers? In particular, what's the proper way to install a native driver on a SCSI disk: is there a special partition type for a native driver, or should there be a standard SCSI disk driver that loads a PowerPC code fragment?*

A Apple doesn't support native SCSI drivers yet (this will be a feature of Copland, the next generation of the Mac OS). You *can* write a native SCSI Interface Module (SIM). Remember that a driver is the software that handles a particular SCSI device, while a SIM is responsible for SCSI controllers (for example, PCI or NuBus™ cards).

Normally, SCSI 4.3 drivers are loaded off the Apple_Driver43 partition, and SIMs are typically loaded from the disk controller firmware (PCI card). If you want to load a native SIM off of the disk, you'll have to encapsulate the code fragments and read and link them in from your standard 680x0 driver.

Q *We have SCSI routines that transfer 64K data blocks (to get the highest transfer rates possible with the tape drive we're using). On the Power Macintosh (8100/80 or 8100/100), if the mouse is moved during a 64K transfer, the cursor is jumpy. Lowering the cache size to 16K reduces the problem to an acceptable level but kills the transfer rates. We're using SCSI Manager 4.3. How do we avoid the jumpy cursor while maintaining maximum throughput?*

A Your jumpy cursor is an indication that you aren't properly implementing SCSI direct memory access (DMA). When using the 8100 (and PCI-based machines) for DMA transfer efficiency, you should ensure the following:

- Your data is aligned on 8-byte block boundaries. Since the DMA hardware can't do odd transfers, it must perform programmed I/O to handle at least part of the transfer if the data isn't aligned.
- Your physical memory buffer is contiguous (which you ensure by calling LockMemoryContiguous). Otherwise, the DMA transfer will have to be broken up; this will especially be a problem if virtual memory is turned on.

Also, if you have disconnects enabled in your device or driver, it's possible that the transfer is getting broken up and some VBL activity is occurring. The bottom line is that you don't want a SCSI disconnect to occur during your transfer.

Q *Is it possible to create and resolve aliases asynchronously?*

A No, you can't resolve aliases asynchronously because the Alias Manager uses all synchronous File Manager and Device Manager requests.

Q *Is there a QuickTime codec for converting QuickDraw GX pictures to QuickDraw PICT format? If so, can you provide this?*

A At a session during Apple's 1995 Worldwide Developers Conference, a new technique for exporting QuickDraw GX pictures as QuickDraw PICT files was demonstrated. The method makes use of QuickTime and a new codec that's included in the QuickDraw GX extension version 1.1 and later. With this codec, you can embed a flattened QuickDraw GX picture into a PICT file (or a QuickTime movie). We recommend that you use this method if you want to allow your QuickDraw GX application to exchange pictures with existing QuickDraw applications. You can find sample code demonstrating the use of this codec in the Macintosh Technical Q&A "Embedding a GX Picture into a PICT" (GX 07).

One important feature of this codec is that it does not convert QuickDraw GX pictures to QuickDraw PICTs in the traditional sense of the word "convert." What it allows is the embedding of QuickDraw GX objects inside a PICT file. The advantage of this is that it allows QuickDraw GX pictures to be viewed (but not edited) in any application that can open a PICT file. Although "embedding" is very useful, it's quite different from "conversion."

Strictly speaking, it's not possible to convert QuickDraw GX pictures to QuickDraw PICTs without loss of information, because QuickDraw GX has much greater functionality than traditional QuickDraw. You can, of course, draw the QuickDraw GX picture offscreen and capture the result in a QuickDraw PICT, but you'll lose much of the information. There's no way to represent

complex transfer modes, perspective, advanced typography, and so on under the QuickDraw imaging model. By using the new codec, you don't lose any of the QuickDraw GX features.

Note that this technique is quite different from that used in the older PicturesAndPICTLibrary.c, which embeds a QuickDraw GX shape into a PICT by using picture comments. We recommend that you use the codec instead because picture comments have several weaknesses, including these:

- They're limited to 32K.
- Many applications strip out any picture comments they don't recognize.
- DrawPicture ignores all picture comments.

Using the codec to embed the QuickDraw GX picture avoids these problems.

Q *How can I find out which printer is selected in the Chooser?*

A Under the old printing architecture, you can locate the driver for the currently selected printer by accessing the 'STR ' -8192 or 'alis' -8192 resource in the System file. The 'STR ' -8192 resource contains the name of the current driver and the 'alis' resource contains an alias record that will take you right to the driver. Note that with older system software the 'alis' resource doesn't appear in the System file. If the 'alis' resource is present, resolve it; if not, look in the Extensions folder and in the System Folder for a file with the same name as 'STR ' -8192.

With QuickDraw GX installed, the 'STR ' -8192 resource still exists for backward compatibility with applications that don't use QuickDraw GX printing. In this case the 'STR ' -8192 resource gives the name of the default desktop printer file. For applications that are QuickDraw GX savvy, the concept of a default printer isn't important because the user can pick any printer from the QuickDraw GX Print dialog.

Once you've located the 'STR ' -8192 resource and you have the name of the current printer, you can then determine the printer's zone and type using the 'PAPA' -8192 resource in the driver (if the traditional printing architecture is in use) or by accessing the printer's 'comm' resource (if the QuickDraw GX printing architecture is in use). Sample code demonstrating this can be found in the Macintosh Technical Q&A "Locating the Selected Printer" (GXPD 36).

Q *When I call FDecompressImage during printing, it appears that the custom StdPix bottleneck of the LaserWriter 8.3 driver isn't called. Why not?*

A FDecompressImage doesn't call through the StdPix bottleneck. The workaround is to directly call the StdPix bottleneck in the current graphics port (or the StdPix obtained from calling SetStdCProcs if there are no custom bottlenecks in the current graphics port). For more information, see "Printing Images Faster With Data Compression" in *develop* Issue 24.

Q *When I attempt to open the built-in Ethernet driver with the OpenDriver call on a PCI-based Macintosh, the call fails. What's the correct way to access the built-in driver on these new Macintosh computers?*

A The new PCI-based Power Macintosh computers use Open Transport for network services. This architecture is a precursor to the changes expected for Copland. Since Open Transport is PowerPC-native, there's no longer a dependence on the 680x0-based Device Manager and Slot Manager. To maintain compatibility for the built-in Ethernet driver, an ENET shim was implemented so that applications that called the Ethernet driver directly could continue to work. (Note that the ENET shim is missing from the original Power Macintosh 9500 release, but became available with later software updates.)

The ENET shim opens when OpenSlot is called to open the Ethernet driver in NuBus slot 0. The shim intercepts this request and loads a .ENET driver entry into the driver table. Subsequently, applications that call OpenDriver will get the driver reference number for the shim driver, which then handles the various Control calls it receives. The shim works only for the built-in Ethernet device, not for an installed PCI Ethernet card.

For some code demonstrating how to do this, see the Macintosh Technical Q&A "Ethernet Error on a PowerMac" (NW 14).

Q *How can I check whether the Open Transport IP protocol stack is loaded?*

A Open Transport provides the option to delay the complete loading of the protocol stack. This reduces the use of system memory for the IP protocol stack until a TCP/IP application is launched. To check whether the stack is loaded, call OTInetGetInterfaceInfo. If it returns an IP address, the stack is loaded. If the returned address is 0 or the call fails, the stack is not yet loaded. Note that the call to OTInetGetInterfaceInfo doesn't force the load of the IP stack.

Q *When we call DirFindRecordGet, we get the message kOCEInvalidCommand (-1501). Is there another way to get all records from a given catalog?*

A The catalog about which you're attempting to get record information doesn't support the DirFindRecordGet function (few out there actually do). To check whether a particular catalog supports this function, you need to first call DirGetDirectoryInfo and check the features flags that are returned. Check the kSupportsFindRecordBit (see *Inside Macintosh: AOCE Application Interfaces*, page 8-31) to see if this call is supported. If it's not supported, you'll have to use DirEnumerateGet instead to get all the records from a catalog.

You might want to look at the DTS Catalog Peek sample code on the Mac OS Software Developer's Kit, which uses the DirEnumerateGet call.

Q *Sometimes, after I copy an HFS volume one-to-one to a CD-ROM, aliases that look perfectly fine on the source volume are disconnected on the CD-ROM — the Alias Manager claims that it can't find the volume. What should I do to detect and fix a possible disconnected alias before writing it to CD-ROM?*

A Sometimes, when aliases move from hard drives to CD-ROMs, volume information changes, rendering the alias unresolvable. The Alias Manager requires the following pieces of information in order to identify a volume:

- the volume's name
- the volume's creation date (which should be a unique number)

- the volume's kind (ejectable, nonejectable, floppy disk, or foreign file system)

The Alias Manager expects all three pieces of information to match. If they don't all match, the Alias Manager attempts to identify the volume by matching two of the three items, trying for a volume match in this order:

- by name and creation date
- by creation date and volume kind (if the volume name changed)
- by name and volume kind (if the creation date is not stable, as with some network file systems)

When pressing a CD-ROM, you're moving aliases from a hard drive (nonejectable volume kind) to an ejectable volume kind. If the volume name or creation date of the hard drive changes after alias creation, the aliases may not resolve properly. You can avoid this problem by ensuring that the volume name of the hard disk doesn't change while you're building a CD-ROM's content. Also, do not back up, reformat, or restore a hard disk while you're building a CD-ROM's content, so that the creation date doesn't change.

Sometimes, valid-looking aliases fail to resolve. Because the Finder creates alias files, the Finder is responsible for resolving them. The Finder doesn't always check and update aliases as carefully as you might. Additionally, the Finder always uses a relative search path when resolving aliases.

You might want to test to see whether installing QuickTime makes a difference in the cases where perfectly valid looking aliases fail to resolve. QuickTime includes patches that make the Alias Manager work better.

Q *What does holding the Shift key down at startup turn off under System 7?*

A This information isn't documented; the following list isn't guaranteed complete or accurate and is certain to change in the future. Under System 7.0 through System 7.5 the following files are explicitly skipped:

- A/ROSE
- Virtual memory
- Files of type 'scri' (Roman still works), 'cdev', 'RDEV', 'INIT', 'cbnd', 'fbnd', 'tbnd', 'adev', 'ddev', 'appe', 'fext', 'AINI', and 'thng'
- Finder Startup and Shutdown items (since the Finder Scripting Extension controls these tasks)

MacsBug will not load under System 7.0, but it will load under System 7.5 if both the Option and Shift keys are held down. In addition, System 7.0 sets the disk cache to 64K, while System 7.5 sets it to 96K.

Q *What does turning everything off in the Extension Manager actually turn off?*

A The only files the Extension Manager turns off are those that it shows. Which files will be turned off isn't documented; the following list isn't guaranteed complete or accurate and is certain to change in the future.

There are four creator types that the Extension Manager doesn't show: 'mntr', 'DMOV', 'extE', and '8INI'. Items of type 'extE' and '8INI' aren't shown

because the Extension Manager extension has the creator of 'extE' and the Extension Manager control panel has the creator of '8INI'. This prevents you from using the Extension Manager to disable itself.

Also, the Extension Manager won't show any items of type 'INIT', 'RDEV', or 'cdev' if they have the "no INITs" Finder flag set.

The Extension Manager shows only items of type 'INIT', 'RDEV', 'cdev', 'PRES', 'PRER', 'adev', 'fext', 'scri', 'cbnd', 'fbdn', 'tbnd', 'ddev', 'appe', 'gc24', 'adrp', 'dbgr', 'dfil', 'APPL', 'FFIL', 'pext', and 'vbnd'.

Q *Do I always need to call PrJobDialog to print a document? Why?*

A Yes, you do. The reason for this is that many drivers (notably LaserWriter 8) don't initialize the job-specific settings until PrJobInit is called. Without this call, they fall back on the default, which is usually stored in the driver in the PREC 0 resource.

The normal definition of the PREC (which maps to a TPrint structure) doesn't have as much space as LaserWriter 8 needs. Because of this, LaserWriter 8 stores some settings in this PREC 0 resource and others in the LaserWriter 8 Prefs file. This separation of LaserWriter settings can wreak havoc on a job run without the PrJobDialog call.

If you absolutely *must* avoid displaying the job dialog box, there are two ways to work around it. Note that these are not supported methods, and by using either of them you'll make your application hostile to QuickDraw GX and to Copland.

- Have users invoke the Print dialog as part of their preferences setup, and save the resulting print record. Every time you print, merge that print record in with a call to PrJobMerge. This way each document can have its own page setup, accommodating things like printing on A4 paper instead of US letter.
- "Display" the job dialog, but never let the user see it. You can accomplish this by calling PrJobInit, moving the resulting dialog offscreen, patching ModalDialog, and calling PrDlgMain. Your patched version of ModalDialog can return 1 for the OK button immediately, and you've got the added benefit of actually calling all the code and having a relatively normal print loop.

Q *I'm writing a printer driver, and I've noticed that when I print a window from the Finder with my driver the icons don't show up. What gives?*

A What you've uncovered is an "optimization" in the Icon Utilities. When drawing an icon, the Icon Utilities use CopyMask rather than go through the standard bottlenecks. This is true unless you're saving to a PICT or you set a certain low-memory global (which isn't well documented) indicating that CopyBits should be used.

The following two macros tell the Icon Utilities to use CopyBits instead of CopyMask:

```
#define setPrinting() {*((short *)0x948) = 0;}
#define clearPrinting() {*((short *)0x948) = -1;}
```

Call setPrinting in your PrOpenPage function and clearPrinting in your PrClosePage function, and all should be well.

Q *Which LaserWriter drivers are ColorSync aware?*

A Currently, LaserWriter driver 8.3 is the only ColorSync-aware LaserWriter driver; versions earlier than 8.3 are not.

Q *What does the 7.5.2 Printing Update 1.1 update? Why do I need it?*

A This extension fixes a printing problem that may occur on Power Macintosh 7200, 7500, 8500, and 9500 computers using System 7.5.2. Without this fix, your computer may freeze if you attempt to print on a network-based printer that's busy.

The update contains a new version of the LaserWriter driver (8.3.2) and also a fix to serial DMA. The changes fix the ATP and PAP networking protocol layers.

An updated version of the PAP.WrkStation.o library will be distributed on a future version of the Mac OS Software Developer's Kit; developers who have licensed the library will be notified when the new library is available.

Q *Do QuickDraw 3D mesh contours run in the same or the opposite direction as the face? For example, if the face runs clockwise, will the contour run counterclockwise?*

A The exterior face needs to be defined in a counterclockwise direction, but the contours defining holes can run in either direction. The mesh code is able to identify holes in a face.

Q *I'm using QuickDraw 3D 1.0 on a Power Macintosh 7100. If I try to launch my application when QuickDraw 3D is disabled with the Extension Manager, I get a message that the application couldn't be opened because QuickDraw 3D could not be found. Since the application has to run even if the user doesn't have QuickDraw 3D, what should I do ?*

A You're "hard linking" to the QuickDraw 3D shared library. You need to "weak link" to the library instead. With Metrowerks CodeWarrior this is trivial: simply select the project window, click the small triangle to the right of the library in the window, and choose the Import Weak option from the pop-up menu. In your code, use the Gestalt selector for QuickDraw 3D to determine whether the library exists. If it does, you should additionally check at least one symbol in the library against kUnresolvedCFragSymbolAddress to be sure the library was linked successfully, as described in *Inside Macintosh: PowerPC System Software* on page 1-25. It's possible for Gestalt to indicate that the library is available even though the weak link failed — for example, if there isn't enough memory available.

Q *I've been trying to use QuickTime to step through a movie of PICTs activated by keyboard input (that is, press a key and the next frame is displayed). My problem is that I can't consistently step one frame at a time. What's the easiest way to move to the next frame?*

A The easiest way is to use a movie controller to accomplish this. You can send mcActionStep to the controller to bump the movie to a new frame. A little more work, but perhaps more suitable for you if you don't want to use a controller, is to use GetMovieNextInterestingTime to find the next frame.

Q *Is there any way to ask QuickTime, at the operating system level, whether any movies are currently playing?*

A No, there's no supported way to do this.

Q *We're writing a screen saver that plays QuickTime movies. Our WaitNextEvent loop is very basic. We've noticed that other background applications don't get any time, even if we use WaitNextEvent and make sure that MoviesTask doesn't spend too much time playing the movie. However, if we add code to track update events with BeginUpdate and EndUpdate the problem goes away. Why?*

A QuickTime and other parts of the operating system are sending update events to your application. If these events aren't handled, they're resent, resulting in no time yield to other applications. By calling BeginUpdate and EndUpdate or otherwise taking care of the update event inside your event loop, you allow yielding to other applications. See Macintosh Technical Note "Pending Update Perils" (TB 37) for more information.

Q *Is the data rate stored somewhere in a QuickTime movie? If not, how can I compute it?*

A The data rate isn't stored in the movie, because a QuickTime movie isn't required to have a constant data rate: it can change over the duration of the movie. Typically, in the case of video, one sample equals one frame; in the case of sound and other media, this one-to-one relationship doesn't necessarily hold. Additionally, none of the video samples in a continuous stream are exactly the same size, even if in practical terms this is often assumed.

There are several possible methods of measuring the rate of samples, but the quickest and easiest is to do what Movie Player does: for each track, get the media size and divide by the media duration. This provides a rough estimate of the data rate that should be suitable for most purposes and works for all types of movies, video and otherwise.

Q *It's critical in my CD project to be able to load small QuickTime loop movies entirely into RAM. This is still not supported in QuickTime for Windows 2.0.3. Will this functionality be available soon, or should I focus on developing a workaround?*

A Support for loading a movie into and playing it from RAM will not be in QuickTime for Windows 2.1. It is, however, still on Apple's to-do list. For now, you can improve performance by copying your small movie to a temporary file on the hard drive. From there you can force it into the disk cache or DOS buffers by opening and reading all of it yourself.

Q *How can I place blue-screen video over a QuickTime VR panorama?*

A For starters, you need to know the exact view over which you want to place the video. Note that if you're only warping in one dimension, it may be a bit tricky.

To get a very close match, take the following steps:

1. Push each individual frame of the video sequence through the Stitcher (assuming that the motion fits within a single photograph) with **wrap** turned off, and the same **vfov** set as for the panorama. The resulting images will be warped into the same space as your complete panorama.
2. Either turn your single-frame image into a partial panoramic movie or replace the appropriate part in your background panorama with the single frame.
3. Run the image through the **p2mv** and **msnm** tools, and use QuickTime VR to dewarp it with **warpMode 1** with the precise **hpan**, **vpan**, and **zoom** data set. You may want to use **p2mv** with the "raw" compressor to maximize your image quality.
4. Capture the image from the screen.
5. If you do this for every image (it can be automated with scripting), you should get a completely matched motion sequence that you can turn into a QuickTime movie with standard tools. This is where you should do your compression (not at step 3).

This should mostly take machine time, not your time. Steps 1 and 3 can be scripted in MPW. Step 2 can be scripted in AppleScript using PhotoFlash or in DeBabelizer, and step 4 in HyperCard or Director. Step 5 uses ConvertToMovie. Once you develop these scripting tools the first time, each sequence should be pretty quick to fire up.

Q *My cat exhibits a behavior I hope you can explain. Every now and again, he'll sniff something with particular attention and intensity — the bend of my wrist, say, or a spot on the rug. So far so good. But then when he lifts his head, he holds his mouth open, his lower jaw hanging stupidly. It looks ridiculous! And he seems totally unaware of it, invariably sitting there for several seconds, mouth gaping, looking around blithely as if there's nothing out of the ordinary, before finally licking his chops and closing his mouth. What's going on? Is there any way I can curb this behavior? It's embarrassing.*

A First off, don't worry. Your cat is perfectly normal. All cats do this, and there's nothing wrong with it (except of course that it looks silly). Your cat is simply making use of a little-known sensory organ called the *vomeronasal* organ, or Jacobson's organ. It's a second scent organ, located far forward on the roof of the mouth, and is supplemental to — but distinct from — the nose. It's even wired into slightly different areas of the brain, areas dealing with feeding and complex sexual behaviors. Many other animal species have a Jacobson's organ, from rattlesnakes to bighorn sheep, but humans don't.

The behavior you've noted is called the *flehmen* reaction (*flehmen* is a German word with no satisfactory English translation). By holding his mouth open and not breathing, your cat is concentrating the molecules to be sensed over his Jacobson's organ, allowing it to do its job. The behavior is exhibited by all cats, domestic and wild, large and small.

These answers are supplied by the technical gurus in Apple's Developer Support Center. For more answers, see the Macintosh Technical Q&As on this issue's CD or on the World Wide Web at http://dev.info.apple.com/techqa/Main.html. (Older Q&As can be found in the Macintosh Q&A Technical Notes on the CD.)•

Printing, Patching, and Fonts

See if you can solve this programming puzzle, presented in the form of a dialog between guest puzzlers Dave Hersey and Cameron Esfahani (cam). 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.

DAVE HERSEY AND CAMERON ESFAHANI

	Dave	Hey cam, it's kinda quiet. Where are KON and BAL?
	cam	Since the local salad bar closed, I haven't seen KON. BAL disappeared after he left the video game industry. Have you been getting enough sleep? You look tired.
	Dave	I've been under a lot of pressure to track down this bug.
	cam	Maybe I can help. What's the problem?
	Dave	I have a Power Mac 6100/66 running System 7.5 with QuickDraw GX 1.1. When I try to print from a word processor, I get the message "The application has unexpectedly quit, because an error of type 11 occurred." What's an error of type 11?
	cam	That's an unhandled exception from native code. What word processor are you using?
	Dave	Um, a very large one in a very large office suite from a very large company up north.
	cam	Have you updated to version 1.1.3 of QuickDraw GX?
	Dave	Yeah. The problem still happens.
	cam	Does it happen on any other machine?
100	Dave	Yes. It crashes on any Power Mac but works fine on 680x0 machines.
	cam	Hmm. Is the word processor native on the Power Mac?
	Dave	Yes — it's fat.

DAVE HERSEY (AppleLink HERSEY) works in the QuickDraw GX PrintShop level 4 bio-containment facility, thousands of feet beneath the Cupertino R&D campus. There, he develops PowerPC-native QuickDraw GX printing code, works on Copland, and relaxes by dabbling with an occasional hot agent over lunch.•

CAMERON ESFAHANI (AppleLink DIRTY, Internet dirty@powertalk.apple.com) is the shortest member of the Graphics team at Apple. To add a few more inches to his height, he sometimes wears roller blades in meetings. If that doesn't help, he has been known to don his large purple hat with sparkles.•

cam It sure is. But I have the same version of system software and the same word processor, yet my machine doesn't crash.

90 Dave Well, I have a standard system installed, but I added a bunch of whizzy fonts.

cam If I install one of your fonts, will my machine crash?

Dave Sometimes. If you install all my fonts, it crashes all the time.

cam That's easy, then: bad fonts. Here, take out this Thingamajigs font.

80 Dave No way, man. This is a standard bitmap-only font. It should work. Ike's machine doesn't have Thingamajigs on it and his machine still crashes.

cam Does he have bitmap-only fonts installed?

Dave Yes.

cam At what point in the printing process do you crash?

Dave The crash occurs just as the application starts spooling the print file.

cam Is this word processor QuickDraw GX–aware?

70 Dave Yes. It has support for the new QuickDraw GX print dialogs, and it calls the QuickDraw GX translator to translate QuickDraw drawing commands into QuickDraw GX shapes during printing.

cam Good for them. Have you tried to reproduce the crash with other QuickDraw GX–aware applications?

Dave Yup. I tried to reproduce it with several QuickDraw GX–aware and QuickDraw GX–savvy applications. No luck.

cam Try running the 680x0 version of this program on your Power Mac. It will be slow and piggy, but try it anyway.

60 Dave The problem went away! So, the crash seems to have something to do with the PowerPC code in this application.

cam Hmm. Let's install MacsBug and take a look at this from the debugger.

Dave I tried that before, but I couldn't see any symbols in the PowerPC code where it crashes. I couldn't tell which routine the PC was in.

cam You should install the new version of MacsBug. Version 6.5.2 understands native exceptions and can use embedded symbols.

Dave Nifty. . . . OK, I've done that. But I still crash.

cam Why do you crash? Type **how**.

Dave MacsBug claims that there was a "PowerPC access exception at 001DB030 ConstructNFNTDirectory+002B4."

cam What does ConstructNFNTDirectory do? Hey, wait, there's Alex Beaman. Alex, can you help us out here?

50 Alex Sure. QuickDraw GX views all fonts as type 'sfnt'. It's really elegant: ConstructNFNTDirectory will make an NFNT font appear to have an 'sfnt' directory. It can build either just the directory header or the entire directory, and this is controlled by a Boolean parameter passed into the function. OK, gotta run!

Dave Thanks, Alex. When I disassemble ConstructNFNTDirectory with MacsBug, I get this:

```
ilp ConstructNFNTDirectory
 Disassembling PowerPC code from ConstructNFNTDirectory
  ConstructNFNTDirectory
   +00000 001DAD7C  stmw      r14,-0x0048(SP)
   +00004 001DAD80  mflr      r0
   +00008 001DAD84  clrlwi    r27,r5,0x18
   +0000C 001DAD88  addi      r28,r3,0x0000
   +00010 001DAD8C  mfcr      r12
   ...
   +00060 001DADDC  addi      r3,r30,0x0000
   +00064 001DADE0  addi      r4,r28,0x0000
   +00068 001DADE4  bl        GetNoLoadResource
   ...
   +000E4 001DAE60  addi      r3,r20,0x0000
   +000E8 001DAE64  bl        ComputeSearchFields
   +000EC 001DAE68  crmove    cr7_SO,cr7_SO
   +000F0 001DAE6C  cmpwi     cr2,r27,0x0000
   ...
   +002B4 001DB030  *lwzx     r5,r19,r5
   ...
   +002F0 001DB06C  lhz       r5,0x0004(r20)
   +002F4 001DB070  li        r16,0x0001
   +002F8 001DB074  addic     r5,r5,0x0001
   +002FC 001DB078  sth       r5,0x0004(r20)
   +00300 001DB07C  beq       cr2,ConstructNFNTDirectory+00324
   ...
   +003C8 001DB144  addic     SP,SP,0x00A0
   +003CC 001DB148  mtcrf     0x38,r12
   +003D0 001DB14C  mtlr      r0
   +003D4 001DB150  lmw       r16,-0x0040(SP)
   +003D8 001DB154  blr
```

cam An access exception means we're trying to read or write to an invalid address. That, of course, could be caused by many things, such as uninitialized variables or trashed memory. Let's check the heaps with **hc**.

Dave Both the system heap and the application heap are fine.

cam OK, I restart the program and use **brp** in MacsBug to set a breakpoint at ConstructNFNTDirectory. **brp** is just like **br**, except it works for PowerPC code. After I start printing and the breakpoint is hit, I step through this function to follow the code flow.

45 Dave At offset 0x0300 you don't take that branch, and you eventually begin executing code that will corrupt the QuickDraw GX heap.

cam But that's wrong — we should've taken that branch. The caller didn't ask ConstructNFNTDirectory to create the entire directory, just its header; it didn't allocate enough space for all of it. Check the heaps again.

Dave The heaps seem fine. QuickDraw GX allocates out of its own heap, which MacsBug doesn't know about. Even if it did know about it, it wouldn't be able to tell us if the heap was corrupt, as QuickDraw GX has its own memory manager.

cam Darn, memory corruption bugs are the worst. You can trash memory and not see the effects of it until you're miles away from that code. OK, why didn't it take the branch at offset 0x0300?

40 Dave Well, CR2 is true, so the branch won't be taken.

cam How can you tell that CR2 is true?

Dave The PowerPC chip has eight condition register fields, CR0 through CR7, stored in nibbles in a 32-bit condition register (Dave Evans talked about this in his column in *develop* Issue 21). So the value of CR2 would be bits 8 through 11 of the condition register. The chip has its bits numbered from 0 through 31, from left to right. We can tell that CR2 contains a true value because its second logical bit isn't set. That bit corresponds to the equals operator, so the fact that it's 0 means the operation that set this register was not equal.

cam Who sets up CR2?

Dave The code at offset 0x00F0. As Alex mentioned, one of the parameters to this function is a Boolean that controls whether the whole directory is created or only the header. Because this parameter is a Boolean, the PowerPC processor can just compare it against 0 and use the result as a flag for later branches. Parameters passed in PowerPC code are put from left to right into registers R3 through R10; since this parameter is the third parameter to the function, it's passed to the routine in register R5. (A much better description of this is in *Inside Macintosh: PowerPC System Software*.)

cam I love this chip. I'll reexecute the program and get back to the start of this function and examine CR2.

35 Dave It starts out false.

cam So someone's trashing it along the way. Well, we can't use some of our normal tricks for detecting when memory gets trashed. One problem is that step spy doesn't work yet for PowerPC. Another problem is that we would want to step spy on CR2, which is a register, and step spy never worked on registers. We'll have to do this the hard way: let's step through this function, watching CR2 to see just when it gets changed.

30 Dave The subroutine GetNoLoadResource at offset 0x0068 changes CR2 from false to true. GetNoLoadResource is a wrapper to GetResource.

cam I restart the program and trace over the GetResource call.

Dave Yep, that's the function that trashes CR2.

cam Is it legal for the compiler to rely on CR2 being preserved across function calls?

25 Dave Yes. According to the PowerPC ABI (Application Binary Interface) documentation — section 3.6 in the first edition — CR2 through CR5 are nonvolatile and need to be saved across function calls.

cam Look at the code for GetResource. Since in System 7.5 GetResource is a native trap with a routine descriptor, I can use the MacsBug dcmd **drd** to dump that out. Here's what I get:

```
drd GetResource
 The RoutineDescriptor at:    011EDFEC
   Mixed Mode Magic Trap: AAFE, version: 07,
   routine descriptor flags: 00 (NotIndexable),
   loadLocation: 00000000, reserved2: 00,
   selectorInfo: 00 (No Selector),
   routine count: 0000
```

```
--- Routine Record 00000000 -----
procInfo: 000002F0, reserved1: 00, ISAType: 01 (kPowerPCISA),
Routine Flags: 0004 (IsAbsolute, IsPrepared, NativeISA,
   PassSelector, IsNotDefault), procPtr: 01219EEC,
storedOffset: 00000000, selector: 00000000
```

20 Dave There's only one routine associated with the trap and it's the native implementation.

cam Where's that function? On the Power Mac, every ProcPtr is actually a data structure that contains the routine's real address and TOC. This is called a TVector (transition vector). This allows every fragment to have its own globals, because the correct TOC gets loaded for each routine by the runtime environment. So, to find the routine's address, you need to dereference the ProcPtr.

```
wh 1219eec^
 Address 00E77B78 is in the "Porky WordProcessor" heap at 00DFC430
 The address is in a CFM fragment "Porky WordProcessor" [non-write exec]
 It is 00073058 bytes into this heap block:
  Start      Length       Tag  Mstr  Ptr  Lock  Prg  Type  ID  File  Name
· 00E04B20   003D35D8+0C  N
```

15 Dave Apparently it's in the heap of the application.

cam So this program is patching GetResource. At least they have a native patch — a good habit these days because you don't know what traps will go native from now on. If you're patching native PowerPC code with 680x0 code, performance-sensitive code will run slower. For this reason, you should make all of your patches fat. Let's disassemble the patch on GetResource.

```
ilp 1219eec^
 Disassembling PowerPC code from 1219eec^
  No procedure name
      00E77B78  stwu    SP,-0x0058(SP)
      00E77B7C  mflr    r12
      00E77B80  stw     r12,0x0060(SP)
      00E77B84  stmw    r26,0x0040(SP)
      00E77B88  stw     r3,0x0070(SP)
      00E77B8C  sth     r4,0x0074(SP)
      00E77B90  extsh   r4,r4
      00E77B94  lis     r5,0x4D42
      00E77B98  ori     r5,r5,0x4446
      00E77B9C  cmplw   cr2,r3,r5
      ...
      00E77C10  lmw     r26,0x0040(SP)
      00E77C14  lwz     r12,0x0060(SP)
      00E77C18  mtlr    r12
      00E77C1C  addic   SP,SP,0x0058
      00E77C20  blr
```

10 Dave At 0x00E77B9C they do a compare and store the result in CR2. However, they don't save and restore CR2 across this function, so it's trashed when we return to ConstructNFNTDirectory.

cam OK, I restart the program and manually save and restore the value of CR2 across the GetResource calls. I do this by futzing with bit 2 in CR2.

5 Dave Everything prints fine.

cam It looks like a compiler bug. Either they shouldn't be using CR2 or they should be preserving it. In any case, the GetResource patch is trashing CR2, and that changes a Boolean which causes us to read in extra data. The caller never allocated enough space for the extra data, so the QuickDraw GX heap gets corrupted.

Dave Holy cow! A compiler bug. Shouldn't we notify the compiler developer?

cam Well, this company has their own in-house development tools group. They write their own compilers, linkers, and debuggers. We should contact them anyway, so that they can create a patch that fixes this problem. *[This patch, "Office4.2x Update for Power Mac," is now available on most online services.]*

Dave Why are they patching GetResource?

cam It looks like they were looking for resources of type 'MBDF' (menu bar definition procedures). I can tell this from the instructions at addresses 0x00E77B94 through 0x00E77B9C. The PowerPC architecture has a limitation of 16 bits on the size of an immediate constant. So, if you wanted to compare a value against a 32-bit constant, you would have to build the 32-bit value with two instructions. This is what occurs at addresses 0x00E77B94 and 0x00E77B98, where they insert 0x4D42 and 0x4446 together into a 32-bit value. If you look at the ASCII of this constant, it's 'MBDF'. At address 0x00E77B9C, they compare this constant to the resource type parameter passed to GetResource. Since that parameter is the first parameter, it will be in register R3.

Dave Why didn't we crash when we had only one NFNT font installed?

cam This patch would cause ConstructNFNTDirectory to always overwrite the buffer passed in. But that wouldn't always cause your machine to freak out. By adding enough NFNT fonts, we trashed the QuickDraw GX heap significantly enough to cause the crash.

Dave Wow, all this and it was an application patch that caused the problem! It sure would have been cool if we could have used the **patch** dcmd.

cam Yeah. The **patch** dcmd does works on the Power Mac — but we didn't know that was the problem when we started.

Dave It's interesting that it was an application bug. That would explain why I crash in a spreadsheet application by the same company. They share the same patch.

cam Nasty.

Dave Yeah.

SCORING

80–100 You could have a promising career writing compilers for a company up north.
45–70 Dr. MacsBug could always use another assistant.
25–40 Don't worry, it took us a while to figure it out too.
5–20 Visual Basic fan, are you?•

Thanks to Alex Beaman, Tom Dowdy, Ron Voss, KON (Konstantin Othmer), and BAL (Bruce Leak) for reviewing this column.•

INDEX

For a cumulative index to all issues of *develop*, see this issue's CD.•

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

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 This complimentary catalog offers worldwide access to development tools, resources, training products, and information for anyone interested in developing applications on Apple platforms. It 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 and Mac OS–compatible hardware, 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, multimedia self-paced courses and low-cost mini-course tutorials are available through the Apple Developer Catalog.

Macintosh Developer Programs Macintosh developers have a choice of three programs, each providing technology seeding, development software, technical information, discounts on equipment, and more. The programs vary in the level of technical support provided.

The Macintosh Associates Program is a low-cost self-support program for Macintosh developers who don't need programming-level technical support from Apple.

The Macintosh Associates Plus Program enables Macintosh developers to have up to ten programming-level technical support questions answered (via e-mail) per year.

The Macintosh Partners Program is for developers who need unlimited programming-level technical support (via e-mail).

Newton Developer Programs Newton developers have a choice of three programs, each providing technical information as well as discounts on equipment and developer training. The programs vary in the level of technical support provided.

The Newton Associates Program is a low-cost self-support program for Newton developers who don't need programming-level technical support from Apple.

The Newton Associates Plus Program enables Newton developers to have up to ten programming-level technical support questions answered (via e-mail) per year.

The Newton Partners Program offers Newton developers unlimited programming-level technical support (via e-mail) along with additional hardware purchasing privileges and platform seeding opportunities.

Apple Developer Catalog To order products 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 AppleLink APDA, Internet apda@applelink.apple.com, America Online APDAorder, or CompuServe 76666,2405. Or write Apple Developer Catalog, P.O. Box 319, Buffalo, NY 14207-0319.

Apple Developer University (DU) Course descriptions and schedules can be found in the Developer Services areas on AppleLink (Developer Support) and the World Wide Web (http://dev.info.apple.com). You can also call (408)974-4897, fax (408)974-0544, send e-mail to AppleLink DEVUNIV, or write to DU at Apple Computer, Inc., 1 Infinite Loop, M/S 305-1TU, Cupertino, CA 95014.

Apple Developer Programs Call the Developer Support Center at (408)974-4897, send e-mail to AppleLink DEVSUPPORT, or write Developer Support, Apple Computer, Inc., 1 Infinite Loop, M/S 303-2T, Cupertino, CA 95014, for information or an application form. Developers outside the U.S. and Canada should instead contact the Apple office in their country for information about developer programs.