The Wayback Machine - https://web.archive.org/web/20000819144834/http://www.mactech.com:80/articles/…

MacTech MAGAZINE

**Volume Number: 14**
**Issue Number: 7**
**Column Tag: Power Graphics**

# 3D Programming With QuickDraw 3D

*by Kas Thomas*
*Edited by the MacTech Editorial Staff*

### *Getting up-to-speed quickly in 3D programming doesn't have to be painful*

## Introduction

Have you noticed how the whole world has gone nutty over 3D graphics lately? Computer-generated imagery was already coming on strong before the movie *Titanic* hit, but now you can hardly open a magazine or turn on the TV without being bombarded with raytraced logos. "Three-D" is no longer just the province of gamers and VRML diehards; it's mainstream.

If you haven't considered tapping the power of 3D graphics in your own programming (or maybe you ruled it out because it seemed like such a monumental undertaking), now might be a good time to rethink the whole issue. As you may know, in 1995 Apple unveiled a cutting-edge "enabling technology" for 3D programmers called QuickDraw 3D, which allows easy cross-platform access to a comprehensive library of highly optimized 3D routines. You no longer have to be a Ph.D. mathematician to get 3D graphics to happen. With the arrival of the latest G3 chipsets, the Mac platform is fully capable of handling the formidable computational demands of 3D graphics. There's no longer any reason to fear 3D. The way has been paved.

In this article, we'll take a look at Apple's extensive QuickDraw 3D API and how you can use it to add that elusive third dimension to the already powerful 2D graphic capabilities of the Mac. Along the way, we'll try to simplify (or at least demystify) some of the seemingly arcane concepts of 3D graphics programming and show how (and why) the QD3D way of doing things generally puts you ahead of the game.

To get the most out of this discussion, you should be comfortable with object programming idioms (although the code will be plain, procedural C) as well as basic 3D concepts, like up, down, and sideways. It also wouldn't hurt for you to have your own copy of the latest QuickDraw 3D software developer's kit (SDK), which is available online at Apple's web site; you'll need it to compile the sample app developed in this article. In keeping with the spirit of QuickDraw 3D, I'll try to hide as many of the ugly details of low-level 3D programming from you as possible while concentrating on how to get maximum onscreen magic to happen with minimum effort. By the time you've finished reading this article, you should at least have an appreciation for the power and scope of QuickDraw 3D, and you'll be in a position (if you so desire) to bootstrap your way up through the rest of the vast, complex, awe-inspiring QD3D API on your own.

## What's So Great About QD3D?

Third-party 3D libraries are nothing new. What makes QuickDraw 3D unique is that it represents the first time programmers have had support for 3D graphics at the core operating system level. Mind you, we're talking about far more than just a fancy set of draw routines here. QuickDraw 3D is breathtakingly broad in scope and was designed from the ground up with developers in mind. Consider some of the design features of QuickDraw

3D:

- Easy access to a comprehensive set of high- and low-level 3D geometries, with support for 20 built-in primitives.
- A fully cross-platform API in which device dependencies are abstracted out to well-isolated layers. (Porting your code to Windows doesn't require a squad of Redmond-trained consultants.)
- Transparent access to graphics accelerators. (No changes to your code are needed to take advantage of QD3D-compliant boards.)
- A flexible cross-platform 3D file format.
- An extensible, plug-in rendering architecture.
- Strong support for texture-mapping using 2D (e.g., PICT) imagery.
- Built-in pointing and picking support. (User selection of 3D objects is easily handled.)
- A 3D pointing-device manager for input devices with more than two degrees of freedom. (These guys think ahead, don't they?)
- A strongly object-oriented API, but 100% accessible in procedural C. (Think of it as all the benefits of C++, with none of the 3-by-5 cards.)
- Efficient use of RAM: QuickDraw 3D takes care of various aspects of object storage so that you don't have to keep large arrays around. (Your QD3D-savvy application can run in a small partition, even if it creates sizable objects.)
- Flicker-free realtime rendering courtesy of automatic double buffering.
- Reasonable speed for most applications.

Some people would quibble with the last statement. Hardcore game programmers, for example, aren't likely to be impressed with QD3D's speed. In my own work, I've found that objects can be created (or instantiated) at speeds upwards of 100,000 polygons per second on a G3 Mac with no graphics accelerator. *Displaying* objects is another matter, of course. Doing a 15-degree rotation of a Phong-shaded 13,500-polygon trigrid (rendered at 320 by 240 pixels) can take a full second, using Apple's interactive software renderer (sans accelerator). That's fast by raytracing standards, but if you're looking to do full-screen realtime animation of complex, photorealistic scenes, you're apt to be disappointed in QD3D (as well as most other software-only solutions). But for many kinds of user interaction, QuickDraw 3D's speed is quite good. The best way to get a handle on this, of course, is to run some tests of your own, using (for example) the sample app developed later on in this article. (Full source code and executables can be found at the MacTech web site ftp://ftp.mactech.com/src/14.07/.)

## What's Not So Great About QD3D?

Before signing on for the steep learning curve associated with all 3D programming (QD3D included), you'll want to consider the known drawbacks of QuickDraw 3D, which are few in number but potentially important, depending on the kind of work you intend to do.

The first potential drawback, which we've just alluded to, is speed. For some types of work, QuickDraw 3D simply isn't fast enough. Custom-written special-use code is almost always faster than general-purpose library code, but be forewarned: You'll have a tough time improving on some of QD3D's optimizations. (Also, remember that any QuickDraw 3D code you write is instantly accelerator-compatible. The real answer to your speed problem may, after all, be better hardware, not better software. Don't discount the possibility, too, that someday soon *all* Apple computers may ship with 3D acceleration hardware built-in.)

Another feature of QuickDraw 3D that's bound to be a disappointment to some is its lack of any kind of built-in raytracing support. (Raytracing is a rendering method that achieves photorealistic effects by accounting for reflection and refraction.) Many highly realistic effects are possible with QD3D's built-in Phong shading facility, but for true raytracing you'll either need to write your own rendering plug-in or license one of the commercial plug-ins available from LightWork Design (see http://www.lightwork.com).

The rendering shortfalls of QD3D also extend to lack of *software-generated transparency effects* (available only when a QD3D-compatible accelerator is present) and lack of support for *constructive solid geometry* (CSG or "Boolean") operations, except when an accelerator is present. If you're already a 3D graphics whiz, you may be able to program your own workarounds for these shortcomings. (CSG, for example, is implemented in the

POV-Ray freeware raytracing program, for which the complete CodeWarrior C source is available on the Internet.)

One other shortcoming of QuickDraw 3D that may annoy some hardcore 3D graphics aficionados is the lack of support for procedural shaders (a la RenderMan or Lightwave). Creative texture-mapping can overcome this flaw in most cases, but texture maps are *not* true 3D effects: A texture map is actually a 2D picture "wrapped" onto a 3D surface. Procedural or volumetric textures, by contrast, are truly three-dimensional, in the sense that if you saw into a 3D tree branch, you'll expose optically correct 3D veins of lignin throughout the cut, no matter how complex the cut or where it occurs. This is hard to fake with a wrapped 2D texture.

The lack of support for procedural textures may seem like a minor annoyance, but it also means there is no support for *bump-mapping*, which has lately become a very popular technique for achieving the appearance of complex surface geometry without the effort and expense (in terms of RAM and computation) of actually modifying the core surface geometry. Again, this is probably a moot point to any but the most demanding 3D dilettante, but it helps explain why QD3D is used only for "preview mode" operations in commercial 3D packages like Inifini-D, Strata Studio Pro, and Lightwave 3D.

Interestingly, procedural shaders have been promised for QD3D since its original release (and will no doubt come some day). But the reality is, it may be a while before you'll see convincing smoke or fog effects in QuickDraw 3D (unless you write your own custom rendering plug-ins).

## The QD3D Way

Bear in mind as we forge ahead that there are several aspects to the 3D programming problem, each one vitally important:

- Geometry *creation*: How to create, represent, modify, and manipulate 3D geometries. (This also has a non-trivial human interface aspect: How best to enable the computer user to *accomplish* the geometry creation task -- still pretty much an open question.)
- Geometry *presentation* (scene arrangement, lighting, and rendering). If you can't present your 3D data in 2D form, your efforts, in all likelihood, have been for naught.
- Geometry *transportation* (i.e., reading and writing 3D data to various physical and virtual media for purposes of storage and interchange).

QuickDraw 3D offers interesting solutions in all three areas. We'll get to the *presentation* and *transportation* issues some other time, but right now let's take a look at the geometry-creation problem.

The digital representation of 3D geometries (that is, finding suitable data structures with which to maintain spatial data) can be challenging, to say the least. You not only need the flexibility to handle arbitrarily complex shapes, but you need to be able to make changes to those shapes on the fly *and* associate a variety of important attributes (like color, transparency, specularity, etc.) with various portions of various objects. Traditionally, 3D graphics programmers have come at the "geometry representation" problem either by encoding shapes *implicitly* (using pure mathematics), or by representing shapes *explicitly*, via large grids or *meshes*. (The 2D analog to this situation is vector versus bitmap graphics.) The advantages of the pure-math approach include efficient use of RAM and resolution independence. The chief disadvantage of this approach is lack of flexibility: It's often just not practical to try to represent complex shapes in terms of, say, NURB patches. (NURB stands for non-uniform rational B-spline, and a *patch* is a surface based on curves. For an explanation of these concepts, see any good graphics text.)

A somewhat more flexible alternative to implicit modelling is explicit representation of geometry through vertex arrays. This time-honored approach has the advantage of letting the programmer model shapes of arbitrary complexity, to any desired degree of accuracy (RAM permitting). The downside of the brute-force "big mesh" method should be obvious: For all but the most trivial objects and scenes, we're talking about huge amounts of array indirection, array reallocation, and array storage. Even if RAM is no problem, managing all that array data can be a nightmare.

Most programmers would agree that the only reasonable way to approach this kind of data-manipulation nightmare is to impose an object paradigm on it, so that data is hidden when necessary, visible when necessary, and the division of responsibilities between and among objects can be known in advance and enforced. This is the approach taken by QuickDraw 3D.

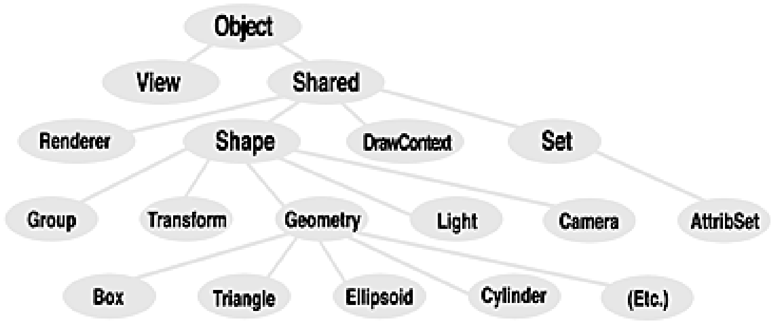

***Figure 1.*** *QuickDraw 3D has a well defined object hierarchy.*

QD3D is an object-based system, in that just about everything is an object with a position in a well-defined hierarchy (see Figure 1). Don't be put off by this if you're not an object-oriented programmer. To use QuickDraw 3D, you don't have to know a bit of C++, because the API is entirely implemented in procedural C. (A Pascal interface is also available; check Apple's web site.) What's important here is that by encapsulating geometric data and methods in objects, Apple has succeeded in hiding a lot of the ugly details of geometry manipulation (and storage) from the programmer, while enforcing uniform behavior, yet allowing access, when necessary, to objects' internals. And the good news is, you can use implicit or explicit geometries, or both. QD3D accommodates conics, quadrics, quartics, and NURB patches, as well as a rich set of polyhedral-mesh tools -- the best of both worlds. Nomenclature

Of course, an object-oriented system the size of QuickDraw 3D could easily become overwhelming if classes, methods, and constants were not organized according to an intuitive and consistent naming scheme. This is one area where QD3D really shines. For example, function names always begin with 'Q3' and take the form ClassName_Method. Thus, Q3Triangle_GetData() is a function defined in the Triangle class that fetches a triangle's data. Likewise, the function Q3Object_Dispose() is defined in the Object class and disposes of any object passed as an argument. Note that since Object is the system's root class, Q3Object_Dispose() can actually operate on any kind of object, including Triangles, which (like other geometric objects) are at the bottom of the QD3D class hierarchy.

Classes and class data structures follow a similar self-documenting nomenclature. All class names (and related data structures) begin with 'TQ3', so that, for example, lights are TQ3LightObjects, shaders are TQ3ShaderObjects, etc.

Constants begin with 'kQ3' and take the form CategoryTypeOption, as in kQ3GeometryTypeBox or kQ3TransformTypeRotate.

The bottom line is that just by reading the variable and function names, you can usually figure out what something does in QD3D -- a good model for all of us to follow.

## Object Sharing and Dereferencing

One of the strong points of QD3D is the way it promotes object sharing, which results in efficient storage and low computational overhead for many types of 3D operations. The vast majority of objects you'll work with in QD3D will inherit from the Shared class. There can be multiple references to shared objects (hence the name), but this gives rise to a serious programming concern in that someone, somewhere, has to keep track of all the references to all the objects so that objects that are no longer needed don't persist and cause memory leaks. QuickDraw 3D keeps track of shared objects by means of a reference count, initially set to 1 the first time an object comes into existence. For example:

```
myNewPoint = Q3Point_New( &pointData );
        // myNewPoint now has a reference count of 1
```

Once you've created an object, you may want to add it to a group, like this:

```
Q3Group_AddObject( myGroup, myNewPoint );
        // myNewPoint's reference count is incremented
```

Note that after the foregoing two calls are made, myNewPoint has a reference count of 2. But once you've added the point to the group, there is no need to keep the second copy around. So it's important to decrement the reference count by means of a dispose call:

```
Q3Object_Dispose( myNewPoint );
        // the reference count for myNewPoint is decremented
```

What confuses a lot of newcomers is that the call to Q3Object_Dispose doesn't necessarily eradicate the object or its data from memory; there will always be a copy of the object in memory as long as the reference count is greater than zero. *To delete an object from memory completely, you must call* Q3Object_Dispose *enough times to decrement the object's reference count to zero.*

It's important to make dispose calls when copies of objects are no longer needed, because failure to do so will cause serious memory leaks as your program's heap fills up with unused objects. But by the same token, you don't want to dereference an object too many times, or the reference count might go to zero when there are still routines counting on the object being there. (This is a good way to crash.) The rule is, *you should dispose of an object before the scope of the variable expires.* Example:

```
{  // Beginning of block; variables come into scope.

  TQ3GeometryObject  myObject = Q3Mesh_New();

  // Do something with myObject.

  . . .

  // The scope of myObject will end at the next
  // closing brace, so be sure to dispose of it
  // before we go out of scope:

  Q3Object_Dispose( myObject );

}  // End of block.
```

That's all there is to it. Disposing of objects as soon as they're not needed quickly becomes second nature, with a little practice. Follow this rule and you'll never get in trouble with memory leaks in QD3D. (Well, maybe not never, but you get the idea. Failing to dereference objects properly is the No. 1 most popular way to generate memory leaks in QD3D.)

What about Group objects? The good news here is that when you dispose of a group with Q3Object_Dispose(), QuickDraw 3D takes care of disposing of the individual constituents for you. In other words, if you have a Group that contains three Transform objects, four AttributeSet objects, and two dozen Geometry objects, you *don't* need to loop over all the members of the group and dispose of them one by one. QD3D does this automatically when you tell it to dispose of the group.

## Staying out of Trouble

Before we move on to some actual code, let's spend a minute talking about what it takes to produce bug-free, leak-free code in a QD3D app. (It's possible. Don't believe the rumors.) We've already mentioned the object-dereferencing gotcha. The next most important caveat involves uninitialized variables: in particular, unassigned attributes. Many of the data structures that QuickDraw 3D uses to create objects (including almost all of the geometric objects, from points on up) include a TQ3AttributeSet field. This is so you can apply colors, transparency, etc. to geometry components in a selective fashion. It does not mean you can ignore the TQ3AttributeSet field in those cases where you don't want to apply colors or other effects. The rule is: Whenever a data structure contains an attribute field, you should always set that field either to a valid AttributeSet object, created with Q3AttributeSet_New(), or set it to NIL. *If you don't intend to assign an attribute, set the field to NIL. Failure to do so may cause your application to crash.* (It's kind of like some of the low-level File Manager calls that expect storage to be set for the ioName field of a parameter block, even if you don't supply an actual name.) Leaving a garbage value in the attribute field of a data structure is an invitation to disaster.

You can get a variety of strange bugs in your QD3D app if you aren't aware of certain compiler issues. First, be sure all enumerated constants are of the four-byte int variety. If you let shorts or longs be used in enums, it's likely QuickDraw 3D will interpret your constants incorrectly, which could be disastrous given the huge number of enumerated constants in QD3D. In CodeWarrior, go into Project Settings and set the 4-byte enum option. In other development environments you may need to issue the appropriate #pragma directive.

QD3D also expects pointers or data of type long, float, or double to be aligned on longword boundaries. The QD3D.h interface file (included in the SDK) contains appropriate #pragma directives for several popular C compilers. Check to be sure yours is included.

QuickDraw 3D defines several levels of abnormal conditions that are flaggable. An error is a generally nonrecoverable condition that causes the currently executing QD3D routine to fail, perhaps fatally. A warning is an unexpected condition that could lead to an error if your application continues execution without handling the warning. A notice is an unexpected condition that, because it is less adverse than either an error or a warning, probably will not cause immediate problems, but is sufficiently abnormal to warrant flagging. QuickDraw 3D notifies your application of errors, warnings, and notices by executing application-defined callback routines that you have previously registered with the QD3D Error Manager. Implementation details are described fully in the Addison-Wesley book *3D Graphics Programming with QuickDraw 3D* (available online; see end of article) and Apple's SDK contains sample code and headers for ready-to-use error handlers. Once a handler has been defined and declared, installing it takes just one line of code:

```
Q3Error_Register( MyErrorHandler, 0L );
```

The result is that when QD3D encounters an abnormal condition in your program, your handler can either put up an appropriate alert box or write a log file to disk (or take other action as you see fit).

Apple has facilitated the bug-squashing process by including special "debug" versions of the various QuickDraw 3D extensions, and a special app called "3Debug," with the QuickDraw 3D SDK. The debug versions of the extensions generate informative error messages in great abundance, and the 3Debug app lets you watch for memory leaks, making the development process much easier and faster. If you use these tools as intended, you won't find yourself spending nearly so much time posting newbie questions to the QD3D mailing list along the lines of "Why does my application crash when I exit QuickDraw 3D?" (The answer to that one is usually: *You didn't manage your reference counts correctly.* See above.)

## A Quick Test

If you've paid attention so far, you should have no trouble passing a quick quiz. See if you can spot the bug in the following line of code:

```
Q3Group_AddObject(myGroup, Q3Point_New( &myPointData ));
```

This nested call, which creates a new Point "in place" and adds it to the myGroup object, will compile, link, and run without errors. But it will leak memory. Why? Because the Point object created by Q3Point_New() has a reference count of 2 when this code is done executing, and there is no way to decrement the reference count properly. If you write it out in multiple lines of code, assigning the output of Q3Point_New() to a local variable, you'll see the problem -- and the solution.

```
myPoint = Q3Point_New( &myPointData );
      // myPoint has a reference count of 1

Q3Group_AddObject(myGroup, myPoint);
      // myPoint has a reference count of 2

Q3Object_Dispose( myPoint );
      // myPoint has a reference count of 1
```

Note the proper way of doing things, which is to create the object in a local variable, pass it as an argument, then dispose of it.

## Initializing and Exiting QD3D

Before you use QuickDraw 3D, you should determine whether it is available on the host system and has been duly recognized by the Code Fragment Manager. First, you should check to see if the address of the Q3Initialize function has been resolved, with a function like the following:

```
Boolean  MyComputerHasQD3D( void )
{
  return ((Boolean) Q3Initialize !=
        kUnresolvedSymbolAddress);
}
```

If Q3Initialize has been resolved, you should now call it to initialize QuickDraw 3D:

```
OSErr    MyInitializeQD3D( void )
{
  TQ3Status  myStatus;

  myStatus = Q3Initialize();

  if ( myStatus != kQ3Success )
    DebugStr( "\pQD3D not available!" );

  return noErr;
}
```

Most often, QD3D routines return a value of kQ3Success or kQ3Failure. (There are many other values, however; check out the header file QD3DErrors.h.) Note that kQ3Success is not the same as noErr. In fact, kQ3Success has a value of 1 whereas noErr and kQ3Failure have a value of zero. Don't use your usual "noErr" error-checking habits with QD3D or you may run into trouble.

When you're done using QuickDraw 3D (i.e., at program termination), you should call Q3Exit() to close the connection:

```
  myStatus = Q3Exit();
  if ( myStatus != kQ3Success )
    DebugStr( "\pQ3Exit returned failure." );
```

If your program drops into the debugger when you make this call, it's probably because you failed to dispose of one or more QD3D objects properly. Be sure to write a cleanup routine that disposes of any persistent objects in your app (such as View objects) at program termination.

## Hello World

Creating geometric objects in QD3D is very straightforward -- anticlimactic, even. But recall that geometry creation is only one aspect of 3D programming. Getting your objects to show up on the screen is another matter. (Providing for easy user interaction is yet another cylinder-of-worms.) Achieving the equivalent of "Hello World" in QD3D takes several hundred lines of setup code. Part of the reason for this is the amount of control QD3D gives the programmer: You have the power to set scores of parameters (involving lights, camera positioning, drawing styles, and object attributes) any way you want. Apple recognized that this might create confusion for non-graphics programmers whose only goal might be to put a 3D model on the screen for a short period of time. Accordingly, QD3D includes a 3D Viewer library that provides a high-level interface for displaying 3D objects quickly and easily. If you're willing to give up a lot of the control you'd otherwise have over the scores of parameters that go into creating a scene from scratch, the 3D Viewer is something you should look into. (See Chapter 2 of *3D Graphics Programming with QuickDraw 3D* or the article by Nick Thompson in develop No. 29.) Working with the 3D Viewer is a lot like working with QuickTime Movie Controllers. With just a few lines of code (literally!), you can get a completely functional user interface on the screen in no time, with minimum pain.

Working with the 3D Viewer doesn't really teach us much about QuickDraw 3D, however, any more than using Movie Controllers teaches you much about the QuickTime architecture. Hence, we're going to skip over the 3D

Viewer routines and go straight to hand-crafting a scene. Our first sample app, Spinning Thang, shows how to put a variety of objects on the screen and animate them. (Note that in the QD3D subculture, there are no things -- only thangs.) In the code for Spinning Thang, there's a lot of more-or-less standard textbook example code for such routine tasks as setting up lights, cameras, and views. But there are a few twists, too, such as:

- Code for making QD3D render to a limited portion (or "pane") of a window. To make the pane easier to see, we also set the QD3D background color to grey.
- Code for creating several different geometries, including some of the newer primitives that shipped with QD3D 1.5, such as the ellipsoid, cone, and torus.
- Object cloning: 3D objects are drawn in quadruplicate, then rotated as a set.
- On-the-fly scene recreation: When you choose a different primitive from the Geometry menu, Spinning Thang disposes of the old scene and recreates a new one (with the new geometry) from scratch, quickly and without memory leaks.
- Code showing how to attach different colors to different portions of a geometry.
- Algorithmic animation: Just for fun, we modify one axis of the rotation with a trig function. (You'd think this sort of thing might slow our animation down, but it doesn't -- it just makes it more fun to watch.)

To keep the code as simple as possible, I've made Spinning Thang a "resource-less" app; all menus and windows are created programmatically, on the fly. Also, to keep the code readable I've been rather lax with error-checking. In a real application, you would handle error-checking much more rigorously. But our compiled app is less than 12K in size, compiles with no warnings, and doesn't elicit any complaints from QD3D. (Space doesn't permit listing all 1,000 or so lines of C code here, so be sure to check this month's CD or the MacTech web site for the full project.)

## Creating Geometric Objects

QuickDraw 3D has a rich set of built-in primitives, ranging from lines and triangles to cubes, ellipsoids, and tori, not to mention the powerful freeform polyhedral primitives (of which there are currently four different types). Most of the 20 or so primitives are created in a similar fashion, which involves first filling out the fields of a data structure unique to the geometry in question, then passing a pointer to that data structure to a "Geometry_New()" function. For example, to create a box, you declare a TQ3BoxData data structure and fill in the fields as follows:

```
typedef struct TQ3BoxData {
  TQ3Point3D      origin;
  TQ3Vector3D    orientation;
  TQ3Vector3D    majorAxis;
  TQ3Vector3D    minorAxis;
  TQ3AttributeSet    *faceAttributeSet;
  TQ3AttributeSet    boxAttributeSet;
} TQ3BoxData;

TQ3BoxData         myBoxData;

Q3Point3D_Set(&myBoxData.origin, 0.0, 0.0, 0.0 );
Q3Vector3D_Set(&myBoxData.orientation, 0.0, 1.0, 0.0);
Q3Vector3D_Set(&myBoxData.majorAxis, 0.0, 0.0, 1.0);
Q3Vector3D_Set(&myBoxData.minorAxis, 1.0, 0.0, 0.0);
```

Note that all numeric arguments in QD3D are of type float. The box's origin is set by a 3D point (three floats, giving x, y, and z coordinates). Don't confuse the box's "origin," by the way, with the *geometric center* of the box. The geometric center is actually at one of the corners. The box's height, length, and width are given, respectively, by the orientation, majorAxis, and minorAxis fields of the TQ3BoxData structure, which you'll notice are 3D vectors. Why vectors? After all, shouldn't the lengths of a box's sides be scalar quantities? Actually, that would be true in the special case where the sides are 90 degrees to one another. But in the interest of full generality, the authors of QD3D wanted programmers to be able to skew the directions of a box's parallel sides in any manner. Hence, a box's axes have direction as well as magnitude, letting you get the same effect as drawing a cube in a non-orthogonal coordinate system. (Imagine that you can, if you want, have x, y, and z axes that are not at right angles to one another.) It's important to understand this point, because many of the other geometric primitives in QD3D work the same way. For example, a sphere, in QD3D, is actually a special case of an ellipsoid: It's the special case where the orientation, majorAxis, and minorAxis of the ellipsoid are equal in magnitude and mutually orthogonal.

After setting up the TQ3BoxData data structure (including attributes: see below), you create the geometry by calling Q3Box_New():

```
  // create the box itself
  myBox = Q3Box_New( &myBoxData );
```

At this point, you can add the box to a group, store it in a global, render it by itself, or do anything you want with it. Recall that adding it to a group object increments the box object's reference count (as explained further above), so once you are done adding it to a persistent group object, you should decrement the box's reference count by calling Q3Object_Dispose(). Listing 1 shows a box-making function, MyNewBox().

**Listing 1: MyNewBox()**

```
MyNewBox()
Create and return a box object.

static TQ3GeometryObject MyNewBox( float wherex,
                                float wherey,
                                float wherez )
{
  TQ3GeometryObject    myBox = nil;
  TQ3BoxData          myBoxData;
  TQ3SetObject        faces[6] ;
  short              face;

  // set up attribute storage
  myBoxData.faceAttributeSet = faces;

  // we'll color sides individually
  myBoxData.boxAttributeSet = nil;

  MyColorBoxFaces( &myBoxData ) ;

  // set up all the box info...
  Q3Point3D_Set( &myBoxData.origin, wherex, wherey, wherez );
  Q3Vector3D_Set(&myBoxData.orientation, 0, 1, 0);
  Q3Vector3D_Set(&myBoxData.majorAxis, 0, 0, 1);
  Q3Vector3D_Set(&myBoxData.minorAxis, 1, 0, 0);

  // create the box itself
  myBox = Q3Box_New(&myBoxData);

  // dispose of the attrib objects we created
  for( face = 0; face < 6; face++)
  {
    if( myBoxData.faceAttributeSet[face] != nil )
      Q3Object_Dispose(myBoxData.faceAttributeSet[face]);
  }

  return myBox;      // return the object; if nil, caller must handle
}
```

What happens if you later decide (after instantiating a geometric object) that you want to change the object's features, which are accessible only via the underlying data structure? QuickDraw 3D provides a rich set of data-access calls for retrieving and editing object data structures. This means it's possible not only to edit geometries (e.g., vertex coordinates) but to edit attributes (colors, surface normals, etc.) and -- in some cases -- topologies (the number of vertices), on the fly, during program operation. (If you're a non-OOP programmer who balks at having to use accessor functions to get at geometry structures, rest assured you can also keep raw data structures around and render them later with special Submit() calls, without ever creating an object. But the OOP way has certain advantages -- one being that you don't have to manage tons of persistent data structures.)

## Object Cloning

In Spinning Thang, we store our geometries in a *model*, which is simply a Group object containing several component objects. Our model happens to consist of a TQ3ShaderObject (or an "illumination shader"), four translation transform objects, and four copies of our geometry. The transforms move the copies of our objects to various positions in space. Why not just create four individual geometric objects, each one positioned in the "right place" at birth? That would certainly be the right thing to do if you wanted to animate the objects individually or let the user edit or manipulate them individually at runtime. But in 3D work, it often happens that you want to create groups of objects that "fly in formation," or move as a group. Maybe you want to create a meteor field containing 100 (or 1,000) like-sized rocks, or a giant wave of 8th Air Force bombers on a raid over Berlin. If each individual meteor (or bomber) consists of a 10,000-polygon mesh, it would be extremely wasteful of RAM to have to store each object individually, and it would require lots of CPU cycles to apply rotations, translations, etc. individually to each object. The smart way to do this is to clone one meteor (or bomber) multiple times, cutting storage and computational requirements drastically. The code in MyNewModel(), Listing 2, illustrates how to do this.

**Listing 2: MyNewModel()**

```
  MyNewModel()
Create and return a group object containing an illumination shader
and our cloned geometry. Notice that once the geometric object
is added to the group object, we dispose of its reference.

TQ3GroupObject MyNewModel(long geometry)
{
  TQ3GroupObject      myGroup = nil;
  TQ3GeometryObject    myGeometry;
  TQ3ShaderObject      myIlluminationShader ;
  TQ3Vector3D        translation;

  // Create a group for the complete model.
  if ((myGroup = Q3DisplayGroup_New()) != nil ) {


    // Define a shading type for the group
    // and add the shader to the group
    myIlluminationShader = Q3PhongIllumination_New();
    Q3Group_AddObject(myGroup, myIlluminationShader);
    if( myIlluminationShader )
      Q3Object_Dispose(myIlluminationShader);

    switch (geometry) {

      case BOX :
        myGeometry = MyNewBox( -0.5, -0.5, -0.5 );
        break;
      case CYLINDER :
        myGeometry = MyNewCylinder( -0.5, -0.5, -0.5 );
        break;
      case ELLIPSOID :
        myGeometry = MyNewEllipsoid( -0.5, -0.5, -0.5 );
        break;
      case TORUS :
        myGeometry = MyNewTorus( -0.5, -0.5, -0.5 );
        break;
      case CONE :
        myGeometry = MyNewCone( -0.5, -0.5, -0.5 );
        break;
      default:
        DebugStr("\pUndefined argument to MyNewModel.");

      }

    // put four copies of object into the group, each one with its own translation
    translation.x = -1;translation.y = 0;translation.z = 1;
    MyAddTransformedObjectToGroup( myGroup, myGeometry,
                                    &translation ) ;

    translation.x = 2;translation.y = 0;translation.z = 0;
    MyAddTransformedObjectToGroup( myGroup, myGeometry,
                                    &translation ) ;

    translation.x = 0;translation.y = 0;translation.z = -2;
    MyAddTransformedObjectToGroup( myGroup, myGeometry,
                                    &translation ) ;

    translation.x = -2;translation.y = 0;translation.z = 0;
    MyAddTransformedObjectToGroup( myGroup, myGeometry,
                                    &translation ) ;
  }

  if( myGeometry ) // once it's part of group, dump the reference to it
    Q3Object_Dispose( myGeometry );

  return ( myGroup );
}
```

Notice how each translation transform "picks up" from where the previous one left off, kind of like how ordinary QuickDraw starts drawing wherever the pen location was left at the end of the previous operation. Each time the model gets rendered, QD3D executes the transforms sequentially, in macro-like fashion, cloning our geometry. (This means, of course, that you can't color individual items in the group differently; the objects, after all, are *clones* of a single object.)

## Attribute Objects

An important design principle of QuickDraw 3D is that attributes (which, broadly speaking, means surface-related properties, like color and specularity) occur in AttributeSet objects, which are associated with geometries by means of their underlying data structures. Every kind of geometric object, from points and lines up to the meshes, has at least one TQ3AttributeSet data field in its data structure. This means that not only the base object but its constituent components (vertices, faces, etc.) have their own attribute sets, which apply in a natural hierarchical fashion so that the attribute set with the lowest-level association has precedence. For example, you can apply a solid color to a mesh object; but you could also apply colors to individual faces or vertices. Vertex colors, when present, "win out" over face colors, and face colors win out over whole-object colors, just as you'd expect.

The use of attributes in this hierarchical fashion achieves a couple of goals. First, it gives the programmer great flexibility, obviously. Secondly, it keeps storage requirements to the absolute minimum. If you want just one face of a multi-face object to be orange, and the rest of the object to be green, you can give the orange face one attribute set and the object itself another attribute set -- there's no need to apply unique attribute sets to each and every face or vertex. Not only does this reduce the amount of coding for the programmer, but it greatly eases the storage and computational overhead for QD3D.

There are presently eleven predefined attribute types in QD3DSet.h (and you can extend these with your own custom-defined attributes). They include diffuse color, specular color, transparency color, surface UV (parameterization), surface normal, and surface tangent, among others. Various object classes (View, Group, Geometric, Face, Vertex) have their own natural attribute types that can be applied to them; surface normals, for example, make sense for a polyhedral mesh but not for a View object. The relationships between classes and attributes are spelled out in detail in Chapter 5 of *3D Graphics Programming with QuickDraw 3D* (the official documentation for QD3D). For now, it's important merely that you know:

1. AttributeSets are *objects* that have to be created and disposed of like any other object.
2. An AttributeSet object can contain many different kinds of attributes, or none. (The attributes themselves are generally *not* objects: They're things like colors and vectors.)
3. Every geometric primitive at the level of Point or above has at least one AttributeSet field in its underlying data structure.
4. The AttributeSet field must be set to nil if it is not used. You don't have to use every AttributeSet field, but *you can't leave a garbage value in an unused AttributeSet field!*

Let's go back and look at the box again. The TQ3BoxData data structure has two AttributeSet fields: one for faces and one for the entire box. You can use one or the other, or both, or neither. In Spinning Thang, we apply different colors to the individual box faces. But suppose we simply want to color the entire box red. Here's how:

```
TQ3ColorRGB      color;

myBoxData.faceAttributeSet = nil;
myBoxData.boxAttributeSet = Q3AttributeSet_New();

color.r = 1.0;      // 100% red

color.g = color.b = 0.0;

Q3AttributeSet_Add( myBoxData.boxAttributeSet,
        kQ3AttributeTypeDiffuseColor,
        &color);

// be sure all the other data fields are filled out

...

// then create the box...
myBox = Q3Box_New( &myBoxData );

// dispose of no-longer-needed AttributeSet object...
if( myBoxData.boxAttributeSet != nil )
  Q3Object_Dispose(myBoxData.boxAttributeSet);
```

The box will now be solid red. Note that we have to follow a definite sequence: create the AttributeSet, then add our diffuse color to it, *then* create the box using our data structure, then dispose of the AttributeSet object. (Once the box is created, it has its own copy of the attributes.) It wouldn't make sense to dispose of our attributes before creating the box. Nor do we let the unused AttributeSet object persist after myBoxData goes out of scope.

Notice, incidentally, that QD3D uses an RGB color model, but a TQ3ColorRGB is not the same as a Color QuickDraw RGBColor. The latter uses three unsigned shorts. (If you want to convert between the two types, you'll have to write your own utility routine.)

By now, you're either totally confused, or some of this is actually starting to make sense -- in which case you just might be getting hooked. (Watch out.)

## Drawing

Once you have a geometry, actually displaying it requires that you create a View object, which is a collection of objects needed to render a scene, namely: camera, lights, draw context, and renderer. If you understand how geometric objects are created, the creation of cameras, lights, etc. will seem natural to you because the same overall methodology applies. Only the underlying data structures will be unfamiliar (although if you've spent any time using commercial 3D packages, many of the concepts should ring a bell).

One object that may be rather unfamiliar-sounding at first is the DrawContext. This is simply QD3D's analog of the GrafPort (a place to draw), except that it is designed to be much more general. QuickDraw 3D is a cross-platform library, which means (for example) that it needs to be able to draw into more than one kind of windowing system. The rest of the API needs to be insulated from the machine-specific implementation details of drawing into various kinds of windows. The DrawContext object layer achieves that.

When QD3D is running under the MacOS, it can use two specific kinds of draw contexts: a so-called Macintosh draw context, or a pixmap (or offscreen) draw context. The former lets QD3D draw into CGrafPorts while the latter lets QD3D draw into GWorlds or memory. (You don't have to use pixmaps associated with GWorlds, incidentally: You can force QD3D to draw straight into any arbitrary address in memory, if you want.) Why would you want to draw into memory yourself, when QD3D already handles double-buffering for you? Well, for one thing, you might want to be able to output your renderings as PICT files. You might want to put PICTs on the scrap, or insert them directly into a QuickTime movie. You might want to apply 2D convolutions or filters to your renderings. Or you might want to combine images offscreen to achieve special compositing effects, such as underlays or overlays. There are lots of possibilities. (Maybe we'll explore some of them in future articles.)

When drawing into a CGrafPort, it's not necessary to give QD3D control over the entire window. In Spinning Thang, we show how to make drawing occur in a specified portion of a window (a "pane") -- a technique that comes in handy when you need to draw inside dialogs or alerts, for example. (We also show how to set the "clear color" -- or background color -- to whatever color you want it to be.) The source code is pretty self-explanatory, except maybe for one thing: When drawing to a restricted portion (or pane) inside a window, you want to be doubly sure to alert QD3D to the height/width ratio of the pane; otherwise your objects could look stretched or squashed. The solution is simple: When setting up your camera data, set the aspectRatioXToY field of the TQ3ViewAngleAspectCameraData structure to reflect the *aspect ratio of the pane, not the overall window*.

Space doesn't permit an exhaustive discussion here of the details of setting up camera and light objects. For an excellent discussion of the basic techniques, see Brian Greenstone's introduction to QD3D in Chapter 9 of *Tricks of the Mac Game Programming Gurus* (Hayden Books).

## Rendering

So, how do you draw something in QD3D? The answer is, you set up a rendering loop. This is another example of an area where the QD3D way of doing things often catches beginners off-guard, so let's spend a moment talking about it.

In QD3D, each element of a scene must be "submitted" to the renderer, in a particular order. Style and shader objects have to be submitted before geometries, for example, because QD3D needs to know how to properly represent visible objects before they can be drawn. Rendering is done in a do-while loop, to allow for the possibility that rendering might have to be done in passes. (This really isn't a concern when using Apple's renderer, which finishes in one pass, but a raytracing or radiosity renderer could, in theory, require multiple passes.) The screen-drawing function from Spinning Thang is shown in its entirety in Listing 3.

**Listing 3: DocumentDraw()**

```
DocumentDraw()
This is the main drawing routine in our sample app.

TQ3Status DocumentDraw( DocumentPtr theDocument )
{
  TQ3Status status;

  status = Q3View_StartRendering(theDocument->fView );
  if (status != kQ3Success)
    DebugStr("\pTrouble with StartRendering().");

  do {

    Q3Style_Submit( theDocument->fInterpolation,
            theDocument->fView );
    Q3Style_Submit( theDocument->fBackFacing,
            theDocument->fView );
    Q3Style_Submit( theDocument->fFillStyle,
            theDocument->fView );
    Q3MatrixTransform_Submit( &theDocument->fRotation,
            theDocument->fView );
    Q3DisplayGroup_Submit( theDocument->fModel,
            theDocument->fView );

  } while (Q3View_EndRendering(theDocument->fView) ==             kQ3ViewStatusRetraverse );
  return kQ3Success ;
}
```

Rendering begins with a call to Q3View_StartRendering(), which tells QD3D to prepare for drawing using our View object. (Note that in Spinning Thang, all of our persistent scene objects are conveniently stored in a single application-defined data structure, the DocumentRec; see below.) One by one, we submit our Style objects (the exact order of these isn't important); then a transform matrix to perform our (global) rotations; and finally our geometry. As we make these "submit" calls, QD3D constructs the scene offscreen (assuming double-buffering is enabled in our DrawContext); then when Q3View_EndRendering() is called, the image, if it's ready, is blitted to the screen. If it's not ready, the loop cycles.

```
struct _documentRecord {
  TQ3ViewObject  fView ;           // the view for the scene
  TQ3GroupObject  fModel ;         // object in the scene being modelled
  TQ3StyleObject  fInterpolation ;  // interpolation style used when rendering
  TQ3StyleObject  fBackFacing ;     // backface removal
  TQ3StyleObject  fFillStyle ;     // draw as solid filled object?
  TQ3Matrix4x4  fRotation;         // the rotate transform for the model
};
typedef struct _documentRecord DocumentRec, *DocumentPtr, **DocumentHdl ;
```

Note that somewhere in all this, there needs to be an Illumination Shader object, to set the renderer's illumination model to Lambert, Phong, or Null. (Consult a graphics text for mathematical details of these methods. Phong is essentially Lambert shading with specular highlights added. Null is an option that lets you shade objects uniformly, without respect to light direction.) In Spinning Thang, the illumination shader is attached to the model (the group object that contains our geometry). But it doesn't have to be. It could be submitted individually in the rendering loop.

Any number of additional objects (geometries, transforms, group objects containing combinations of other objects, etc.) can be submitted in the rendering loop. It all depends on what you need to draw.

## Looking to the Future

We've covered a lot of ground in just a few pages, but even so, we've really only begun to scratch the surface of QD3D. Thoroughly exploring QuickDraw 3D would require volumes. We haven't even touched on such important topics as texture mapping, freeform mesh geometries, file objects, or the 3D Metafile Format, let alone user-interface issues. But by now, you should have some appreciation for the power and scope of QuickDraw 3D. With a little imagination, QD3D can take you to some pretty interesting places. Imagine putting custom 3D controls in your dialog boxes; doing 3D camera fly-throughs of 3D-plotted data; connecting QD3D's Pointing Device Manager to a MIDI controller and using QuickTime's MIDI-track facilities to create, edit, record, and play back intricate, gestural animations; modelling a new gadget in QD3D and outputting it in QuickTimeVR format for others to play with; or showing a QuickTime movie on the surfaces of a spinning cube. (Code for the last two tasks can be found on Apple's web site.) The mixing of QuickTime and QD3D capabilities is an especially exciting prospect for animators. And to think, you can harness all of this magic *and still produce cross-platform code.*

No matter what aspect of the 3D graphics programming challenge you want to tackle, QuickDraw 3D has a variety of tools that can help you realize your dream. How far you take it is up to you.

---

**Kas Thomas**, tbo@earthlink.net has been a Macintosh user since 1984 and has been programming in C on the Mac since 1989. He wrote two of the ten most downloaded Photoshop® plug-ins on America On-Line and holds U.S. Patent No. 5,229,768 for a high-speed data compression algorithm. His current project involves a QD3D-powered Photoshop® plug-in, code-named Callisto.