MacTech

Volume Number: 14
Issue Number: 11
Column Tag: Power Graphics

Poor Man's Bryce, Part II: More Terrain Generation with Quickdraw[TM] 3D

by Kas Thomas

Texture-mapped terrains are easy to create in Quickdraw 3D

Quickdraw 3D is a cross-platform graphics library designed with performance and ease of programming foremost in mind. It's a tremendously powerful tool for giving users an interactive 3D data-visualization experience. With not much effort, it's possible for the non-graphics specialist to put together a program that can let a user manipulate fully shaded 3D objects in real time - something only supercomputers could do just a few short years ago, and then only with great effort.

In a previous article (Part 1 appeared in October's MacTech), we saw how easy it is to set up a simple Quickdraw 3D program that converts a user's 2D input (in the form of a TIFF, PICT, JPEG, GIF, or Photoshop 3 file) to a 3D "terrain" using the popular technique of displacement mapping. The basic trick here is to set up a flat (planar) grid with vertices equal - or at least proportional - in number to the number of pixels in the input image, then loop over the grid vertices (and image pixels), assigning an "elevation" value to each vertex based on the corresponding pixel intensity in the source image. In this way, a series of polka dots becomes a mountain range, a letter 'O' becomes a crater, etc. Or, a U.S. Geological Survey DEM (digital elevation map) file can be converted to a 3D topographic terrain.

For version 1.0 of our PMB (Poor Man's Bryce) application, we chose Quickdraw 3D's TriGrid mesh primitive as the starting point for our geometry because of its inherently Cartesian (rows-and-columns-oriented) layout. The TriGrid, however, is only one of four freeform mesh primitives available in Quickdraw 3D (the others being the Mesh, TriMesh, and Polyhedron). The pros and cons of the various mesh types were discussed in some detail last time. Suffice to to say, we chose the TriGrid partly for performance (it renders quickly compared to, say, a plain Mesh), partly for its efficient use of RAM (vertices are shared in a near-optimal fashion), and partly for ease of coding. I also mentioned last time that the TriGrid would offer certain advantages when it came to applying texture maps. The significance of this will (at last) be clear shortly.

For version 2.0 of PMB, we're going to add features that improve the appearance of our pseudo-terrains. One thing we'd like to be able to do is make the terrains a bit less polygonal-looking (i.e., less faceted; smoother). Another thing that would be very useful is the ability to overlay PICT images (texture maps) on our grid. Both of these things are remarkably easy to do in Quickdraw 3D. Follow along and I'll show you what I mean.

Smoothing the Rough Edges

With the rise in popularity of computer-generated 3D images, we've all become a bit accustomed to seeing objects that look faceted or polygonal (**Figure 1**). In some instances, such as when recreating real-world objects that really are faceted (like geodesic domes), the polygonal look is perfectly natural. But in most instances, it detracts from the appearance of the object, because in the real world most objects are not polygonal-looking. A real tomato tends to be curved, not flat-sided. (The flat-sided ones, you leave in the bin.) The question is, how do you get an object with lots of flat surfaces to look curvy and smooth?

Figure 1. *The typical "faceted" appearance that results when Gouraud shading is not utilized.*

To understand the answer, we need to understand why faceted objects look faceted in the first place. The reason facets look like facets is that the human visual system is extremely sensitive to discontinuities in light intensity. At the edges of adjoining facets, the intensity of reflected light takes a sharp jump, and it's this sudden change of color that gets our attention. Back in 1971, a fellow by the name of H. Gouraud proposed a simple workaround for making edges vanish in computer-generated images. Gouraud's trick relies on the fact that most illumination models calculate reflected light intensities based on the angle between the light vector and the so-called "surface normal" vector associated with a reflective surface. (The surface normal is just a vector sticking straight out of a planar surface at right angles to it, like a stop-sign pole sticking up out of a sidewalk.) Gouraud reasoned that if one were to calculate something called a vertex normal at each of a polygon's vertices, then calculate intermediate normals between vertices by simple interpolation, it should be possible to use the intermediate (interpolated) "normals" to calculate reflection; and since these would vary smoothly from corner to corner and edge to edge, the light values would vary smoothly as well, eliminating the sharp discontinuity in lighting across edges. (See **Figure 2**.)

Figure 2. *The same object as Fig. 1, but with Gouraud shading.*

Technically speaking, linear interpolation doesn't really do away with the C1-discontinuity problem (as mathematicians like to call it), but for most purposes the Gouraud trick works well enough - it does fool the eye into thinking edges have vanished or, at the very least, become much smoother.

And the neat thing for us is that Quickdraw 3D's renderer will do the hard work of between-vertex interpolation for us (and calculate our lighting), if we'll just provide the vertex normals. Note that if we don't provide vertex normals (as a type of vertex attribute), QD3D's renderer calculates a polygon (surface) normal for each surface, on the fly, and uses that, which of course is why our surfaces look faceted.

Vector Math in a Nutshell

In case you haven't had any exposure to vector math since high school, it might be good to stop for a moment and brush up on basic concepts. (Skip this section if you're already up-to-speed.) In Quickdraw 3D, we deal mostly with (what else?) 3D vectors. A 3D vector, in turn, can be any quantity that has three numerical components. For example, points in 3-space have x,y, and z coordinates. Does this mean points are vectors? Not per se, no. A vector has both direction (or orientation) and magnitude. Points don't qualify, strictly speaking, but differences between points do constitute vector quantities. If you say "Start at Fisherman's Wharf and proceed south until you reach the Cow Palace," you've given me a vector defined by two points. In like manner, a vector can be defined by a point's position in 3-space relative to the origin (which after all is just another point in 3-space). We call this the point's position vector. More generally, we call the vector defined by any two points a difference vector.

The neat thing about vectors, of course, is that they can be added, subtracted, and/or multiplied to give new vectors. (Vector division is not defined.) Adding vectors is a simple matter of adding their respective components together. Subtraction works by subtracting individual components. Multiplication is a bit more complicated. The so-called dot product of two vectors is obtained by multiplying the respective components together and summing them, which gives a scalar (one-dimensional) quantity. The cross product or vector product is obtained by multiplying the two vectors' components against each other in a particular way to yield new components (and a new vector).

Now here's the amazing part. The new vector that you get when you cross two 3-space vectors together will always be at right angles to each of the original vectors. What's more, the direction of the new vector will be dependent on the order in which you crossed the two original vectors. A 5 B gives a vector that points 180° opposite of B 5 A.

In order to calculate lighting, a renderer needs to know the direction from which the light is coming and the angle at which the light hits a given surface. The necessary information is contained in the dot product of the surface normal and the normalized lighting vector. Before going further, let's clarify this funny word "normal." We mentioned that vectors have magnitude as well as direction. The magnitude is something you can calculate very simply using the Pythagorean distance formula. That is, you square each of the vector's components (x,y, and z), add the squares together, and take the square root of the sum. Child's play.

Vector division may be undefined, but you can certainly divide each of a vector's components numerically, one at a time. If you do that using the vector's magnitude as the divisor, you end up with individual components less than unity, but the new vector will have a magnitude of (guess what?) exactly one. This is vector normalization. (Prove that a 3D vector with x = y = z will always, upon normalization, have components equal to 0.57735.)

Normalization is extremely handy, because the dot product of two normalized vectors is exactly the cosine of the angle between them. (Observe also that the cross product of two normalized vectors is equal in magnitude to the sine of the angle between the vectors.)

Vertex Normals

Now we're in a position to try to calculate vertex normals for our terrain grid. What we're really trying to do is loop over all vertices in our grid, obtain surface normals for all neighboring surfaces of which a given vertex is a member, and average the normals together. (See Figure 3.) This sounds worse than it is. It turns out Quickdraw 3D has some handy utility routines to make our job a lot easier.

Listing 1 shows the first of two routines we call on in PMB to accomplish vertex-normal calculation. Actually, the routine shown in Listing 1 just retrieves our geometry's data and sets up a nested loop to traverse all grid vertices. The actual vector manipulations come in Listing 2.

Figure 3. *A vertex normal can be calculated by averaging together the surface normals of all polygons of which the vertex is a member.*

We begin by getting access to our TriGrid data, which is embedded in a display group object. The code in

Listing 1 shows some very typical calls that are used for this purpose; you'll see this kind of code over and over again in Quickdraw 3D programs. The API call Q3Group_GetFirstPositionOfType() fetches the address (if any) of the first embedded object of the type indicated. In this case, we're passing the constant kQ3GeometryTypeTriGrid to indicate that we're looking for a TriGrid inside the group object. If there is no such geometry inside the group, we'll get a position value of NIL (but the function may return kQ3Success nonetheless), so it's important to check not only the return value but the validity of the TQ3GroupPosition value.

'The subsequent call to Q3Group_GetPositionObject(), which relies on the "position" given us by Q3Group_GetFirstPositionOfType(), should return a valid reference to our geometry. Again, we check not only for a successful return value but a non-NIL geometry object. If we succeed in getting our geometry at this point, we will have incremented the reference count for our TriGrid, which in turn means we'll have to *decrement* the ref count later (before our function returns) if we want to keep all references balanced and stave off memory leaks. (The subject of reference counts was addressed in my article on QD3D fundamentals in the July 1998 issue of *MacTech*.)

We still can't work with the geometry's data, however, until we make a call to Q3TriGrid_GetData(). Until we make this call, the locations of our vertices and other particulars relating to the TriGrid are known only to QD3D. The "get data" call will make all the information public again, but really what's happening is that QD3D is making a *copy* of internally stored info for us. It's important to understand that we are *not* just being given a TQ3TriGridData structure containing a few bytes of information and a few pointers pointing to arrays in the system heap. Quickdraw 3D is actually giving us the TQ3TriGridData structure *and* a fresh copy of all the array data pointed to by the struct. To make sure this memory later gets freed up, we have to remember to call Q3TriGrid_EmptyData() prior to exiting our function for any reason.

Once we have access to the TriGrid's data (and such items as the number of rows and columns in the grid), we can go ahead and loop over all the vertices one by one. For each one, we call our own routine, GetVertexNormal(), which calculates the vertex normal (see Listing 2). We then add the vertex normal to the attribute set of the vertex in question - and store the modified attribute set back into the vertex.

The code for fiddling with vertex attributes should look familiar to you, but in case it doesn't, here's what's going on. (Again, this is a common programming paradigm in QD3D, so be sure to understand it. You'll use this type of routine over and over again.) Attributes are, in general, surface properties of various kinds. They can be colors, transparency values, surface normals, or any of a zoo of other types. A vertex, in QD3D, is actually a data structure encompassing both a 3D point (a TQ3Point3D) and a pointer to an attribute set (of type TQ3AttributeSet). The essential thing to remember is that an attribute set is a container object that holds attributes. The container may be empty, or it may contain one kind of attribute, or it may be full of attributes, but the important thing is, you don't add attributes directly to vertices (or to other objects) - rather, you add attributes to attribute sets, and then add attribute sets to the objects they modify. If you look carefully at Listing 1, you'll see that this is indeed what takes place. We try, first, to access any attribute set that might already be attached to our vertex. If we come up empty-handed, we call Q3AttributeSet_New() to create a new, blank attribute set. Once we have the attribute set, we add our vertex normal (a vector quantity, remember) to it with Q3AttributeSet_Add(). But notice that we still haven't changed anything's appearance until we call Q3TriGrid_SetVertexAttributeSet(). It's this call that actually attaches the new attribute set to the vertex.

In case you're wondering, a call to Q3AttributeSet_Add() overwrites any "like data" that may be contained in an attribute set. If there had been any preexisting surface-normal data, we would not (technically speaking) be obliged to clear it out with an AttributeSet_Clear call, although the API does provide for such things.

Note that since attribute sets are objects, obtaining a reference to one (for purposes of editing it) means you later need to decrement its reference count with a call to Q3Object_Dispose(). By doing this inside our loop, we avoid memory leaks and keep the QD3D gods happy.

Listing 1: CalculateVertexNorms()

```
CalculateVertexNorms
void CalculateVertexNorms( TQ3GroupObject grid ) {

   TQ3TriGridData       data;
   TQ3GeometryObject    tri;
   TQ3GroupPosition     pos;
   TQ3AttributeSet      attribs;
   TQ3Status            s;
   TQ3Vector3D          vertexNormal;
   unsigned long        i,j;
   s = Q3Group_GetFirstPositionOfType( grid, kQ3GeometryTypeTriGrid, &pos );
   if (s != kQ3Success || pos == nil)
      DebugStr("\pNo trigrid in group!");
   s = Q3Group_GetPositionObject( grid, pos, &tri); // get the trigrid
   if (s != kQ3Success || tri == nil)
      DebugStr("\pCan't get trigrid!");
   s = Q3TriGrid_GetData( tri, &data );
   if (s != kQ3Success)
      DebugStr("\pCan't get trigrid data!");

   for (i = 0; i < data.numRows ; i++) {   // for all interior rows
                                           // and all interior columns
      for (j = 0; j < data.numColumns ; j++,attribs=nil) {
         vertexNormal = GetVertexNormal( tri, i, j, data.numRows, data.numColumns );
         // get attribute set for this vertex...
         Q3TriGrid_GetVertexAttributeSet ( tri, i,j, &attribs );
         if (attribs == nil)
            attribs = Q3AttributeSet_New();
         // add the normal to the attributes
         s = Q3AttributeSet_Add( attribs, kQ3AttributeTypeNormal, &vertexNormal);
         if (s != kQ3Success)
            DebugStr("\pCouldn't add attribute.");
         // store the amended attributes...
         s = Q3TriGrid_SetVertexAttributeSet(tri,i,j,attribs);
         if (s != kQ3Success)
            DebugStr("\pCouldn't set attribute.");
         Q3Object_Dispose( attribs ); // ditch the object when done
         } // j
      } // i
   Bail:
   s = Q3Object_Dispose(tri);          // dispose of object when done
   Q3TriGrid_EmptyData( &data );   // free the data
}
```

It's in Listing 2 - and the routine GetVertexNormal() - that we actually do the "hard work" (if you can call it that) of calculating vertex normals for each vertex in our TriGrid. The routine in question takes as arguments a reference to our TriGrid object, the row and column indices for our vertex-of-the-moment, and the overall row and column dimensions of the grid. We return a TQ3Vector3D to the caller, representing the actual vertex normal for vertex i, j.

One of the wonderful things about the Quickdraw 3D API is the rich set of utility routines available for working with various kinds of objects, such as TriGrids. It's not always necessary or convenient, for example, to retrieve a TriGrid's entire data set just in order to work on one vertex; in which case, you can simply call Q3TriGrid_GetVertexPosition() with the row and column number of the vertex you're interested in. QD3D, in turn, hands you the 3D coordinates of that vertex's point. Since a point (a TQ3Point3D) isn't an object, there is no need to worry about object referencing/dereferencing or calls to Q3Object_Dispose(). You just fetch the point coordinates and do whatever you want with them. (If you need to store them back into the grid, there's a corresponding Q3TriGrid_SetVertexPosition call you can use.)

For purposes of vertex normal calculation, it's important that we know not only the coordinates of the

vertex we're working on but those of its neighbors to the North, South, East, and West, too. Accordingly, we fetch the coordinates of these "nearest neighbor" points with additional calls to Q3TriGrid_GetVertexPosition(). Of course, if we're working with vertices at the very edges of the TriGrid, there won't necessarily be neighboring vertices on all sides; hence it's important to check for these boundary cases. If we're at an edge, we just ignore the missing vertices and go on with a partial smoothing based on vertex info from points on the other side of the edge.

Ordinarily in a routine like this, we'd want to start by forming a difference vector by subtracting our central point from its northern neighbor, then another difference vector made by substracting the center point from (say) its eastern neighbor. We might call these difference vectors the "noon" vector and the "3 o'clock" vector. To get the surface normal for the northeast quadrant, we'd cross the noon and 3-o'clock vectors, and normalize the result. Then we'd proceed all the way around the clock this way until we had surface normals for all quadrants, add up the normals, renormalize them, and return the result to the caller.

Instead of doing all that, we take some shortcuts. First, instead of forming difference vectors by hand and crossing them, we rely on a little-known QD3D call, Q3Point3D_CrossProductTri(), which takes three *points* as arguments, along with a *pointer to a vector* (where the result is placed). This API call "knows" what we're trying to do: It forms difference vectors between the three points and returns the cross product to us, all in one line of code! It makes perfect sense, of course, because intuitively we know (and QD3D knows) that any three points in space define a *plane*, and from any plane we can surely derive a surface normal. The API call doesn't *normalize* the resulting vector for us, but that doesn't matter. We can normalize it at the end, after accumulating (summing) all the "quadrant normals" into a single resultant vector.

Purists will argue as to whether it's best to normalize each cross product first, before accumulating the resultant vector, or normalize once, at the end. There is a difference, of course, and it's not at all hard to visualize. (Try it.) In one case you're adding together a bunch of vectors of *unequal magnitude*, then taking a weighted average of them; in the other case, you're averaging vectors of *equal* magnitude. The latter case is like saying that all directions on the compass are equally important; we just want the arithmetic average of the directions. The former (weighted average) case is like saying *bigger* surfaces are more important than *smaller* surfaces, so they should have "more say" in the directional result. There's obviously no one "right answer." In the code shown here, I have left it a weighted average, not only because it results in fewer lines of code but because it implements (at least in a brain-damaged way) a technique known as edge preservation, which is where really sharp edges get less smoothing than not-so-sharp edges. (The need for edge preservation may not be obvious now, but when it comes to modeling things like rocket nose cones, airplane wings, and knife-edged equipment, it's critical to let some - but not all - edges be as obvious as possible.)

Listing 2: GetVertexNormal()

```
GetVertexNormal
// Get the 4 points surrounding the vertex at i,j then do a bunch of
// cross-products and averaging to get an average vertex normal at i,j.
TQ3Vector3D GetVertexNormal( TQ3GeometryObject tri,
                      unsigned long i,
                      unsigned long j,
                      unsigned long rows,
                      unsigned long cols ) {

   TQ3Point3D       pSouth,
                    pWest,
                    pNorth,
                    pEast,
                    pCenter;

   TQ3Vector3D      vertexNormal = {0.,0.,0.},
               cross;
```

```
    // get the central vertex coords...
    Q3TriGrid_GetVertexPosition( tri, i,j, &pCenter );
    // get the coords of our neighbors...
    Q3TriGrid_GetVertexPosition( tri, (i > 0) ? i-1 : 0, j,
    Q3TriGrid_GetVertexPosition( tri, i,           (j > 0) ? j-1 :0, &pWest );
    Q3TriGrid_GetVertexPosition( tri, (i < rows-1) ? i+1 : i, j, &pSouth );
    Q3TriGrid_GetVertexPosition( tri, i,           (j < cols-1) ? j+1:j, &pEast );
    // South X West
    Q3Point3D_CrossProductTri ( &pSouth, &pCenter, &pWest, &cross );
    Q3Vector3D_Add( &cross, &vertexNormal, &vertexNormal );
    // West X North
    Q3Point3D_CrossProductTri ( &pWest, &pCenter, &pNorth, &cross );
    Q3Vector3D_Add( &cross, &vertexNormal, &vertexNormal );
    // North X East
    Q3Point3D_CrossProductTri ( &pNorth, &pCenter, &pEast, &cross );
    Q3Vector3D_Add( &cross, &vertexNormal, &vertexNormal );
    // East X South
    Q3Point3D_CrossProductTri ( &pEast, &pCenter, &pSouth, &cross );
    Q3Vector3D_Add( &cross, &vertexNormal, &vertexNormal );
    // normalize the result
    Q3Vector3D_Normalize( &vertexNormal, &vertexNormal );
    vertexNormal.z *= -1.; // make it face the right way
    return vertexNormal;    // return the average
} // end function
```

Critics will notice, incidentally, that I have not done as thorough a job as possible in smoothing the vertex normals, because (in this implementation) we are only taking the average of four surface normals: North, South, East, and West. We are totally ignoring Northwest, Southwest, Northeast, and Northwest. (A true dilettante would weight the corner vertices by a factor of .707.) I would argue that fewer lines of code make for faster code, and anyway, by simply taking into account the major points of the compass, we have achieved 90% of the smoothing that we'd get any other way, in any case. (The proof is in the pudding: **Figure 4** shows a comparison of smoothed and unsmoothed geometries.)

Feel free, of course, to experiment with the source code yourself. (Complete CodeWarrior project code is available at the MacTech web site, http://www.mactech.com.)

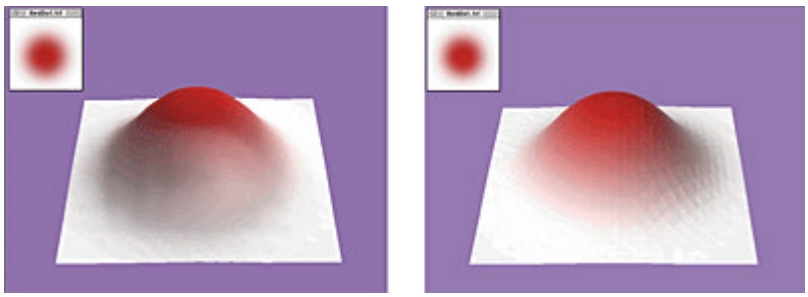

Figure 4. *Side-by-side comparisons of the same PMB object with and without vertex normals. The small amount of effort required to calculate vertex normals for Gouraud shading is well worthwhile.*

Texture Mapping

In version 1.0 of our PMB app, we elected to apply colors to vertices in a very straightforward way, faithfully mapping RGBColors from our source image's PixMap to the individual vertices of the TriGrid. (See Part 1 of this article, October.) This results in darker colors mapping directly to higher elevations and vice versa. Although QD3D does a good job of shading our grid in an eye-pleasing manner, it would still be nice to apply other color schemes to our geometry. For example, it might be interesting (when rendering terrains) to make low-lying areas green - lush with vegetation, perhaps - but higher areas brown (or even white, above the snow line). Using the code from Part 1 of this article, it shouldn't take you long to figure out how to accomplish this. You could easily write a routine that maps a particular color palette to elevations so that higher areas have a different coloration than lower areas. This is the essence of what hardcore 3D render-junkies know as "procedural shading," which is a fascinating topic in

its own right. Algorithmic shading converts 3-space coordinate info into colors or shadings, giving volumetric correlation to pixel hues. Some very complex, subtle, and impressive shadings can be achieved this way. (For details, consult Ebert's *Texturing and Modeling*, 2nd Edition, Academic Press, 1998.)

For the ultimate in creative shading, there's no beating the tried-and-true technique of *texture mapping*, which is where a 2D image is "wrapped onto" the surface of a 3D object. Quickdraw 3D supports texture mapping with a rich set of API calls which (coincidentally) take full advantage of hardware support, if present, with no additional code required on the QD3D programmer's part.

To understand how texture mapping works, it's necessary to begin with a discussion of *surface parametrization*, which is basically the process of constructing and deconstructing three-dimensional surfaces two-dimensionally. Most 3D objects that you or I are likely to work with are bounded such that they have an *inside* and an *outside*. The boundary between a 3D object's inside and outside - its *surface* - is, in some sense, intrinsically two-dimensional in that it can usually be considered as adjoining 2D polygons. A flea walking on a donut doesn't know that the world isn't flat. At some scale, *all* surfaces are flat.

The problem is how to create a mapping from the 2D worldview to the 3D worldview, so that every point on a 2D image can be mapped to a corresponding point on a 3D surface. If you can successfully do this, you can apply an image to a 3D object, like so much shrink wrap.

Now obviously, if you're simply trying to map a picture of Aunt Ruth onto the sides of a cube, there's nothing to worry about. The mapping is straightforward. The upper left corner of the picture maps to the upper left corner of each face of the cube, and the lower right corner of the picture maps to the lower right corner of each face. Aside from a possible aspect-ratio correction factor, there's not much to worry about.

But suppose you want to map the same picture to the sides of a tetrahedron. If the picture is triangular, no problem. More often than not, though, the picture will be rectangular (as most pictures of Aunt Ruth are) and you may be saddled with a requirement not to crop anything out, nor leave any unused space on the tetrahedron. What then?

What then is, you perhaps resort to a more creative type of mapping - a non-linear mapping where the units of 2D distance don't necessarily march in lockstep to units of 3D distance. One possibility is to tell the renderer to apply severe spatial compression (in the x-direction) to the top edge of the photo, so that all pixels in that top raster line map to the apex (the top vertex) of any given face of the tetrahedron. You also tell the renderer to squeeze the bottom edge of the photo just enough so that it exactly fits along the bottom edge of each triangular face. For all intermediate parts of the photo (between the top and bottom edges), you instruct the renderer to use a compression factor that is linearly interpolated between top and bottom. Aunt Ruth may end up looking like a pinhead using this technique, but at least all of her is there, and there is no unused space on any of the tetrahedron's faces.

What we need to talk about now is the special way that you tell Quickdraw 3D's renderer how to do all of this. The technique is colloquially known as u,v mapping or u,v parametrization.

Working in (u,v) Space

To map a 2D image onto a 3D surface, we have to assign u,v coordinates to every vertex in 3-space. The idea is simple. The lower left corner of our 2D image will correspond to u,v coordinates of (0,0). The upper right corner of our rectangular image will be (1,1). Everything in between will be some ordered pair of appropriate floating-point values between zero and one. For example, the geometric center of the image will be (0.5,0.5). Conversely, if I tell you to locate the part of the image given by (0.25,0.75), you will immediately know to go to a spot 25% of the way across the image and 75% of the way up it, relative to the lower left corner. (Note that QD3D does things a little differently here than in some

graphics systems, such as ordinary Quickdraw , where y increases in the downward direction.)

The trick is to assign corresponding u,v values (covering the entire domain of u,v space) to the surface of your 3D object, whatever it happens to be. It turns out that for a TriGrid, the parametrization is quite natural. One corner of the grid corresponds to (0,0) and the diagonally opposite corner corresponds to (1,1). Since the grid has a fixed number of rows and columns, the rest is easily obtained by linear interpolation.

In order for QD3D to know what's going on, it's necessary for you to assign a u,v parametrization to every vertex in the TriGrid. This just means setting up a rows-and-columns nested loop and looping over all the vertices, divvying up u,v space in straightforward manner. Obviously, in a grid with ten rows and ten columns, the vertex spacing (in u,v-space) would be 0.11. (That's no misprint: The spacing is 0.11, not 0.10. Remember that there are *nine intervals* separating *ten points*, hence a spacing of 1/9 == 0.11.) Why does everything have to fall between 0.0 and 1.0? By specifying our mappings in "normalized units" this way, we make it possible for QD3D to use any size image as a texture map. The source image could measure 50 pixels square or it could be 1024 by 768 pixels, or something even biggger; no matter. The image will always map correctly (provided we don't run out of video RAM along the way!), with no adjustment of u,v coordinates necessary between re-textures. Map once, texture many.

Listing 3 shows how the mapping is done. As usual, we begin by fetching our geometry from the group object in which it is nested; we access the TriGrid data with a call to Q3TriGrid_GetData(); and we set up a rows/columns double loop. Not unexpectedly, our u,v parameters must be added to the vertices as attributes in an attribute set. (As before, we follow the familiar paradigm of accessing or creating the attribute set, loading it with our desired attribute, and attaching it to the vertex in question.) The u,v values are put into the appropriate fields of a TQ3Param2D data structure. Notice how the u,v values are calculated directly from the row and column indices. (This is possible because we know that our TriGrid has a fixed number of rows and columns.) We invert the ordering by subtracting each u and v from 1.0 in order to make the texture map not look like it's backwards.

When we add each TQ3Param2D to the appropriate attribute set, we tell QD3D what we're doing by specifying the constant kQ3AttributeTypeShadingUV. (As explained earlier, there are many possible types of attributes, and many pre-defined constants associated with them.)

To stave off memory leaks, we dispose of unneeded objects when finished and make a call to Q3TriGrid_EmptyData() to free the memory that QD3D reserved for us when we requested a copy of the grid's data. (We really only needed that data in order to obtain the row and column counts.)

Listing 3: TriGrid_AddUVParams()

```
TriGrid_AddUVParams
void TriGrid_AddUVParams( TQ3GroupObject grid ) {

   TQ3TriGridData     data;
   TQ3GeometryObject  tri = nil;
   TQ3GroupPosition   pos;
   TQ3AttributeSet    attrib;
   TQ3Status          s;
   TQ3Param2D         param;
   long               i,k;
   // Get the first trigrid position from this group...
   s = Q3Group_GetFirstPositionOfType( grid, kQ3GeometryTypeTriGrid, &pos );
   if (pos == nil || s != kQ3Success) return;
   s = Q3Group_GetPositionObject( grid, pos, &tri);// now get trigrid
   if ( s != kQ3Success)
      DebugStr("\pCan't get position object.");

   if (Q3TriGrid_GetData( tri, &data ) != kQ3Success)
      DebugStr("\pCan't get trigrid data."); // failure, so return now
```

```
   for (i = 0; i < data.numRows; i++) {
      for (k = 0; k < data.numColumns; k++,attrib = nil) {

      Q3TriGrid_GetVertexAttributeSet( tri, i, k, &attrib );
      if (attrib == nil)
         attrib = Q3AttributeSet_New();

      param.v = 1. - (float)i/(data.numRows-1);
      param.u = 1. - (float)k/(data.numColumns-1);
      Q3AttributeSet_Add(   attrib,
                     kQ3AttributeTypeShadingUV,
                     &param);

      Q3TriGrid_SetVertexAttributeSet( tri, i, k, attrib );

      if (attrib != nil)
         Q3Object_Dispose( attrib );
      } // k
   } // i
Q3TriGrid_EmptyData( &data ); // very important! frees memory
 Q3Object_Dispose(tri);

}
```

So far, so good. Now, how do we actually tell Quickdraw 3D to map an image onto our geometry? First we let our user select an image file by means of a Standard File dialog. In this case, we're working with PICT files, so we open the selected file, read it into a handle, and pass the PicHandle to a routine where the image PixMap gets transcribed into a TQ3StoragePixmap. Why all the transcription? Two reasons. First, you probably don't want to keep a big PixMap floating around in your application's heap forever. By converting it to a TQ3StoragePixmap, you essentially offload the image into video RAM and/or system heap, where it's less likely to contend for application memory. Secondly, recall that QD3D is a cross-platform API. It wouldn't do the QD3D cause much good if Windows programmers had to be relied upon to convert everything to PicHandles and PixMaps before texture mapping could happen. The TQ3StoragePixmap is a machine- and OS-independent type of storage which, despite the presence of "pixmap" in the name, is not tied in any way to the Mac OS.

The storage pixmap has a data structure that looks like this:

```
typedef struct TQ3StoragePixmap {
   TQ3StorageObject   image;
   unsigned long      width;
   unsigned long      height;
   unsigned long      rowBytes;
   unsigned long      pixelSize;   // must be 16 or 32
   TQ3PixelType       pixelType;
   TQ3Endian          bitOrder;
   TQ3Endian          byteOrder;
} TQ3StoragePixmap;
```

As you can see, there's nothing mysterious about the storage pixmap; it's just another kind of bitmap. Currently, the pixel size field is limited to values of 16 or 32 (assuming you're working with Apple's default interactive renderer), although in the future other pixel sizes will no doubt be accommodated. Conversion of our user's texture PixMap to TQ3StoragePixmap form occurs in our application's routine LoadMapPICT(), shown in Listing 4.

Listing 4: LoadMapPICT()

```
LoadMapPICT
short LoadMapPICT(
   PicHandle           pict,
   unsigned long       mapSizeX,
   unsigned long       mapSizeY,
   TQ3StoragePixmap    *bMap)
```

```
{
   unsigned long       *textureMap;
   unsigned long       *textureMapAddr;
   unsigned long       *pictMap;
   unsigned long       pictMapAddr;
   register unsigned   long    row;
   register unsigned   long    col;
   Rect                rectGW;
   GWorldPtr           pGWorld;
   PixMapHandle        hPixMap;
   unsigned long       pictRowBytes;
   QDErr               err;
   GDHandle            oldGD;
   GWorldPtr           oldGW;
   short               success = 1;
   GetGWorld(&oldGW, &oldGD); // save current port
   SetRect(&rectGW, 0, 0, (unsigned short)mapSizeX, (unsigned short)mapSizeY);
   // create 32-bit Gworld in Temporary Memory...
   err = NewGWorld(&pGWorld, 32, &rectGW, 0, 0, useTempMem);
   if (err != noErr)
      return 0;
   hPixMap = GetGWorldPixMap(pGWorld);
   pictMapAddr = (unsigned long)GetPixBaseAddr (hPixMap);
   pictRowBytes = (unsigned long)(**hPixMap).rowBytes & 0x3fff;
   // Get ready to draw offscreen...
   SetGWorld(pGWorld, nil);
   LockPixels(hPixMap);
   EraseRect(&rectGW);
   // Do it...
   DrawPicture(pict, &rectGW);
      // Allocate an area of memory for the texture
   textureMap = (unsigned long *)NewPtr(mapSizeX * mapSizeY * sizeof(unsigned long));
   if (textureMap == nil) {
      success = 0;
      goto bail;
   }
   // copy the PICT into the texture
   textureMapAddr = textureMap;
   for (row = 0L; row < mapSizeY; row++) {
      pictMap = (unsigned long *)(pictMapAddr
                              + (pictRowBytes * row));
      for (col = 0L; col < mapSizeX; col++) {
         *textureMap++ = (*pictMap++ | 0xff000000L);
      }
   }

   // Now make it into a storage pixmap...
   bMap->image = Q3MemoryStorage_New(
               (const unsigned char *)textureMapAddr,
               mapSizeX * mapSizeY * sizeof(unsigned long));

   if (bMap->image == nil) {
      success = 0;
      goto bail;
   }
   UnlockPixels(hPixMap);

   bMap->width      = mapSizeX;
   bMap->height     = mapSizeY;
   bMap->rowBytes   = bMap->width * 4;
   bMap->pixelSize  = 32;
   bMap->pixelType  = kQ3PixelTypeRGB32;
   bMap->bitOrder   = kQ3EndianBig;
   bMap->byteOrder  = kQ3EndianBig;

bail:
   SetGWorld(oldGW, oldGD);
   DisposeGWorld(pGWorld);
   if (textureMapAddr != nil)
```

```
      DisposePtr((Ptr)textureMapAddr);
   return success;
}
```

At first glance, it seems like an awful lot's going on here, but actually it's all quite straightforward. First, we need to get an offscreen drawing area (a GWorld) in which to draw our PICT so that we can have access to an honest-to-gosh PixMap. Next, we allocate some memory (via NewPtr) to hold the pixel data, and copy it over. The call to Q3MemoryStorage_New() sets the image field of the storage pixmap to our pixel data and makes QD3D copy everything into its own internal memory. (At this point, we can actually dump the temporary "spooling buffer" we created with NewPtr.) Because we requested a 32-bit-pixel GWorld earlier, we can confidently assign a value of 32 to the pixelSize field of the storage pixmap, and since we're running on a Macintosh we set the bit and byte order fields to kQ3EndianBig. All the rest is pretty self-evident.

Finally, our geometry gets "textured" when we call our routine, AddTextureToGroup(), shown in Listing 5. Here, we're converting our storage pixmap into a TQ3TextureObject, which in turn gets passed to Q3TextureShader_New() in order to yield a TQ3ShaderObject that can be added to our display group (i.e., our model, which contains our geometry). Notice that we dispose of the image field in our storage pixmap as soon as we've made a TQ3TextureObject out of it. Remember, at this point the image has already been cached by QD3D and is no longer needed in our heap. (We can still get it back, any time we want, by using GetTexture and GetPixmap API calls that are provided for just this purpose.)

If it seems like there's lots of layers of "New This" and "Get That," with lots of object-passing, etc., that's because QD3D is an elaborate, object-oriented system with many layers of abstraction - many "strata," if you will. It seems a bit bewildering at first, but once you get used to the coding idioms, it all becomes familiar quite quickly.

Listing 5: AddTextureToGroup()

```
AddTextureToGroup
TQ3Status AddTextureToGroup( TQ3GroupObject theGroup, TQ3StoragePixmap *textureImage)
{
   TQ3TextureObject   textureObject =
   Q3PixmapTexture_New(textureImage);
   // Since the local copy is no longer needed, kill it:
   Q3Object_Dispose(textureImage->image);

   if( textureObject ) {
      TQ3ShaderObject textureShader =
      Q3TextureShader_New(textureObject);
      if( textureShader ) {
         Q3Object_Dispose(textureObject);
         Q3Group_AddObject(theGroup, textureShader);
         Q3Object_Dispose(textureShader);
         return(kQ3Success);
         }
   }
   return(kQ3Failure);
}
```

Things to Try

If you've followed the discussion this far, your imagination is probably already running wild with ideas to try. But in case it isn't, let me suggest just a few possibilities:

- Instead of drawing a PICT image into an offscreen GWorld for use as a texture map, try drawing something offscreen yourself, procedurally, using ordinary (2D) Quickdraw calls such as LineTo(), InvertRect(), DrawString(), etc.
- Using the foregoing technique, show how it's possible to let the user paint in one window and have

colors appear on the 3D geometry in another window, in real time.

- Create a simple routine that loops over the TriGrid's vertices and "jitters" them in x, y, and/or z. Create a dialog, control, or menu item to let the user specify the amount of jitter (displacement noise) as a percentage of the nominal inter-vertex spacing.
- Expand on the foregoing idea by writing a function that jitters vertex coordinates in x and/or y as a function of z, where z is the elevation coordinate. This will selectively produce "noisier" terrain regions in areas of high (or low) elevation.
- Let vertex spacing be influenced by dz/dx or dz/dy (i.e., local terrain slope).
- Add jittering/dithering to vertex normals. Modify vertex normals algorithmically to produce different effects. (This is a type of "bump mapping.")
- Create an entire terrain algorithmically, using fractal mathematics. Wrap the resulting TriGrid around a sphere. Texture-map it and call it a planet.

Summary

With the addition of some code to enable a "Texture..." item in our program's File menu, we now have a fully functional texture-mapping feature in our PMB app. The user can supply any PICT image, of any size (subject to video-RAM constraints) and any aspect ratio, to "overlay" onto his or her terrain. In this way, no matter what the original colors of the terrain were, the user can apply any detailed color scheme whatsoever, for some spectacular improvements in the terrain's appearance. And the incredible part is, the terrain grid still shades in real time and can be free-rotated in fully textured form without bogging the host machine down.

We've also managed to rid our terrains of that oh-so-tiresome "polygonal" look (which is the trademark of so many bad computer 3D graphics), thanks to some very simple vertex-normal manipulations. Yet, as with texture-mapping, the performance hit here is so slight as to be negligible. You'd think that bonafide, realtime, per-vertex Gouraud shading might slow our screen-redrawing down unbearably - but in fact, by providing QD3D with pre-calculated normals, we save the renderer from having to generate its own surface normals on the fly (which it would otherwise have to do), offsetting much of the speed hit of Gouraud shading and mooting the whole render-speed issue.

As terrain-generation programs go, our little test-app certainly is no Bryce-killer. But we've laid the groundwork for an advanced elevation-grid program that - with not much extra work - could provide the basis for some very interesting game environments (particularly of the flight-simulator kind) or intriguing camera fly-throughs of scientific data - the kind of things that used to be associated with big-budget NASA projects.

Of course, it would be nice to see our little app (as fast as it is) do screen redraws even faster, so that, for example, we could look at even bigger grids and use even more elaborate texture maps, to create even more impressive 3D worlds. It turns out there's still more work to be done in the speed department. (If there's anything every 3D programmer wants more of, it's speed.) Next time, we'll take a look at ways to speed up our PMB app - and other QD3D-based applications as well. As it turns out, there's a lot to know about squeezing the last ounce of performance out of Quickdraw 3D.

Kas Thomas, tbo@earthlink.net, has been a Macintosh user since 1984 and has been programming in C on the Mac since 1989. He is the author of a QD3D-powered Photoshop® plug-in called Callisto3D (available at http://users.aol.com/callisto3d), which uses many of the techniques discussed in this article.