Volume Number: 14
Issue Number: 10
Column Tag: Power Graphics

Poor Man's Bryce (Part I): Terrain Generation in Quickdraw 3D

by Kas Thomas

Displacement mapping is easy to do in QD3D and is a good way to learn about freeform mesh geometries

Introduction

In a previous article (MacTech, July 1998), we saw how Apple's cross-platform Quickdraw 3D API - which is now part of QuickTime 3.0 - gives programmers easy access to a comprehensive set of highly optimized 3D graphics routines. With the aid of QD3D, you no longer need to be a math whiz or a graphics professional to put world-class 3D graphics onscreen and give your end user a powerful interactive 3D experience. What ordinary Quickdraw did for 2D graphics, Quickdraw 3D does for 3D graphics - and it does it in cross-platform fashion. (QD3D code ports easily to Windows, because the "device layer" has been abstracted out in elegant fashion.)

This month we're going to show how QD3D's freeform Mesh primitives can be used to visualize 2D data three-dimensionally. In particular, we're going to take a crack at *terrain generation* - making topographic grids on the fly, based on 2D input. Our sample app, PMB (which stands for "Poor Man's Bryce"), will let you convert any PICT, TIFF, GIF, JPEG, or Photoshop 3 image into a fully shaded three-dimensional terrain grid that can be rotated, scaled, or "flown through" like some of the NASA graphics of the surface of Mars that you may have seen on TV. The basic technique we're going to use is displacement mapping, which is a popular method of getting 3D effects from 2D input (used in many high-end 3D animation and graphics packages).

In this article, we'll show how to set up the core geometry, do the elevation mapping, and accomplish some basic colorizing. In followup articles, we'll talk about how to do texture mapping, vertex-normal recalculation (to eliminate faceting), and various tricks for making terrains look more "realistic." Along the way, we'll discuss speed issues (always a concern in 3D work), human interface considerations, and a zoo of other topics. In the process, we'll learn a lot more about Quickdraw 3D.

There's plenty to do. Let's start by talking about mesh geometries.

Major Freeform Geometries

In Quickdraw 3D, there are currently four major freeform primitives to choose from: the Mesh, TriGrid, TriMesh, and Polyhedron. Of these, the Mesh and TriGrid have been around since QD3D version 1.0; the others came along in version 1.5. The details of working with the four mesh geometries were discussed in a develop #28 article by Philip J. Schneider ("New Quickdraw 3D Geometries"); I won't rehash any of that material here. If you're interested in knowing more about the mesh geometries, you should definitely consult Schneider's article as well as Chapter 4 of *3D Graphics Programming With Quickdraw 3D* (the official documentation for QD3D, available online at Apple's web site). In the meantime, it might be a good idea for us to review some of the basic design issues associated with the four major mesh types before going on, since the choice of geometry will have tremendous ramifications later, when we get into our "terrain modelling" project. (Choosing the right geometry is essential in this as in any 3D programming project.)

Mesh

In the beginning, there was the Mesh. This geometry was designed from the outset to be (and remains, today) the most powerful and flexible of QD3D's freeform geometries. It's also the only geometry that has no explicit data structure: If you look through the QD3DGeometry.h header file, you'll find no TQ3MeshData struct or typedef. The structure is private, known only to QD3D's internals. How do you create one, then? The trivial answer is: You call Q3Mesh_New(). That will give you an empty Mesh. You go about building the Mesh by adding vertices and faces (lists of vertices comprising polygons in the Mesh) using functions Q3Mesh_VertexNew() and Q3Mesh_FaceNew().

The key thing to note is that the Mesh was designed for iterative construction and editing. That is to say, you construct it bit by bit; and you can edit it, on the fly, bit by bit. Thus, you'll want to use the Mesh if you intend to give your user realtime, interactive geometry-editing capability (as in a modelling program). This is not to say that QD3D's other mesh types can never be used in an interactive editing scenario; it's just a lot less convenient to use the other freeform geometries this way, because you typically have to retrieve and reorder large vertex arrays, which may involve memory reallocation and a host of bookkeeping concerns. With the Mesh, you don't have to worry about any of this because QD3D takes care of everything behind-the-scenes, so to speak. All you have to do is tell QD3D which Mesh parts you want to work on (the API provides literally dozens of part-editing and traversal routines) and QD3D takes care of memory allocation, reordering of edges, clockwise vertex numbering, etc. (And incidentally, the polygons that comprise a Mesh's faces do not have to be triangles - they can have any number of points. They don't even have to be *planar*, strictly speaking, although you'll probably get rendering artifacts if you don't at least ensure that your Mesh faces are planar).

The drawbacks of the Mesh are worth noting. Largely because it is such an immensely powerful, flexible, easy to use primitive, the Mesh is extremely "greedy" when it comes to memory usage. The reason for this is that in order to enable the Mesh's powerful topological editing and traversal functions, a Mesh's parts (vertices, faces, edges, contours, and "mesh component" groupings) have to contain pointers not only to their own building blocks, but also to the parental components of which they are parts. In a large, complex Mesh (think of a Jurassic Park T. rex), the connectivity information can take up as much as 75% of the memory required to store the model. This not only means poor I/O performance but a big rendering speed hit, too, since the entire structure has to be decomposed in orderly fashion at render time.

Fortunately, QD3D provides a couple of more streamlined freeform geometries that you can use for representing large models efficiently. The Mesh remains a useful primitive for many purposes - it's still the best one to use for realtime interactive editing - but when fast rendering is a priority, it's usually best to start out with a different primitive (or translate data back and forth between the Mesh and other primitives at edit time versus render time).

TriGrid

The TriGrid was included in Quickdraw 3D version 1.0 in order to give programmers a kind of "hot rod" version of the Mesh: i.e., a freeform primitive suitable for representing large, complex models, but capable of efficient memory usage and fast rendering. Unlike the Mesh, the TriGrid has a straightforward public data structure:

```
typedef struct TQ3TriGridData {

   unsigned long       numRows;
   unsigned long       numColumns;
   TQ3Vertex3D         *vertices;
   TQ3AttributeSet     *faceAttributeSet;
   TQ3AttributeSet     *TriGridAttributeSet;

} TQ3TriGridData;
```

To create a TriGrid, you fill in the fields of the data structure, then pass a pointer to the structure to Q3TriGrid_New().

The TriGrid has an inherently rectilinear topology, as you can see from the first two fields in the TQ3TriGridData data structure. (The one-dimensional vertices array is analogous to the baseAddr field in a PixMap.) So if, for example, you know that your 3D model can be built with 400 vertices in a 20-by-20 lattice, it probably makes sense to implement the model as a TriGrid.

At first glance, the TriGrid might not seem particularly flexible, since it doesn't allow rows or columns with arbitrary numbers of points. The TriGrid is obviously perfect for flags and billboards. But what else can you do with it? Actually, you can implement a surprising number of objects with TriGrids, including spheres, cylinders, cones, tori, and helices (plus lots more). The key is to remember that although there has to be an equal number of points in each row of the grid, the points don't have to be equally spaced. (They can even overlap.) You'd be surprised how many shapes you can wrap a TriGrid around.

We'll get deeper into the subject of attributes later, but for now suffice it to say that the TriGrid (like most QD3D objects) can have its own attribute set; and the faces can have separate attributes of their own. (Although it's not obvious, the vertices in a TriGrid can also have *their* own attributes.)

The fixed topology of the TriGrid means that interactive insertion/deletion of points or faces is (for all intents) impossible. But there are compensating virtues. First of all, Quickdraw 3D implements the TriGrid as strips of triangles, which means that rendering is extremely fast and there is never any concern about "tears" or rendering artifacts caused by nonplanar polygons. Secondly, the triangles share vertices in an optimal manner, cutting storage requirements and speeding up I/O. (Note that in a large TriGrid, there is effectively a 1:1 ratio of vertices to triangles.) A third advantage of the TriGrid topology, which may not be of particularly obvious importance at the moment, is that it lends itself to easy UV parametrization. (UV parametrization is a technique for mapping 2D surfaces or functions onto "three-space." We'll talk more about this when we get into texture-mapping.)

The TriGrid may not be as flexible as the Mesh, but what it lacks in flexibility it makes up for in efficiency. Good rendering speed, efficient memory use, and fast I/O characterize the TriGrid. When a model lends itself to implementation as a TriGrid, you can't go wrong choosing this primitive.

TriMesh

The TriMesh, which made its debut in QD3D version 1.5, is one of the most widely used freeform primitives (because it's generally conceded to be the fastest). It's also the most controversial. Many QD3D programmers disdain the TriMesh, however, because it results (in many instances) in code that's bloated and/or hard to read and maintain. The reason is simple. The TriMesh was designed from the beginning to be a low-level, high performance freeform primitive, optimized for rendering speed, not for programmer convenience. It's the SR-71 of Mesh primitives: costly to build and maintain, but faster than anything else out there.

Similar to the TriGrid (but unlike the Mesh), the TriMesh has an explicit public data structure and is created by passing a pointer to the data structure to Q3TriMesh_New(). The data structure looks like this:

```
typedef struct TQ3TriMeshData {

   TQ3AttributeSet          TriMeshAttributeSet;

   unsigned long         numTriangles;
   TQ3TriMeshTriangleData      *triangles;

   unsigned long         numTriangleAttributeTypes;
   TQ3TriMeshAttributeData   *triangleAttributeTypes;

   unsigned long         numEdges;
   TQ3TriMeshEdgeData          *edges;

   unsigned long         numEdgeAttributeTypes;
   TQ3TriMeshAttributeData   *edgeAttributeTypes;

   unsigned long         numPoints;
   TQ3Point3D                 *points;

   unsigned long         numVertexAttributeTypes;
   TQ3TriMeshAttributeData   *vertexAttributeTypes;

   TQ3BoundingBox             bBox;

} TQ3TriMeshData;
```

The TQ3TriMeshData structure consists of an attribute set for the overall TriMesh; five parallel arrays describing the mutual relationships of the object's points, edges, faces (triangles), and attributes; and a bounding box. The precalculated bounding box helps speed up rendering. The parallel arrays work more or less as you'd expect; some of them (the edges and attribute arrays) are optional. As with the TriGrid, the underlying polygon type here is the triangle, but whereas the vertex/triangle memberships are implicit in the TriGrid layout, those relationships are explicit in the case of the TriMesh. They have to be, because the TriMesh lets you specify any number of points in completely arbitrary order.

Why is the TriMesh so disdained by some (but not all) QD3D programmers? First, there's the uniform-attributes requirement. This means that if you want one of the faces in your TriMesh to have a transparency attribute, they all have to have a transparency attribute. (The same holds true for vertex and edge attributes.) This is counter to the way attributes are handled in all other Quickdraw 3D primitives. Another awkward feature of the TriMesh is that attributes are not closely linked to the actual objects they apply to. (Again, this runs counter to the rest of the API.) If you want to change the vertex normal for, say vertex number 99, you have to retrieve the 99th element of the array of vertex normals. The vertex normals might, in turn, be just one subarray of the vertexAttributeTypes (which may very well hold arrays, also, for diffuse color and transparency). Adding new attribute types at runtime, or enabling interactive editing of geometry, can require memory reallocation and bookkeeping; and the risk of explosive data growth at runtime is significant because of the uniform-attributes requirement. (In a 10,000-polygon *TriGrid* with faces set to red, white, or blue, all you have to do is set each attribute pointer to one of three addresses. The same model implemented as a *TriMesh* would require 12 5 10,000 = 120,000 bytes of extra array storage. Want the blue faces to be transparent? Add another 120,000 bytes. Want one of the red faces to have an amber specular highlight? Add 120,000 bytes. And so on.)

Another down side to working with the TriMesh is that there are no accessor functions to aid in the editing of points, attributes, etc. For example, to retrieve the coordinates of vertex No. 298 from a *Polyhedron* (see below) you just call Q3Polyhedron_GetVertexPosition(Polyhedron, 298, &pt). This is true of most of QD3D's freeform primitives; you can "peek" and "poke" individual vertices quite easily with special editing functions provided for just that purpose. With a TriMesh, you have to retrieve the entire TriMesh (which might contains thousands of vertices) and index into the points array, then free up the memory when you're done.

If you're one of those programmers who would kill for an extra CPU cycle, you'll probably want to get to know the TriMesh very well, because it's anywhere from two to ten times faster than other Mesh primitives. If, on the other hand, elegant, readable code and efficient memory use are more important to you than raw speed, you'll want to consider the Mesh, TriGrid, or Polyhedron.

Polyhedron

The Polyhedron is generally judged to be the second-fastest Mesh primitive. You can think of it as a TriMesh in a tuxedo. The data is contained in a TQ3PolyhedronData structure:

```
typedef struct TQ3PolyhedronData {

   unsigned long                numVertices;
   TQ3Vertex3D                  *vertices;
   unsigned long                numEdges;
   TQ3PolyhedronEdgeData        *edges;
   unsigned long                numTriangles;
   TQ3PolyhedronTriangleData    triangles;
   TQ3AttributeSet              PolyhedronAttributeSet;

} TQ3PolyhedronData;
```

There are no surprises in the Polyhedron data structure: It uses a triangle list (and, optionally, an edge list) defined by indexes into a vertex list, with an optional attribute set for the entire Polyhedron. Where are all the vertex and face attributes? Vertex attribute sets are stored in the individual vertex data structures, and triangle attributes are stored in the TQ3PolyhedronTriangleData structures. In other words, the attributes are attached to the objects they modify -a safe and sane approach. And you can set any of them (or all of them, or none of them) to NIL.

In many respects, the Polyhedron is just a TriMesh, but with attributes stored differently (some would say properly). The Polyhedron lacks a bounding box field and for this reason (and others) renders a bit slower than the TriMesh. But the sharing of vertices and efficient use of attributes makes for snappy I/O and rendering. Also, editing is easy because the API provides a rich set of accessor functions for getting at individual triangles, edges, vertices, and attributes. As a result, according to Apple, "The polyhedral primitive is the primitive of choice for the vast majority of programming situations and for the creation and distribution of model files if editing of models is desired." (See develop #28, p. 53.)

Choosing a Terrain Primitive

If you've come this far, you may already have a good idea of which Mesh primitive we'll be using for our terrain-generation project. The goal, you'll recall, is to generate an elevation-mapped 3D terrain grid from 2D input. The simple Cartesian (rows-and-columns based), lattice-type layout of a bitmapped image is well suited to -you guessed it -the TriGrid primitive. The main task is to generate a TriGrid with rows and columns equal (or at least proportional) in number to the pixel dimensions of the source image (which we can get from the bounds Rect of the PixMap). A piece of cake, right?

Well, almost. Two potential problem areas we should address up front are storage and performance. Will we have enough memory for the TriGrid? If we use a 1:1 mapping of pixels to vertices, we could run into trouble. After all, if we let the user supply any old PICT image, and he just *happens* to choose a screen shot of his 24-inch Sony W900 monitor (with 1920 x 1200 resolution), we could be talking about an image with *well over a two million 32-bit pixels*. The corresponding TriGrid would contain a whopping *two million triangles*. By anyone's standard, that's a pretty large 3D object. (Jim Cameron's Titanic was 2.5 million polygons.) Even assuming there is enough system RAM to accommodate everything (in this example, two million vertices equates to 32 million bytes), the render speed of such a large grid would be -how shall we say? -a trifle slow. Without special hardware it could easily take 60 seconds or more to do a screen redraw of a two-million-triangle grid on a G3 machine. If you didn't turn double buffering off, you'd see nothing happen for 60 seconds. You might think the machine hung. That's not what most users want.

The bottom line is that it *isn't* okay just to preflight RAM and run with whatever fits in memory. Performance is a key issue in graphics software. A good tactic is probably to preflight the user's RAM *and* silently benchmark his system, before deciding on a maximum TriGrid resolution. If the TriGrid won't fit in memory or won't render fast enough, the logical thing to do is downsample the source image (and make a smaller TriGrid).

Our sample app will take a very simple approach: In the interest of performance, we'll just limit TriGrids to 4,096 vertices, max. That means any source image with fewer than 4,096 pixels is okay. Anything bigger will result in automatic subsampling. We'll lose some topographic detail if we subsample, but that's the price we pay to keep screen updates snappy-feeling. Of course, you should experiment with setting MAX_VERTEX_COUNT to other values. I chose 4,096 because that happens to keep screen updates to half a second or so on my 250Mhz G3 machine. If you're innately patient or have special hardware, you may want to try higher values, but don't be surprised if a 10,000-vertex grid bogs your machine down unbearably, particularly if you're not running at least a G3 Mac.

The PMB Project

The CodeWarrior project for our PMB (Poor Man's Bryce) app is made up of four C modules - PMBMain.c, PMBSupport.c, Image.c, and Terrain.c -which aggregately comprise about 2,000 lines of code. If you followed our introductory article on Quickdraw 3D programming (MacTech, July), the code should look pretty familiar. It follows the same format of keeping persistent data in a custom data structure (the DocumentRec, described in PMBShell.h) and keeping most of the QD3D setup code in PMBSupport.c. The code is a bit too lengthy to reproduce in full here, but you can get the full project on the MacTech web site at http://www.mactech.com.

One aspect of the project that bears discussing is our memory management strategy. This can be very critical in any program that works with large images and/or large 3D objects. You'll notice (if you look at the compiled app) that PMB runs comfortably in one megabyte of RAM -even though it can handle images with millions of 8-, 16-, or 32-bit pixels. How is this possible? The short answer is, we use Temporary Memory whenever possible. This, and the fact that QD3D runs in System memory, keeps us out of trouble. It may seem like a copout to use Temporary Memory like this -after all, aren't we just transferring our memory requirements to a different spot in RAM? It still takes a large amount of RAM to work with large files, doesn't it? Yes, it does. But the application itself doesn't need to keep large objects in memory; once it's done opening, reading, downsampling, and elevation-mapping a big image, it just tosses the PixMap out the window. At that point, it makes no sense to let our app hog a ten-megabyte heap when one megabyte will do. (Suppose you open a large image with PMB, then - while PMB is running - you decide you want to edit the same image in Photoshop. If the two programs are competing for RAM, you'll have to shut one program down while you use the other one. Wouldn't it be better to keep both programs running, and open/close files as needed? The point is: When files contend for RAM, you can selectively close and reopen them. No big deal. When programs contend for RAM, it's a much bigger deal because you have to quit and relaunch them - which can be very time-consuming, for a complex program like Photoshop.)

It turns out that PMB's biggest use of application-heap memory comes not in working with source imagery but in creating 3D vertices for use in the TriGrid. (See below.) Image I/O in PMB is handled for us by QuickTime's graphics importers. (If you search the project source files for File Manager calls, you won't find any!) This lets us handle a variety of image formats -TIFF, GIF, JPEG, PICT, and Photoshop 3 - with ease, while shifting the burden of I/O buffering to the OS (i.e., System heap).

Once a source image has been opened for us, we look at its native dimensions (the bounds Rect) to see how many vertices we're going to have to create in Quickdraw 3D. If the number of pixels in the image is greater than MAX_VERTEX_COUNT, we resort to the cheesy but effective trick of downsampling the image by *making QuickTime draw it into a smaller destination Rect*. (This is kind of like letting CopyBits "dither down" a too-big picture file to fit a small onscreen window - something we've all done at one time or another.) We actually make QT draw into a downsized offscreen GWorld, then send the locked-down PixMap to our TriGrid routines. After the grid is made, we unlock the PixMap and free the GWorld.

The TriGrid gets created in three steps. In MyNewModel(), we call on three routines in quick succession: TriGridBuild(), TriGridConvertToTerrain(), and TriGridColorize(). The first routine simply constructs a flat (planar) grid in x and y. Why not x and z? After all, isn't the y-dimension usually reserved for up-and-down operations? Actually, it's totally arbitrary. I chose to build the grid in x-y, and add elevation data in z, simply to make it easier to visualize the column/row-oriented x-y operations that we'll be discussing later on. It could just as easily have been done the other way.

Once the (flat) grid is built, we give it altitude information - 3D shape -by looping over our image pixel data and mapping pixel intensities to the z-axis of the grid, vertex by vertex. This is the purpose of TriGridConvertToTerrain(), which we'll discuss is greater detail in a minute. Finally, once the (gray, attribute-less) grid has been constructed, we go back and colorize it, in TriGridColorize(), again using pixel information from our image PixMap. At that point, the terrain is complete and we are able to dispose of the (offscreen) image information.

Let's go over some of these routines in a little more detail so that you can see how they work. It's all very straightforward and works just about as you'd expect, with only a couple of minor twists here and there.

TriGrid Creation

The code for creating the TriGrid is simple. Once we know (from the source image bounds rectangle) what the dimensions of the grid are going to be, we just call our TriGridBuild() function (Listing 1) with appropriate column and row arguments.

TriGridBuild()is very general in nature and is typical of QD3D geometry creation code in that a data structure describing the geometry is first filled out, then passed to a Geometry_New() type of function. Since the grid will initially be colorless, we set all attribute fields to nil. (It's important to "nil out" these fields because if you leave them uninitialized, or set to garbage values, your app will almost certainly crash.) We then grab some memory for the grid's vertices, loop over the rows and columns, and fill in the vertex coordinates, do some cleanups, and return the object to the caller.

Listing 1: TriGridBuild()

```
TriGridBuild()
```

This is where we instantiate our terrain grid's geometry. The code shown here is very general and can be used to create TriGrids for any kind of project.

```
TQ3GeometryObject TriGridBuild(long cols, long rows)
{
   TQ3TriGridData           TriGridData;
   TQ3GeometryObject        TriGrid = nil;
   unsigned long            i,j,k;
   float                    dx,dy, aspect;
   extern DocumentRec       gDocument;
   OSErr                    err;
   Handle                   vertsStorageHandle;


   TriGridData.facetAttributeSet        = nil;
   TriGridData.TriGridAttributeSet      = nil;
   TriGridData.numRows                    = cols;
   TriGridData.numColumns                 = rows;

   // now we need to allocate storage for our vertices ( inTemporary Memory)

   vertsStorageHandle =
      TempNewHandle(cols* rows * sizeof(TQ3Vertex3D),&err);

   if (vertsStorageHandle == nil || err != 0)
      DebugStr("\pNot enough RAM for grid vertices.");


   HLock((Handle) vertsStorageHandle);  // lock it down

   TriGridData.vertices = *(TQ3Vertex3D **)vertsStorageHandle;

   aspect = (float)rows/(float)cols;

   k = 0;

   // loop over all columns...
   for (i = 0, dx =  GRID_SIZE * -0.5;
        i < cols;
        dx += GRID_SIZE/(float)(cols-1), i++)

          // ...and all rows
          for (j = 0, dy = aspect * GRID_SIZE * -0.5;
               j < rows;
               dy += aspect * GRID_SIZE/(float)(rows-1), j++) {

          TriGridData.vertices[k].point.x = dx;
          TriGridData.vertices[k].point.y = dy;
          TriGridData.vertices[k].point.z = 0.; // build a flat plane
          TriGridData.vertices[k].attributeSet = NULL;

          k++;

          }

   TriGrid = Q3TriGrid_New(&TriGridData);

   HUnlock((Handle)vertsStorageHandle);    // unlock the handle and...
                                           // free up the memory...
   DisposHandle((char **)vertsStorageHandle);

   return (TriGrid);                       // return the geometry
}
```

Note that because TriGridBuild() has no idea, beforehand, how big the grid will actually be, it can't preallocate local array storage for the grid's vertices. Many of the more trivial QD3D example-code snippets available online show vertices, edges, faces, etc. already stored in preinitialized arrays. That's not how the real world works, of course. In the real world, you don't know in advance how big an object might be. As a result, dynamic memory allocation is the rule. In our case, we need to obtain a handle to Temporary Memory, so we use TempNewHandle() to get vertex memory. (This memory block is freed at the end of the routine.)

To set the vertex coordinates, we use a nested double loop that loops over columns and rows. The constant GRID_SIZE (#defined in Terrain.h) is arbitrary and sets the overall width of the grid in 3-space. Our TriGrid has cols vertices in the width direction and cols-1 intervals connecting them horizontally, hence the spatial increment (in x) between vertices is GRID_SIZE/cols-1. If we start our first point at x = GRID_SIZE * -0.5, we will end up with a grid that is centered on the x-axis. (This looks better than having it start at zero and extend in one direction. When we're done, we want the grid to rotate about its center, not about one corner.) Similar code applies for the y-direction (rows). Note that we include an aspect-ratio adjustment in y, since if we don't, we'll end up with a square grid. We want the grid aspect ratio to match the source-image aspect ratio. If the image is short and wide, we want the grid to be short and wide, right?

When the TriGrid's data structure has been completely filled out, we call Q3TriGrid_New() with a pointer to the data; this creates our TriGrid object. After disposing of vertex memory, we return the TriGrid to the calling routine.

Adding Height Information

Now that we have a flat grid, we need to give it elevation information. This is really the crux of terrain mapping. The function TriGridConvertToTerrain() is where the main magic happens. (See Listing 2.) This function takes a PixMapHandle and a TriGrid as arguments and loops over the source image's pixel values to obtain height information for our grid.

The code in Listing 2 shows a typical "geometry editing" operation in Quickdraw 3D. Notice that we have to call Q3TriGrid_GetData() in order to edit the underlying geometry of our object. Until we make this call, the geometry (the vertex coordinate data) remains opaque -hidden to our application. After the "get data" call, we have access to all of the TriGrid's internals. We can edit vertex coordinates, add attribute sets, or even create an all-new grid from scratch. But the changes won't take place until we call Q3TriGrid_SetData().

One very important thing to note about Listing 2 is that when we call Q3TriGrid_GetData(), Quickdraw 3D *allocates memory that we later have to release if we want to avoid memory leaks*. That's because Quickdraw 3D isn't just handing us a data structure with a few pointers in it. It's giving us the TQ3TriGridData structureand all the vertex data that goes along with it. The bottom line is that if we don't make a call to Q3TriGrid_EmptyData() when we're all done, we'll have a bunch of no-longer-needed vertex data sitting around in our heap doing nothing. That could be a very significant amount of data -enough to cause bigtime memory leakage.

Listing 2: TriGridConvertToTerrain()

```
TriGridConvertToTerrain()
```

Note: The PixMap and TriGrid should be equal in size (i.e., the PixMap's bounds Rect should match the TriGrid's row and column dimensions). There is no sanity check for this inside the following routine. Caller must assure sanity.

```
void   TriGridConvertToTerrain(
      PixMapHandle pix,
      TQ3GeometryObject TriGrid)
{
   TQ3TriGridData       data;
   RGBColor             rgb;
   long                 columns,rows,i,k,n;
   extern void GetColorPixel(PixMapHandle w,
                                long x,
                                long y,
                                RGBColor *rgb );
   extern float PixelLuminance( RGBColor rgb );

   // get grid's data...
   if (Q3TriGrid_GetData( TriGrid, &data ) != kQ3Success)
      DebugStr("\pData can't be fetched from TriGrid.");

   for (i = n = 0; i < data.numRows; i++)    // loop over all rows in grid
      for (k = 0; k < data.numColumns; k++) { // and all columns

         GetColorPixel( pix, i, k, &rgb ); // fetch pixel value
                                             // use it to set z-value
         data.vertices[ n++ ].point.z =
            -HEIGHT_MAX * PixelLuminance( rgb );
      }

   Q3TriGrid_SetData( TriGrid, &data );     // set new data to object
   Q3TriGrid_EmptyData( &data );            // dispose of copied data
}
```

If it wasn't obvious before, I should point out now that a vertex, in Quickdraw 3D, consists of three floats (describing x,y,z coordinates)and a pointer to an attribute set. The data structure looks like this:

```
typedef struct TQ3Vertex3D {

   TQ3Point3D      point;
   TQ3AttributeSet   attributeSet attributeSet;

} TQ3Vertex3D;
```

To set new coordinates for a vertex, in Listing 2, we index into our vertex array, then index into the TQ3Point3D field and the respective coordinate; thus the notation data.vertices[n++].point.z, which (in English) means "the z value of the point represented by vertex 'n' of the TriGrid data." The constant HEIGHT_MAX in Listing 2 is a purely arbitrary number that allows us to give our elevation map an initial maximum elevation. The value here doesn't matter a great deal, because our program gives the user the ability to interactively scale the object in the height axis at runtime by pressing Shift and '+' or Shift and '-'. HEIGHT_MAX just gets us in the right ballpark to start.

The exact value of 'z' (our height dimension) is determined by the *luminance* of the pixel at i,k. The essential concept here is that the height of a given vertex in the terrain grid should be proportional to the *perceived brightness* (rather than the color) of the corresponding pixel in the source image. First, of course, we have to fetch the RGBColor of the pixel at i,k. (To do this, we use some custom-written color-fetching routines that are capable of handling 8-, 16-, or 32-bit pixel data; see the source code in the project for details.) But how do we convert this quadruplet of unsigned shorts (which is what an RGBColor is) to brightness? The answer is that first, we convert the RGBColor to a Quickdraw 3D TQ3ColorRGB (three floats). Then we convert the color to a luminance value with the QD3D utility routine Q3ColorRGB_Luminance(). The function that QD3D uses internally is:

```
luminance  =
(0.30078125 ¥ color.r) + (0.58984375 ¥ color.g) +
(0.109375 ¥ color.b)
```

which is based on the fact that the human eye is not equally sensitive to all colors. You'll find various color decomposition formulas in various textbooks. The coefficients vary from one system to the next based on assumptions about device gammas and such, but the important thing to remember is that each human being has a unique visual cortex and a unique "device gamma" that can't be accurately compensated, necessarily, by a formula in a book. So don't get hung up on the "luminance vector" values. In a pinch, you can just add the red, green, and blue values together to get a rough estimate of luminance.

Note that once again, when we're done modifying the TriGrid data, we call Q3TriGrid_SetData() to update the geometry, and Q3TriGrid_EmptyData() to free up the memory that QD3D allocated for us when we asked to retrieve the TriGrid's data.

Colorizing the Grid

At this point, we have a "terrain grid" in three dimensions, but it's totally colorless and will look pretty boring if we let Quickdraw 3D render it as-is. Wouldn't it be nice to give it some color? Giving the grid a single uniform color is very easy (and requires very little storage), because the TQ3TriGridData structure contains a TriGridAttributeSet field for just this purpose. But, we're going to go the extra step to make our grid look a little prettier still. Rather than just give the grid one overall color, we're going to map the colors of the original source image onto the terrain grid. This is done in our TriGridColorize() routine, in

Listing 3.

Listing 3: TriGridColorize()

```
TriGridColorize()
```

Note that the PixMap's bounds should equal the TriGrid dimensions; no sanity check is done for this inside the routine. Caller must assure sanity.

```
void   TriGridColorize(PixMapHandle pix,
                           TQ3GeometryObject TriGrid)
{
   TQ3Status            status;
   TQ3TriGridData       data;
   TQ3AttributeSet      attribs;
   TQ3ColorRGB          color;
   RGBColor             rgb;
   long                 i,k,n;

   // Try to fetch TriGrid's data...
   if (Q3TriGrid_GetData( TriGrid, &data ) != kQ3Success)
      DebugStr("\pData can't be fetched from TriGrid.");

   for (i = n = 0; i < data.numRows; i++)
      for (k = 0; k < data.numColumns; k++) {

         GetColorPixel( pix, i, k, &rgb );    // fetch pixel value

         RGB32ToRGBFloat( &rgb, &color ); // convert to floats

         attribs = nil;
                                              // get vertex attribs
         status = Q3TriGrid_GetVertexAttributeSet( TriGrid, i,
              k, &attribs );

         if (attribs == nil)                  // if none,
            attribs = Q3AttributeSet_New();   // make some

                                              // add our color to it
         status = Q3AttributeSet_Add ( attribs,
            kQ3AttributeTypeDiffuseColor, &color);

                                              // reattach attributes
         status = Q3TriGrid_SetVertexAttributeSet( TriGrid, i,
            k, attribs );

         status = Q3Object_Dispose(attribs);  // dump object ref
      }

   Q3TriGrid_EmptyData( &data );              // dispose of copied data
}
```

As you can see, we begin (as always) by fetching the TriGrid's data with Q3TriGrid_GetData(). We set up a double loop to traverse the vertices in all rows and columns, and since we know that our source image has the same dimensions as our TriGrid, all we need to do is pluck the color information from each pixel in our PixMap and apply it to each vertex in the grid. Why each vertex? Why not each face? If we just give each face its own color, we'll end up with a terrain that looks like one big mosaic. By contrast, if we assign a color to each vertex and set the rendering view's fill style to kQ3InterpolationStyleVertex, Quickdraw 3D will color the TriGrid's faces for us in a visually appealling way, by *interpolating colors between vertices*. This gives a good visual result without a big speed hit in rendering. This is why we choose to apply colors to just our vertices.

Note that we use a homemade utility function, GetColorPixel(), to fetch color info and parse pixel bits into an RGBColor, because we have to be able to handle 8-, 16-, or 32-bit pixels. The Mac Toolbox API provides a routine, GetCPixel(), that does essentially the same thing, but for some reason the Toolbox routine is extremely (make that painfully) slow, so we use our own routine, which is about 100 times faster.

Once we have pixel data in RGBColor form, we need to convert it to floating-point representation. Again, we rely on a homegrown routine, RGB32ToRGBFloat(), to do this. (This utility could be implemented as a macro. It just divides each of the RGBColor's short ints by 65535.)

To "colorize" a vertex requires giving the vertex a diffuse-color attribute. Note carefully the distinction between an *attribute* (which can be one of a dozen different things, such as a color, a transparency value, a surface-normal vector, etc.) and an *attribute set*. Attributes come in many flavors and types. Attribute sets, on the other hand, are container objects that hold arbitrary numbers and kinds of attributes. The diffuse-color attribute (which is of type kQ3AttributeTypeDiffuseColor) is a TQ3ColorRGB, which we have to add to the vertex's attribute set. First we check to see if the vertex has an attribute set, by calling Q3TriGrid_GetVertexAttributeSet(). If the returned attribute set is nil, we need to create a new attribute set with - what else? - Q3AttributeSet_New(). Once we've got our attribute set, we add an attribute to it with Q3AttributeSet_Add(). In our case, we're adding a *diffuse color*, but we could just as well be adding a vertex normal (a vector) or some other kind of attribute. The vertex still hasn't changed, though, until we call Q3TriGrid_SetVertexAttributeSet() to attach the attribute set to the vertex. The latter call increments the reference count for the attribute set, so to keep our bookkeeping in order we need to be sure to call Q3Object_Dispose() on the no-longer-needed copy of the attribute set.

If the preceding paragraph made any kind of sense to you, congratulations. You're either an experienced QD3D programmer or well on your way to becoming one. (For an explanation of reference counts and how to manage them, see my original QD3D introductory article in the July *MacTech*.)

A New Way to Look at Images

The TriGrid, when it is finished, is added to a group object comprising our model. The group object is a container that not only holds the geometry (the TriGrid) but a rotation transform, a scaling transform, and an illumination shader. The purpose of the transform objects is to let us perform rotations and scaling operations on just this object, independently of any other objects that might be onscreen. In this particular application, we don't happen to have any other 3D objects, but in a more complex app you'd want to have individual control of large numbers of objects, and this is one way to achieve that.

The compiled app, PMB, will let you open any TIFF, JPEG, GIF, PICT, or Photoshop 3 image (providing you've got QuickTime 3.0 installed), display the image in a draggable window, and see the same image converted into a 3D terrain in another window. Clicking in the terrain window, you'll find that you can swivel the terrain in real time by simply dragging the mouse. In the menu bar, you'll note that there is a Render menu to let you choose whether to render the terrain as a filled object, a wireframe object, or a collection of dots (representing the vertices). These mode changes occur in real time.

In the Prefs menu, you'll find three choices for grid resolution: Coarse Grid (the default), Medium Grid, and Detailed Grid. These choices represent a factor of two difference, relative to the preceding subdivision level, with Detailed Grid giving you a TriGrid with MAX_VERTEX_COUNT vertices. (I've set this value to 4,096 in the project files, but you can set it to whatever you want.) The grid resolution that you choose in the Prefs menu *does not* take effect immediately; instead, it takes effect with the *next* terrain you make.

If you play around with PMB, you'll get a good idea of what kind of speed and image quality Quickdraw 3D is capable of. Overall, the results are not bad. (Most people who see QD3D in operation for the first time are pleasantly surprised.) You'll find that the "terrains" created in PMB are complex and in some cases reminiscent of real mountains. It's no substitute for a commercial terrain-generation program, however. For one thing, the "skies" are not realistic; PMB is hard-wired to give a solid blue background. There's also no provision for creating bodies of water, trees, grass, snow cover, etc. On top of that, the terrain looks a bit crumpled and faceted, in an eerily regular kind of way (owing to the uniform spacing of vertices). No real terrain ever looked quite like this.

We can give our terrains more visual appeal in a couple of ways. First, it would be nice to be able to lay an actual texture image over our grid rather than merely coloring the vertices. As it turns out, applying texture images in this fashion is easy to do in Quickdraw 3D; we'll take a look at how it's done in a future article.

It would also be nice if our terrains weren't quite so sculpted-looking -so faceted. Again, QD3D gives us some interesting options here. As it stands now, we're letting Quickdraw 3D's renderer generate our vertex normals for us, but with a little work we can recalculate vertex normals ourselves in a way that causes hard edges to appear rounded or smoothed. We'll also talk about this next time.

We can also make our terrain look more, well, terrain-like by jittering the vertices so that the lattice spacing is not so apparent.

These are all interesting things to experiment with. If you end up looking at the sample code and compiling the project yourself, feel free to try some ideas of your own for making terrains look "realistic."

What Did We Learn?

Creating a height-mapped 3D grid from a 2D image in Quickdraw 3D is not difficult. The trickiest part of the task may well just be knowing which Mesh geometry to use: QD3D provides four different freeform-mesh primitives, with markedly different characteristics in terms not only of topology but storage requirements, rendering speed, and ease of coding. The TriGrid turned out to be the Mesh primitive best suited to our terrain-mapping task, but it's worth pointing out that we could have used Mesh, Polyhedron, or TriMesh models to accomplish the same thing, albeit with different code. (This is left as an exercise for the reader.)

In future articles, we'll take a look at some of the more advanced capbilities that Quickdraw 3D gives programmers in terms of geometry editing, texture mapping, vertex normal recalculation, and group (container) object traversal and manipulation. Once you get used to the QD3D way of doing things, who knows? You may find that 3D isn't so difficult after all.

Kas Thomas, tbo@earthlink.net has been a Macintosh user since 1984 and has been programming in C on the Mac since 1989. He holds U.S. Patent No. 5,229,768 for a high-speed data compression algorithm (licensed to Traveling Software) and is the author of a QD3D-powered Photoshop® plug-in called Callisto (available at http://users.aol.com/callisto3d).