Volume Number: 14
Issue Number: 12
Column Tag: Programming Techniques

Poor Man's Bryce, Part III: Faster Terrains in QuickDraw 3D

by Kas Thomas

Follow a few simple tips and you're guaranteed to get better performance from QuickDraw 3D.

Apple's QuickDraw 3D library offers graphics programmers a powerful, flexible API for implementing 3D graphics in a cross-platform manner. Using QD3D, you can get interactive 3D graphics to happen on screen with relatively minimal effort. The important concept here is "interactive": From the outset, QD3D's designers wanted interactivity to be part of the 3D experience, which means the API has been painstakingly optimized for speed. Unless you're modeling truly enormous objects (i.e., with tens of thousands of polygons), almost anything you create with QD3D can be manipulated onscreen (i.e., rotated, scaled, translated) in real time, fully textured and smooth-shaded. This is far different from most raytracing environments, where you wait from a few seconds to several hours for a single screen to draw.

But in QD3D programming - as in other types of graphics programming - you can never have too much speed. It takes so much floating-point math to render a scene (even if you're just doing shadowless, reflectionless flat-shaded objects) that it's not hard at all to run into situations where there is a noticeable, annoying lag between mouse movements and screen updates, even on a G3 machine with video acceleration.

In previous articles (MacTech, October and November 1998), I showed how to put together a simple QD3D program, called PMB (for "Poor Man's Bryce"), that can convert 2D imagery to 3D "terrain" using displacement mapping. (The full Code Warrior projects are available online at ftp.mactech.com.) In Part I, we set up the code to make a height-mapped 3D grid with vertices equal (or at least proportional) in number to the number of pixels in the source image; and we presented code to render this grid in wireframe, dots, or flat-shaded mode. The terrain objects we made this way weren't particularly attractive, so in Part II we looked at some techniques for "prettying up" our terrains. We talked, for example, about how to smooth out the facets of the terrain by means of vertex-normal recalculation (giving true Gouraud shading); and we showed how to overlay our grids with PICT images (honest-to-gosh texture mapping). Surprisingly, we didn't take much of a speed hit along the way. It turns out, for example, that by precalculating our vertex normals (in preparation for Gouraud shading), we save QD3D's rendering engine from having to calculate normals on the fly, which it would otherwise do.

Still, it would be nice if our terrains rendered a bit quicker, so that large terrains (with, say, 10,000 polygons) could be swiveled, resized, etc. in snappier, more "interactive" fashion. As it is, version 2.0 of our PMB app requires 25 ticks (about four tenths of a second) to rotate an 8,000-polygon object 12 degrees around the y-axis, and display it fully updated. This is for a fully textured (with a 360-by-360-pixel texture map) TriGrid, drawn into a 480x384 window in 32-bit color, on a 250Mhz G3 machine. Mind you, that's two and a half frames per second, which for some types of work is not so bad. Still, it's a long way from being realtime-interactive.

I promised last time that there would be ways to speed up our code significantly. It's time now to deliver on that promise. In the pages to follow, we'll see how it's possible to achieve a better than five-to-one speedup of our program, with no loss of functionality. Many of the techniques we'll discuss can be applied to other QD3D programs, for similar speed gains. So fasten your seat belt, and get ready to rocket.

Choice of Geometry

Let's get right to the core of the matter and start with one of the most important speed considerations, namely choice of geometry. As mentioned in Part 1 of this series, QuickDraw 3D offers four freeform mesh primitives that can be used to represent complex objects. They include the Mesh, the Trigrid, Polyhedron, and TriMesh. (Of course, QD3D also accommodates NURB patches, but that's another story.) By way of review, here are the main distinguishing features of these geometries:

- **Mesh:** This was the original "complex geometry" that shipped with Version 1.0 of QuickDraw 3D. It's by far the most flexible geometry, because it allows you to specify non-triangular polygons, with "holes" cut in them if you desire, and you can attach any number of attributes (in any combination) to any of a mesh's vertices, edges, faces, contours, or component groupings. But precisely because of the mesh's flexibility and generality, it is the slowest-rendering freeform primitive. The mesh, by its nature, brings with it a lot of code overhead at render-time. It is the worst choice of primitive where speed is concerned.
- **TriGrid:** This type of object differs from the Mesh in that it comprises a lattice of connected triangles, with equal numbers of vertices in each row and equal numbers of vertices in each column. Efficient sharing of vertices by neighboring triangles makes the TriGrid a relatively efficient (fast-rendering) object. It happens to be well-suited to our terrain-generation task, since adjoining pixels in a 2D source image can easily be mapped to adjoining vertices in a TriGrid. For more complex modelling tasks, however, it is obviously somewhat limited, since not every 3D object lends itself to a fixed-lattice layout.
- **Polyhedron:** The Polyhedron uses a triangle list defined by indexes into a vertex list. In other words, it's essentially an arbitrary array of triangles. The triangles may share vertices, or they may not - it's totally up to the programmer. This is a more orthodox type of "freeform mesh" object, extremely flexible topologically, yet relatively efficient in terms of RAM requirements and rendering speed. Unlike the Mesh, the Polyhedron can't contain polygons with more than three vertices, nor can it contain nested "component" regions or cutouts. But the Polyhedron's fast rendering speed moots most such concerns. It's an efficient, easy-to-work-with primitive.
- **TriMesh:** The TriMesh is by far the fastest-rendering of QD3D's freeform primitives (as we'll see in a moment). Structurally, it's a lot like the Polyhedron in that it is essentially an array of triangles defined by indices into an array of points. But there are some key differences. Whereas the Polyhedron follows the QD3D tradition of allowing attributes to be individually assigned to triangles, edges, and/or vertices, as well as the whole object, the TriMesh imposes a "uniform attributes" requirement, such that if one triangle has a transparency attribute (for example), all triangles have to have a transparency attribute. That doesn't mean that you can't "nil out" the transparencies of those triangles you don't want to be transparent, but you do have to specify attribute storage for all triangles, regardless of what the attribute value is for each one.

The TriMesh is like a Polyhedron in a straightjacket. It lacks some of the flexibility of the Polyhedron (and other QD3D mesh primitives), and for that reason it is - not surprisingly - a lot better-performing. Simply put, there is very little "special case" code overhead for the TriMesh. The renderer knows in advance what to do to make the TriMesh show up onscreen. It doesn't have to stop and examine every attribute of every face and vertex, because if a given attribute type is present for one face or vertex, it's going to be present for all. Some very good rendering optimizations result.

The TriMesh is a "low level," performance-optimized object. Like many low-level tools, it's a bit harder (and less forgiving) to use than the higher-level, "programmer-centric" primitives. It takes some getting used to. But if your primary need is speed, this is one animal you'll definitely want to spend time getting to know.

Conversion Code

One strategy that's worth considering (for almost any QD3D project) is using two versions of a given object: an easy-to-code, offscreen, "working" version, and a render-time version. The behind-the-scenes "working" version of the object might be a TriGrid or Mesh, while the renderable version might be a TriMesh (for speed). All you need is a routine that can convert from one version to the other. This is what I did for Version 3.0 of our sample app, PMB (code available online). I added a menu option under the Edit menu called "Swap Geometries," and when the user chooses this item, a routine named DoTriMeshConversion() translates our TriGrid to a TriMesh (if it hasn't been made already). In this way, the user can toggle back and forth between TriGrid and TriMesh versions of the terrain very quickly.

At this point it might be a good idea to review the data structure describing a TriMesh.

```
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 code to translate our TriGrid to a TriMesh is, frankly, somewhat lengthy and ugly. It comprises a separate (new) C module in our project, TriMeshConversion.c, which contains roughly 300 lines of code divided among five routines. In terms of creating the TriMesh, the main thing to keep in mind is that it's almost always possible to "nil out" many of the TQ3TriMeshData structure's fields. For example, in PMB, we're not applying an attribute set to the overall object, hence we can set the triMeshAttributeSet field to nil. We also don't want to specify any triangle attributes, per se, nor edges (nor edge attributes), so the relevant fields are all zeroed out. (Frankly, if you start to use very many of these fields, the speed advantage of the TriMesh quickly begins to evaporate.) We do have to specify a point list, of course, as well as a triangle list consisting of indices into the point list, because this is how we describe our geometry. The TriMesh also requires that we precalculate a bounding box for the bBox field (which helps speed rendering). This is not hard to do: You simply loop over all of the points in the TriMesh and find the minimum and maximum x,y, and z coordinates of the points. It's very important to do this correctly, however, because if you don't, you'll crash your computer.

Transcribing our geometry information from TriGrid form to TriMesh form is a snap. Listing 1 shows how we copy our point list data.

Listing 1: ConvertVertices()

```
ConvertVertices()
Copy point list from trigrid into trimesh data.

void ConvertVertices( TQ3TriGridData *tgData,
             TQ3TriMeshData *tm) {
   long i;

   tm->numPoints = tgData->numColumns * tgData->numRows;
   tm->points = (TQ3Point3D *)NewPtr(sizeof(TQ3Point3D)
                                       * tm->numPoints);

   for (i = 0,tm->points != nil; i < tm->numPoints; i++)
      tm->points[i] = tgData->vertices[i].point;
}
```

The number of points is just the number of rows of the TriGrid times the number of columns. We allocate an appropriate amount of memory, then loop over all the vertices in the TriGrid, copying point information directly into the new array. A piece of cake.

It's not at all hard, either, to copy triangle data from a TriGrid to a TriMesh (or to a Polyhedron). Listing 2 shows how it is done.

Listing 2: ConvertGridTriangles()

```
ConvertGridTriangles ()
void ConvertGridTriangles( TQ3TriGridData *tgData,
               TQ3TriMeshData *tm,
               unsigned long *numTriangles)
{
   unsigned long col,row,count,numCols,numRows;
   unsigned long triangle_set = 0;
   TQ3TriMeshTriangleData *ptd;

   ptd = (TQ3TriMeshTriangleData *) NewPtr(
         (tgData->numColumns-1) *
         (tgData->numRows-1) *
         2 * sizeof( TQ3TriMeshTriangleData ) );

   if (ptd == nil) return; // failure? go back
   numCols = tgData->numColumns;

   numRows = tgData->numRows;

   for (count = row = 0; row < numRows - 1; row++)
      for (col = 0; col < numCols - 1; col++)
   {

      // we'll assume an ordering
      // of pts clockwise from upper left

      // first do 1-2-4
      ptd[ count ].pointIndices[0] =
         col + row * numCols;
      ptd[ count ].pointIndices[1] =
         col + 1 + row * numCols;
      ptd[ count ].pointIndices[2] =
         col + numCols + row * numCols;
      count++;

      // then 4-2-3
      ptd[ count ].pointIndices[0] =
         col + numCols + row * numCols;
      ptd[ count ].pointIndices[1] =
         col + 1 + row * numCols;
      ptd[ count ].pointIndices[2] =
         col + 1 + numCols + row * numCols;
      count++;


   } // end of double nested loop

   *numTriangles = count;

   tm->triangles = ptd;
}
```

Since the TriGrid is a rectilinear lattice, you can just raster through the rows of points, forming triangles as you go. (The triangles are just triples of indexes into the array of points: i.e., three point-array indices define a triangle.) In Listing 2, we follow a point-traversal scheme based on the ordering shown in **Figure 1**. You can switch hypotenuse directions within rows and/or across columns, if you want to get creative, but you should specify points in a consistently clockwise (or consistently CCW) order, if you want to avoid "flipped polygons" at render time.

Figure 1.

When triangulating the cells of a TriGrid, it's possible to go in any direction. For our TriGrid-to-TriMesh conversion routine, we chose to copy points in 1-2-4, 4-2-3 order. (Notice that a clockwise orientation is thus maintained.) In this instance, the hypotenuse runs from lower left to upper right, but it's just as easy to have the hypotenuse go from upper left to lower right. You can also vary the orientation of the hypotenuse from one cell to the next, or from one row to the next.

Attributes

The tricky part of setting up any TriMesh involves attributes. You may have noticed, back in the code for Listing 1, that a TriGrid's triangles are based on vertices (type TQ3Vertex3D), whereas a TriMesh's triangles are based on points (type TQ3Point3D). The difference is that a TQ3Vertex3D is a structure that encapsulates both a point and an attribute set, whereas points don't have attributes. With a TQ3Vertex3D, you can easily attach various attributes to particular points; this is in keeping with the object-oriented nature of QD3D. (Attributes are bound tightly to the objects they modify.) The TriMesh way of doing things, by contrast, is to specify a point list in one array, and a corresponding attribute list in a separate (but equal) array.

In our case, we happen to have a terrain object in which individual vertices may or may not have one (or more) of three kinds of attributes: vertex normals, diffuse colors, and/or "UV" (parametric mapping) coordinates. Vertex normals enable us to get a smoothed, Gouraud-shaded object (without ugly faceting). Diffuse colors give each vertex an RGB value. UV parameters are what let us do texture mapping. (For a review of these subjects, see last month's article.) Note that vertex normals are 3D vectors, whereas diffuse colors are given by three floats and UV parameters are pairs of floats.

To get our vertex attributes into a form that the TriMesh can use, we have to loop across all of the vertices in the original TriGrid and call Q3AttributeSet_Get on each vertex. This call plucks the actual data out of the attribute set for each vertex. If you specify kQ3AttributeTypeNormal as an argument, you'll get vector data back. If you specify kQ3AttributeTypeShadingUV, you'll get the TQ3Param2D for the vertex (describing its UV coordinates). But the point is, the TriMesh expects to have contiguous attribute data, which is to say, if there are 1,000 points in a mesh, and they all have vertex normals as well as UV params, you'll need to specify an array of 1,000 vectors and a second array of 1,000 UV params. If the points just have UV params but no other attribute types, then you just need to form an array of UV params.

A TriMesh expects its attribute data to be given in a particular format, as shown below:

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

The first field of this structure will be set to a predefined constant, like kQ3AttributeTypeNormal. The second field is a pointer to the array of attribute data (whether it be an array of vectors, floats, or whatever). This pointer is usually allocated dynamically; you'll seldom, if ever, have an array of 1,000 vectors (or whatever) waiting for you on the stack. This data pointer is (accordingly) something that you'll need to free up eventually, after you've successfully created your TriMesh. When QD3D gets your Q3TriMesh_New() call, it copies all of your attribute (and other) data into private, system-heap storage. After that, you no longer need to have big arrays of attribute data sitting around hogging your application's heap, so get rid of them - but not until you've called Q3TriMesh_New().

The final field of the TQ3TriMeshAttributeData, namely the attributeUseArray field, is only used in the case of custom (user-defined) attributes; most of the time you'll set this to nil.

Every type of attribute associated with any part of a TriMesh has to be specified in a TQ3TriMeshAttributeData structure. Of course, in QD3D, objects can have more than one type of attribute. In our case, we will want to be able to assign vertex normals, diffuse colors, and/or UV parameters (or all, or none of these) to our vertices. How can we do this? We do it with an array of TQ3TriMeshAttributeData structs - one for each attribute type.

So the game plan is: Fill out one TQ3TriMeshAttributeData struct for each type of attribute you need to put in the TriMesh, allocating data storage dynamically as needed. If you have more than one attribute type, make an array of TQ3TriMeshAttributeData structs. Then set the TriMesh's vertexAttributeTypes field to point to the array of TQ3TriMeshAttributeData. Got it?

I told you, the TriMesh takes getting used to.

Listing 3 shows not only how to stuff the right values into the right arrays, but also how to iterate through all of an object's attributes. Remember, in QD3D an object can have up to a dozen different attributes. In our case, we're only worried about three of the possible dozen types, but we may have to iterate a dozen times to find the three types we're interested in.

The call Q3AttributeSet_Get fetches the actual data we need (whether it's vector data, RGB data, or whatever) from the attribute set in question. Unlike Q3Geometry_GetAttributeSet, Q3AttributeSet_Get doesn't actually increase the reference count of anything, so there's no need to call Q3Object_Dispose afterward. In fact, if you do you'll run into weird errors.

Listing 3: ConvertGridAttributes()

```
ConvertGridAttributes()
long ConvertGridAttributes( TQ3TriGridData *tgData, TQ3TriMeshData *tm)
{
   static TQ3TriMeshAttributeData attribs[3];
   TQ3AttributeType theType;
   long i,k;


      // We need to iterate thru all attribute TYPES and copy those we need
      // (namely UV params and/or vertex normals) into dynamically allocated
      // arrays.

   for (i=0,
      theType = kQ3AttributeTypeNone,
      attribs[0].attributeType = nil,
      Q3AttributeSet_GetNextAttributeType(tgData-
                     >vertices[0].attributeSet,
            &theType );

      theType != kQ3AttributeTypeNone && i < 3; // loop termination

      Q3AttributeSet_GetNextAttributeType(tgData-
                     >vertices[0].attributeSet, &theType ))
   {

      attribs[i].attributeUseArray = nil;
      attribs[i].attributeType = theType;

      switch( theType ) {

         case kQ3AttributeTypeNormal :
         {
            TQ3Vector3D *vecs =
               (TQ3Vector3D *)NewPtr(sizeof(TQ3Vector3D)
                      * tm->numPoints);

            for (k = 0; k < tm->numPoints; k++)
            {
         Q3AttributeSet_Get( tgData->vertices[k].attributeSet,
                     kQ3AttributeTypeNormal,
                     &vecs[k]);
            }

            attribs[i].data = vecs;
            i++;
         }
         break;

         case kQ3AttributeTypeShadingUV :
         {
            TQ3Param2D *params =
               (TQ3Param2D *)NewPtr(sizeof(TQ3Param2D)
                       * tm->numPoints);
            for (k = 0; k < tm->numPoints; k++)
            {
               Q3AttributeSet_Get( tgData-
                     >vertices[k].attributeSet,
                     kQ3AttributeTypeShadingUV,
                     &params[k]);
            }

            attribs[i].data = params;
            i++;
         }
         break;

         case kQ3AttributeTypeDiffuseColor :
         {
            TQ3ColorRGB   *color =
               (TQ3ColorRGB *)NewPtr(sizeof(TQ3ColorRGB)
                          * tm->numPoints);
            for (k = 0; k < tm->numPoints; k++)
            {
               Q3AttributeSet_Get( tgData-
                        >vertices[k].attributeSet,
                     kQ3AttributeTypeDiffuseColor,
                     &color[k]);
            }

            attribs[i].data = color;
            i++;
         }
         break;
         default:
         break;

      } // switch

   }   // end for loop

   tm->vertexAttributeTypes = attribs;
// now point at the TQ3TriMeshAttributeData array

   return i; // return the number of attributes copied
}
```

Now at last we're ready to tackle the actual creation of a TriMesh, which is the subject of Listing 4. Our custom data structure (pointed to by theDocument), contains a model that already has our TriGrid geometry loaded in it. So the first order of business is to try to fetch the TriGrid from the model. If we succeed, the next order of business is to start filling out the fields in our TriMesh's data structure. Note that many of the fields are set to zero or nil.

In Listings 1, 2, and 3 we showed how to copy the point, triangle, and attribute data from our TriGrid into the TriMesh's data structure. (Hence the calls to ConvertVertices, ConvertGridTriangles, and ConvertGridAttributes.) Finally, we call our own routine, GetTriMeshBBox, which we haven't shown here (but is provided in the complete project online); it simply finds the minimum and maximum x, y, and z values for the furthest-apart points in the grid. Then it's time to call Q3TriMesh_New. Generally speaking, if you've messed up anything prior to now, Q3TriMesh_New will return nil. (QD3D does a lot of internal consistency checks to spot potential problems before instantiating a TriMesh. If your data fields are bad, QD3D will refuse to create the mesh.)

The rest of the code in Listing 4 is cleanup code to free arrays that have been dynamically allocated and get rid of our TriGrid reference as well as the data we retrieved from it. If you fail to make these calls, you can expect memory leaks.

Listing 4: DoTriMeshConversion()

```
DoTriMeshConversion()
Note: Error checking has largely been eliminated in the interest of clarity.

void DoTriMeshConversion( DocumentPtr theDocument ) {

   TQ3TriGridData      tgData;
   TQ3TriMeshData      trimesh;
   TQ3GeometryObject   tri;
   TQ3Status           s;
   TQ3GroupPosition    pos;
   long                i;

         // Get the first trigrid position from our group...
   s = Q3Group_GetFirstPositionOfType( theDocument->fModel,
         kQ3GeometryTypeTriGrid, &pos );
   if (s != kQ3Success) return;

         // now get the trigrid
   s = Q3Group_GetPositionObject( theDocument->fModel,
                                  pos, &tri);

   if (Q3TriGrid_GetData( tri, &tgData ) != kQ3Success)
      return;

   trimesh.triMeshAttributeSet = nil;
   trimesh.numPoints    = tgData.numColumns * tgData.numRows;
   trimesh.numEdges   = 0;
   trimesh.edges      = nil;
   trimesh.numTriangleAttributeTypes    = 0;
   trimesh.triangleAttributeTypes       = nil;
   trimesh.numEdgeAttributeTypes        = 0;
   trimesh.edgeAttributeTypes           = nil;

         // transcribe our geometry...
   ConvertVertices ( &tgData, &trimesh );
   ConvertGridTriangles( &tgData, &trimesh,&trimesh.numTriangles);

         // transcribe our attributes...
   trimesh.numVertexAttributeTypes =
      ConvertGridAttributes(&tgData, &trimesh);

   GetTriMeshBBox( &trimesh ); // calculate bounding box

         // now create the TriMesh
   if (trimesh.triangles != nil && trimesh.points != nil)
      theDocument->fAlternateGeometry = Q3TriMesh_New( &trimesh );

         // * * * * * * * * * CLEANUPS * * * * * * * * *
   if (trimesh.triangles != nil)
      DisposePtr((Ptr)trimesh.triangles);

   if (trimesh.points != nil)
      DisposePtr((Ptr)trimesh.points);

   for (i = 0; i < trimesh.numVertexAttributeTypes; i++)
      if (trimesh.vertexAttributeTypes[i].data != nil)
         DisposePtr((Ptr)trimesh.vertexAttributeTypes[i].data );

   Q3TriGrid_EmptyData( &tgData );     // free the trigrid data
   Q3Object_Dispose(tri);         // get rid of trigrid object reference
}
```

Does It Work?

Converting our terrain to a TriMesh speeds things up considerably. To benchmark the performance, I wrote a short routine, DoRenderTest, that rotates our object 360 degrees around the y-axis, in 30 increments of 12 degrees, and times the results in ticks (sixtieths of a second). On my 250Mhz G3 machine, a 4,096-vertex test object (with 7,938 polygons) takes 518 ticks to do a complete rotation as a TriGrid, but only 282 ticks to do it as a TriMesh. This is for an unsmoothed, untextured grid.

With the original 512-by-512-pixel source image applied as a texture map, again unsmoothed, the test cycle takes 773 ticks to complete in TriGrid mode but only 438 ticks for the TriMesh.

The performance comparison is even more interesting when vertex normals are included as attributes for smoothing (Gouraud shading). The textured, smoothed object takes 690 ticks to rotate as a TriGrid, but only 261 ticks as a TriMesh - a better than 2.5-to-1 speedup. Notice that adding vertex normals shaves almost 100 ticks off the TriGrid's time and more than 100 ticks off the TriMesh's time. This is because when we provide the renderer with precalculated, pre-cached vertex normals, the renderer doesn't have to calculate any surface normals on the fly.

Lesson No. 1: Always supply vertex normals, whenever you can. The normals don't have to be Gouraud-averaged, if you don't want smoothing. Just be sure to provide normals of some kind. (Code for doing this is in our project.)

Lesson No. 2: Use a TriMesh as your main "renderable" geometry, if possible.

If you apply both lessons, you should be able to see better than two-to-one speed gains on many (if not most) kinds of objects.

Less Is More

A mentor once gave me some truly excellent advice on how to get code to run faster. "The CPU can only execute so many instructions per second," he noted. "So in a sense, there is no such thing as making code go faster. There is only such a thing as making the machine do less." I've often wanted to engrave those words in bronze. To go fast, do less. Let that thought percolate through your brain whenever you try to "speed up" your code.

It might behoove us to try to make our little PMB app do less. In this respect, you may have noticed that PMB isn't terribly smart about how it allocates resources. The flat parts of our terrain, for example, get just as many vertices as the "peaky" parts, even though - clearly - the flat parts could get by on less geometry. Fixing this ought to be easy: just delete triangles in flat areas. But wait: This isn't possible with the TriGrid geometry, which relies on fixed rows and columns.

Once again, TriMesh to the rescue.

By adding a few lines of code to our TriGrid-to-TriMesh triangle translation routine (to check the "altitude" of each point before copying), we can ensure that "sea-level" triangles don't get copied over to the TriMesh. With another line or two of code, we can add a "Trim Excess" toggle to our Preferences menu. The result is that wherever our terrain has zero-elevation triangles, the TriMesh version can be made to omit those triangles (and the associated point and attribute data) altogether, which ought to speed up rendering.

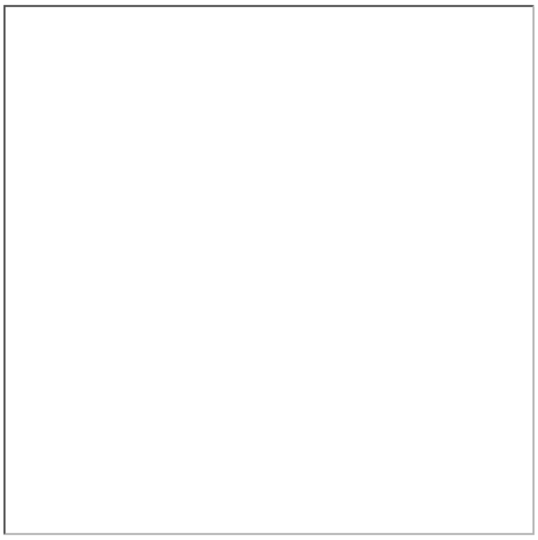

Figure 2. *Test image (scanned penny).*

Our test image (**Figure 2**) is a 512-by-512-pixel color PICT made by scanning a penny on a Microtek ScanMaker II-XE. You'll notice that the edges of the source image are white, which means lots of "edge" triangles can be eliminated. Sure enough, when we use the "Trim Excess" option in our app (see **Figures 3** and **4**), we find it's possible to eliminate over 3,000 polygons (out of 7,938) - which, of course, gives a welcome speed boost. In the rotation test, the spin time drops, in one instance, from 636 ticks for the TriGrid version to 182 for the trimmed TriMesh. (That's with a smoothed, textured object.) The untrimmed TriMesh turns in times in the 260-ticks area, so the speedup is apparently comparable to the reduction in polygon count (as you'd expect).

Figure 3. *Penny as terrain.*

There is much more to do in the "geometry optimization" department, if you think about it. After all, not only are triangles in flat spots (areas of zero slope) superfluous, but triangles in large areas of constant slope are redundant as well. With not much work, one could write a routine that loops over triangles in the mesh, examining all of a triangle's neighbors for coplanarity, the idea being that adjacent, coplanar polygons should be merged. Of course, this only works if the adjoining triangles are, in fact, mergeable: i.e., right triangles joined on a "small" edge (not the hypoteneuse). If you've done a "lazy rasterization" of triangles into the TriMesh - as I've done in our project (to keep the code small) - then none of the triangles will be mergeable, because they're all oriented the same way. (Merging any two of them will give a parallelogram.) Awhile ago, I mentioned that if you were creative, you could alter the hypoteneuse orientation of triangles as you copied them into the TriMesh. Now you know why. (Even so, not all triangle combos will be eligible for merging. Prove that, at most, two thirds will be.)

Figure 4. *Trimmed geometry.*

To reduce the triangle count of our grid to some kind of theoretical minimum requires that we go beyond mere polygon decimation and start over, using an adaptive sampling technique - that is, a technique that puts more vertices in areas of rapid slope change and fewer vertices in areas of less-rapid slope change. One possibility here is to develop an "area operator" (2D convolution kernel) that is sensitive to areas of high standard deviation of pixel luminance in the original image. Or you could simply look at the difference in pixel intensity between a central pixel and its north, south, east, and west neighbors. If north minus "center" equals "center" minus south, and east minus "center" equals "center" minus west, then by definition the pixel is in an area of zero slope change and needn't be made into a grid vertex.

Using an adaptive sampling technique of one sort of another will give you a point list containing vertices heavily concentrated in areas of rapid slope change and less concentrated in constant-slope areas. The next trick is to convert this vertex assortment into triangles. Triangulation of an arbitrary point list is a classical problem in 3D geometry and is harder, frankly, than it first seems. One difficulty you run into if you start connecting points indiscriminately is that you quickly end up with lots of long, skinny (scalene) triangles that don't really represent the underlying topography very well. (A "raster-line" approach gives notably poor results.) If you're interested in pursuing this subject further, search on the World Wide Web using "Delaunay" as a keyword. Also, be sure to consult the various Graphics Gems volumes published by AP Professional. (Look in the index under "triangulation" and "tesselation.")

Tweaks

From this point on, we're looking at relatively minor "tweaks" to improve performance. One that deserves mention involves monitor depth and pixel size. As simple as it sounds, a 32-bit texture uses twice the memory of a 16-bit texture, so when you're using texture maps you should draw them into 16-bit GWorlds and create 16-bit storage pixmaps. You should also set your monitor to 16-bit color mode if you want to see things render quickly. In general, when you can cut the video-byte traffic by half, you'll see a certain amount of performance improvement, even on a machine with special video hardware. The performance improvement may not be much, but it will be there.

In writing this article, I had a chance to experiment with changing all of the texture-map code to reflect 16-bit pixels (versus 32-bit), and in testing I saw a 5% speed improvement with 16-bit-textured objects running on a monitor set to "thousands of colors," versus the same objects with 32-bit textures and 32-bit monitor depth. Admittedly, this is not a terribly important performance gain. A better reason to use 16-bit textures is that you won't run out of VRAM so quickly when using big textures.

Incidentally, almost all 3D accelerator cards require that texture maps have pixel dimensions that are a power of 2 (such as 128, 256, 512, etc.), and in some cases you'll get a performance increase if you stick with square images (128-by-128 instead of, say, 128-by-256). It's also a good idea to render into windows that are aligned on a 32-byte boundary (the size of a PowerPC cache line) and try for window dimensions that are multiples of 32.

Homegrown Math Routines

If you've got a 3D application or subroutine that does a lot of matrix or vector math, you'll want to consider hand-writing your own math functions rather than using QD3D's extensive built-in math library. In the bad old days of CISC chips with very little cache, no floating-point unit, etc., you had little choice but to roll your own math routines. Today, it's not such a pressing necessity. With the advent of the PowerPC architecture (with its big instruction cache, onboard floating-point unit, and abundance of registers), function calls don't have to mean lots of overhead - most compilers nowadays pass arguments in registers rather than on the stack - but just the same, you'll probably find that it pays to inline some math routines of your own now and then, particularly in tight loops that handle thousands of vertices or polygons at a time. It's doubtful you'll actually want to drop down into assembly language, for a variety of reasons. But you may find that using your own matrix-math routines can spare QD3D from having to reload matrices over and over inside a loop, for example. Some of the QD3D math routines have error-checking overhead that (if you're careful) you can profitably sidestep by using your own routines.

One trick I've often found useful is to eliminate square roots whenever possible. Sometimes you're just comparing the lengths of two line segments, for example, in which case you don't need to take the square root of the sum of the squares. Just compare the squares. You'll get the same result. (A longer distance, squared, is always longer than a shorter distance, squared.)

Other QD3D Tricks

There's a little-known QD3D call that (in theory) can affect the speed with which textured objects are drawn. It's called Q3InteractiveRender_SetRAVETextureFilter(). There are 3 predefined constants that can be passed in this call. If you look in Rave.h you'll find them:

```
#define kQATextureFilter_Fast    0
#define kQATextureFilter_Mid     1
#define kQATextureFilter_Best    2
```

The idea here is that you can control the degree to which the renderer tries to subpixel-filter your textures when objects are close to the camera. If you don't mind a "chunky" look (i.e., you'd rather have faster rendering), you can select kQATextureFilter_Fast and have faster rendering at the expense of not-as-pretty texturing. According to Apple's QD3D development team, this scheme isn't actually implemented in QD3D 1.5.4, but will be in a future release. So be ready for it.

If you have a "busy" scene with lots of objects or you do a lot of camera fly-throughs, you should probably consider doing your own object culling. Culling refers to the process of determining which objects, if any, in a scene are outside the viewing range of the camera and therefore needn't be submitted to the renderer. QD3D's interactive renderer does its own culling, but not very efficiently. Your application (unlike the renderer) may "know" a lot about the scene, a priori, and as a result you can often reduce the number of objects submitted for rendering. (Remember: To go fast, do less.)

For more ideas on how to speed up QD3D, be sure and track down the excellent document, "Making Cool QuickDraw 3D Applications" by Apple's Brian Greenstone. This document has changed locations a lot but is on Apple's web site. (Search on "cool3dappspdf.") It contains lots of tips for making QD3D objects render faster.

Conclusion

QuickDraw 3D lives up to its name in most situations, but in QD3D programming (as in all graphics work) there's no such thing as too much speed. In our sample application, we saw that by means of a few relatively simple changes to our code (if you can call translating a TriGrid to a TriMesh simple) we were able to score around a 5-to-1 performance gain, taking a couple-frames-per-second update rate to well over 10 fps on a 250Mhz machine. (In its final version, our test object completed its 30-frame test cycle in less than 150 ticks - versus more than 750 ticks for the worst-case scenario.)

Some of the lessons we learned were:

- Always use the TriMesh data structure where speed is the main concern. This is generally good for at least a two-to-one speedup.
- Reduce the geometry to the absolute minimum number of points and polygons needed to get the job done. We found that simply trimming the edges of our geometry eliminated large numbers of unneeded vertices, giving a corresponding speed boost.
- Apply vertex-normal attributes whenever you can. Supplying precalculated vertex normals keeps the renderer from having to calculate its own normals on the fly.
- Use 16-bit texture maps and 16-bit monitor mode to reduce VRAM traffic.
- Size textures to a power of two on each dimension.
- Size windows to a multiple of 32 pixels in each dimension.
- If any "tight loops" use vector or matrix math routines, write your own routines inline rather than calling out to the QD3D math library.
- In a multi-object scene where some objects are bound to be out of viewing range, do your own culling. Don't just submit all objects to the renderer. The renderer's culling routine is seldom going

to be as fast as one you come up with.
- Most of all: To go fast, do less. The biggest speed gains of all sometimes come when you can figure out a way not to have to calculate something at all.

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