Topic 2:

QuickDraw 3D Optimizations

WHY WE NEED OPTIMIZATIONS

There is a lot going on under QuickDraw 3D's hood and much of what goes on can bring your processor to its knees if you're not careful. This chapter is going to discuss a long list of optimizations, which you should use in your QuickDraw 3D code to get the ultimate in performance. In the previous chapter I talked about the TriMesh geometry whose use is required in order to get great rendering performance. That was just the start. Now we get into some meaty stuff.

OBJECT CULLING

Object culling is the process of eliminating entire geometries from the rendering pipeline since they are known to be completely out of camera view. QuickDraw 3D performs object culling on each TriMesh submitted. Most 3D applications can improve upon this culling scheme since the "nature" of the geometries is known to the application. In other words, knowing information about the model can help us cull more efficiently than QuickDraw 3D can since QuickDraw 3D knows very little about the functionality of our application.

QUICKDRAW 3D'S CULLING SCHEME

There are two reasons why QuickDraw 3D does not cull objects very efficiently for most applications.

1. QuickDraw 3D uses bounding boxes to perform culling tests, and bounding boxes require 8 transforms to do this.

2. A model of an airplane which is made of 5 separate TriMeshes will require 5 separate culling tests (each test requiring 8 transforms). This would be a total of 40 transforms to cull-test the object.

Remember, however, that even though this scheme may seem inefficient, it is the only practical way to perform culling in an arbitrary 3D engine like QuickDraw 3D. This scheme works in all situations and makes it difficult for the programmer to screw it up.

HOW WE CAN DO BETTER

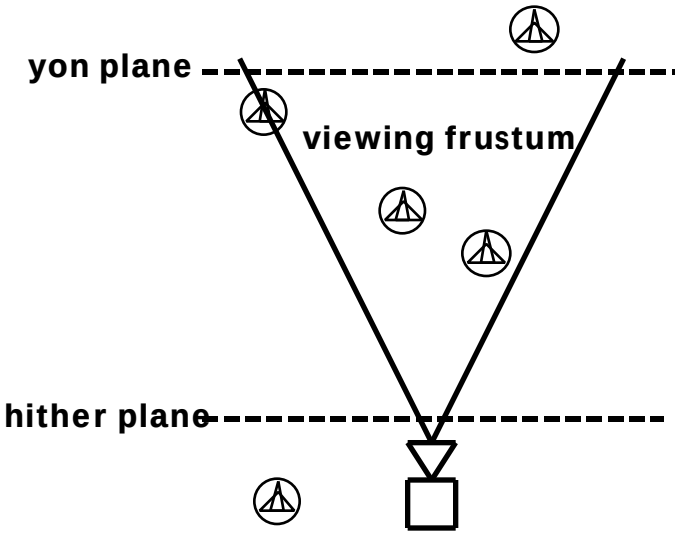
With a little optimization I'm about to show you, we can cull the above-described airplane in only 2 transforms instead of 40. Even if we have a very complex airplane made up of 100 different TriMeshes, we can still cull the model with only 2 transforms; not the 800 that QuickDraw 3D would require. When I was writing Weekend Warrior for Bungie Software, I got a 22% speed boost when I wrote my own model culling function which was much more efficient than QuickDraw 3D at removing models outside of the camera's view. Other projects of mine have seen a 35% speedup when manual culling is implemented.

What makes my method of culling so efficient is that I cull entire models, not individual TriMeshes. We don't want to cull-test each wing and fin of the airplane, but rather the entire airplane all at once. Secondly, we are going to use bounding sphere culling instead of bounding box culling. Rather than having 8 points to represent a bound box, we only need an origin and a radius to define a bounding sphere.

Spherical Culling

The idea behind spherical culling is simple. This diagram shows how it works.

Figure 2.0



A top view of our world which shows the placement of several airplanes, the camera, viewing frustum, and hither and yon planes.

What needs to be determined is whether a model's bounding sphere is outside of the viewing frustum or inside it. Models which are outside of the viewing frustum are culled since they cannot be seen. The steps to determine this are as follows:

1. Transform the bounding sphere's origin to view-space coordinates.

2. If the front of the bounding sphere is beyond the yon plane then cull the model.

3. If the back of the bounding sphere is in front of the hither plane then cull the model.

4. Transform the bounding sphere's origin and radius by the view to frustum matrix.

5. If the bounding sphere's right side is off the left edge of the viewing frustum then cull the model.

6. If the bounding sphere's left side is off the right edge of the viewing frustum then cull the model.

7. If you've made it this far the model is visible.

Spherical Culling Code

First, we need a function which will calculate the two matrices needed to do the culling. This function needs to be called every time the position or orientation of the camera changes.

```
TQ3Matrix4x4  gCameraWorldToViewMatrix;
TQ3Matrix4x4  gCameraViewToFrustumMatrix;

/**************** GET CAMERA MATRIX INFO ******************/
//
// Gets a copy of the World->View and View->Frustum matricies
// for the current camera.
//

void GetCameraMatrixInfo(TQ3CameraObject theCamera)
{
  Q3Camera_GetWorldToView(theCamera, &gCameraWorldToViewMatrix);
  Q3Camera_GetViewToFrustum(theCamera, &gCameraViewToFrustumMatrix);
}
```

Next is the code which does the culling:

```
/**************** CULL BOUNDING SPHERE ******************/
//
// Returns true if bounding sphere is not in camera's cone of vision.
//
// INPUT:   origin = world coords of the origin of the bounding sphere
//          radius = radius of the bounding sphere.
//

Boolean CullBoundingSphere(TQ3Point3D *origin, float radius)
{
float         radius,w1,w2;
float         rx,ry,px,py;
```

```
TQ3Point3D       points[2];              // [0] = point, [1] = radius
TQ3RationalPoint4D outPoint4D[2];

      /* TRANSFORM ORIGIN TO VIEW SPACE */

  Q3Point3D_Transform(origin, &gCameraWorldToViewMatrix, &points[0]);


        /* SEE IF ORIGIN IS BEHIND CAMERA */

  if (points[0].z >= -HITHER_DISTANCE)
  {
        /* SEE IF ENTIRELY BEHIND CAMERA */

    if ((points[0].z - radius) > -HITHER_DISTANCE)
      return(true);

        /* PARTIALLY BEHIND, SO MOVE IN FRONT OF HITHER PLANE */

    points[0].z -= radius;
  }
  else
  {
        /* SEE IF BEYOND YON PLANE */

    if ((points[0].z + radius) < (-YON_DISTANCE))
      return(true);
  }

      /*************************/
      /* SEE IF WITHIN FRUSTUM */
      /*************************/

    /* TRANSFORM ORIGIN & RADIUS BY FRUSTUM MATRIX */

  points[1].x = points[1].y = radius;
  points[1].z = points[0].z;

  Q3Point3D_To4DTransformArray(&points[0],&gCameraViewToFrustumMatrix,
                 &outPoint4D[0], 2, sizeof(TQ3Point3D),
                 sizeof(TQ3RationalPoint4D));


      /* SEE IF LEFT & RIGHT SIDES ARE IN FRUSTUM */

  w1 = outPoint4D[0].w;
  w2 = outPoint4D[1].w;

  px = w1*outPoint4D[0].x;
  py = w1*outPoint4D[0].y;
  rx = w2*outPoint4D[1].x;
  ry = w2*outPoint4D[1].y;

  if ((px + rx) < -1.0f)    // see if off left side of frustum
    return(true);
  if ((px - rx) > 1.0f)     // see if off right side of frustum
    return(true);
```

```
  if ((py + ry) < -1.0f)     // see if off bottom side of frustum
    return(true);
  if ((py - ry) > 1.0f)      // see if off bottom side of frustum
    return(true);

    /* IT'S VISIBLE, SO DONT CULL IT */

  return(false);
}
```

Over 50% of the models in an average scene will be culled in the first part of the above function. Statistically speaking, 50% of the models in a scene are in front of the camera and 50% are in back of the camera, therefore, our initial check to see if the object is behind the camera or too far in front to be visible should quickly cull over half of the models in the scene. In cases where the universe is very large and your hither/yon values are relatively small, this part of the function could cull 80-90% of the models.

Now we need to determine if the models are off the top, bottom, left, or right sides of the viewing frustum, so we transform the origin and radius into frustum-space. If the model's bounding sphere is completely out of the viewing frustum then we don't draw the model because it isn't visible by the camera.

The above code works but it could be faster. To really get the maximum performance out of a culling function like this, you should do all of the culling in a single loop rather than one model at a time (like the above code did). This way you can preload the matrices into registers and process all of the individual models quickly using custom transform code rather than QuickDraw 3D's functions.

Here is some highly optimized code showing how to cull-test all of the models in a linked list pointed to by `gFirstNodePtr`.

```
/************ CULL MODELS IN LINKED LIST ************/
//
// Checks every model in a linked list to see if it
// is in the viewing frustum
//

void CullModelsInLinkedList(void)
{
```

```
float           radius,w,w2;
float           rx,ry,px,py,pz;
ObjNode         *theNode;          // ObjNode is linked list node type
register float  n00,n01,n02;
register float  n10,n11,n12;
register float  n20,n21,n22;
register float  n30,n31,n32;
float           m00,m01,m02,m03;
float           m10,m11,m12,m13;
float           m20,m21,m22,m23;
float           m30,m31,m32,m33;
float           worldX,worldY,worldZ;
float           hither,yon;

  theNode = gFirstNodePtr;          // get & verify 1st node in list
  if (theNode == nil)
     return;

           /* PRELOAD WORLD -> VIEW MATRIX */

  n00 = gCameraWorldToViewMatrix.value[0][0];
  n01 = gCameraWorldToViewMatrix.value[0][1];
  n02 = gCameraWorldToViewMatrix.value[0][2];
  n10 = gCameraWorldToViewMatrix.value[1][0];
  n11 = gCameraWorldToViewMatrix.value[1][1];
  n12 = gCameraWorldToViewMatrix.value[1][2];
  n20 = gCameraWorldToViewMatrix.value[2][0];
  n21 = gCameraWorldToViewMatrix.value[2][1];
  n22 = gCameraWorldToViewMatrix.value[2][2];
  n30 = gCameraWorldToViewMatrix.value[3][0];
  n31 = gCameraWorldToViewMatrix.value[3][1];
  n32 = gCameraWorldToViewMatrix.value[3][2];


           /* PRELOAD VIEW -> FRUSTUM MATRIX */

  m00 = gCameraViewToFrustumMatrix.value[0][0];
  m01 = gCameraViewToFrustumMatrix.value[0][1];
  m02 = gCameraViewToFrustumMatrix.value[0][2];
  m03 = gCameraViewToFrustumMatrix.value[0][3];
  m10 = gCameraViewToFrustumMatrix.value[1][0];
  m11 = gCameraViewToFrustumMatrix.value[1][1];
  m12 = gCameraViewToFrustumMatrix.value[1][2];
  m13 = gCameraViewToFrustumMatrix.value[1][3];
  m20 = gCameraViewToFrustumMatrix.value[2][0];
  m21 = gCameraViewToFrustumMatrix.value[2][1];
  m22 = gCameraViewToFrustumMatrix.value[2][2];
  m23 = gCameraViewToFrustumMatrix.value[2][3];
  m30 = gCameraViewToFrustumMatrix.value[3][0];
  m31 = gCameraViewToFrustumMatrix.value[3][1];
  m32 = gCameraViewToFrustumMatrix.value[3][2];
  m33 = gCameraViewToFrustumMatrix.value[3][3];


  hither = -HITHER_DISTANCE;             // preload into registers
  yon = -YON_DISTANCE;
```

```
        /* PROCESS EACH NODE/MODEL IN LINKED LIST */

do
{
   radius = theNode->Radius;            // get radius of model


     /*********************************/
     /* TRANSFORM ORIGIN TO VIEW-SPACE */
     /*********************************/

          /* CALC WORLD Z */

   px = theNode->Coord.x;               // get coord
   py = theNode->Coord.y;
   pz = theNode->Coord.z;
   worldZ = (n02*px) + (n12*py) +       // transform to view-space
           (n22*pz) + n32;

          /* SEE IF BEHIND CAMERA */

   if (worldZ >= hither)
   {
     if ((worldZ - radius) > hither)    // entirely behind camera?
       goto draw_off;

          /* ONLY PARTIALLY BEHIND */

     worldZ -= radius;                  // move edge over hither plane
   }
   else
   {
        /* SEE IF BEYOND YON PLANE */

     if ((worldZ + radius) < yon)       // too far away?
       goto draw_off;
   }

        /* CALC VIEW X & Y COORDS */

   worldX = (n00*px) + (n10*py) + (n20*pz) + n30;
   worldY = (n01*px) + (n11*py) + (n21*pz) + n31;



     /*************************/
     /* SEE IF WITHIN FRUSTUM */
     /*************************/

     /* TRANSFORM VIEW COORD & RADIUS TO FRUSTUM-SPACE */

   w = (m03*worldX) + (m13*worldY) +          // transform origin x
        (m23*worldZ) + m33;
   px = ((m00*worldX) + (m10*worldY) +
        (m20*worldZ) + m30) * w;

   w2 = (m03*radius) + (m13*radius) +         // transform radius x
        (m23*worldZ) + m33;
```

```
     rx = ((m00*radius) + (m10*radius) +
          (m20*worldZ) + m30) * w2;

     if ((px + rx) < -1.0f)                    // is off left?
       goto draw_off;
     if ((px - rx) > 1.0f)                     // is off right?
       goto draw_off;


     py = ((m01*worldX) + (m11*worldY) +       // transform origin y
          (m21*worldZ) + m31) * w;
     ry = ((m01*radius) + (m11*radius) +       // transform radius y
          (m21*worldZ) + m31) * w2;

     if ((py + ry) < -1.0f)                    // is off bottom?
       goto draw_off;
     if ((py - ry) > 1.0f)                     // is off top?
       goto draw_off;

       /* IT'S IN THE FRUSTUM */

     theNode->CanDraw = true
     goto next;

       /* IT'S NOT IN THE FRUSTUM */

     theNode->CanDraw = false;

          /* NEXT NODE IN LINKED LIST */
next:
     theNode = theNode->NextNode;
   }
   while (theNode != nil);
}
```

It Ain't Quite Perfect

I guarantee that manually culling models with bounding spheres will give you incredible speed boosts in your 3D applications. I got a 22-30% speed boost in Weekend Warrior and Nanosaur when I adopted this method. There are, however, a few issues to be aware of. The reason QuickDraw 3D uses bounding boxes rather than bounding spheres is because you can apply a transform to a bounding box to change it's shape, but you cannot easily do the same to a bounding sphere.

Suppose your model is being scaled on the y-axis. This scaling effect will cause the bounding box to also scale equally. A bounding sphere cannot stretch along any particular axis - it must scale uniformly.

Recalculating the bounding sphere radius when you arbitrarily scale a model can be tricky and lessens the accuracy of the bounding sphere. If, however, you are applying uniform scaling to an object (meaning the same scale on the x, y, and z axes), then you can simply multiply the bounding sphere's radius by that scale amount. Since QuickDraw 3D is a general purpose API and it has to assume that you might apply non-uniform scaling or even shearing to a model, it must use bounding boxes to perform its object culling.

The other problem is that even though we are able to cull ~80% of the models in the scene using our spherical culling function, the remaining 20% are going to get re-culled by QuickDraw 3D. Unfortunately, there is no way to tell QuickDraw 3D that we have already performed the culling tests and that these models are visible. Also, QuickDraw 3D uses its culling test to determine whether a model's triangles need to be clipped or not. If a model is entirely within the viewing frustum then there's no need to clip any triangles, but if the model straddling the edge of the viewing frustum then some of the triangles will probably need to be clipped.

You may remember from earlier that I said that you should always break up complex TriMeshes into separate TriMeshes such that there is only one material per TriMesh. This is still true, however, realize that creating additional TriMeshes causes more culling tests to be performed in QuickDraw 3D. Nonetheless, it is still more efficient to have lots of separate TriMeshes.

So, the bottom line is that we can substantially increase our QuickDraw 3D application's performance by doing our own object culling, but we will always be subject to at least some of the overhead caused by QuickDraw 3D doing it's own culling on any models which remain. Luckily, however, we can cull entire groups of objects whereas QuickDraw 3D culls on a TriMesh by TriMesh basis.

TEXTURE OPTIMIZATIONS

Texture mapped triangles take longer to transform and render than colored triangles. Uploading a texture to VRAM takes time, drawing a textured triangle takes time, clipping a triangle's u/v coordinates and filling out larger data structures takes time. Being careful about how you use textures in a TriMeshes can make a big difference in the performance of your 3D application.

SIZE MATTERS

The first thing to know about texture maps is that size does matter, and it matters in several different ways:

Powers of 2

Almost all 3D accelerator cards require that texture maps be a power of 2 in dimension. This means that a texture may be 16x16, 32x128, 256x64, etc. QuickDraw 3D, on the other hand, does not force you to use textures of this size; you can use whatever size textures you want. The problem is that QuickDraw 3D has to shrink your texture map down to the nearest power of 2 before it can upload it to the 3D accelerator card. This obviously takes time, especially if it is a large texture, but it also mangles your texture. You are far better off shrinking your textures to powers of 2 in PhotoShop than you are letting QuickDraw 3D shrink them for you on-the-fly.

There are a few 3D cards which require square powers of 2 which means that the textures must be square, not rectangular. Luckily, these cards are rare and personally I wouldn't worry about them.

Bit-Depth

When you create your texture maps in PhotoShop, Painter, or some other application, you're probably working with 32-bit pixels. That's

great, but make sure you knock the textures down to 16-bit pixels before you apply them to a QuickDraw 3D model. In a 3D scene with light sources, trilinear mipmapping, etc., you can hardly ever tell the difference between a 32-bit and 16-bit texture map. The only difference you might see is that things run faster with a 16-bit texture.

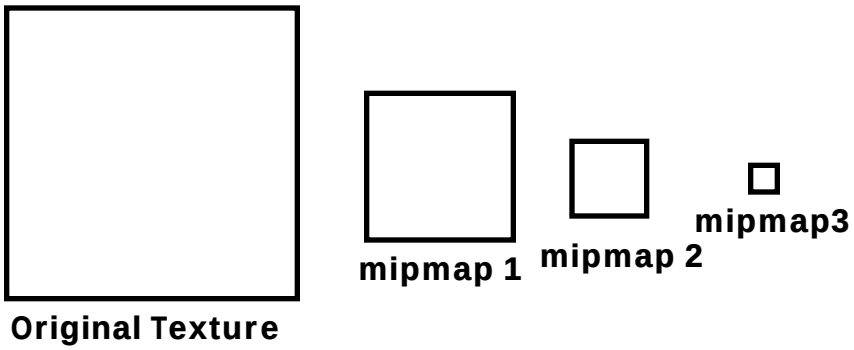
A 32-bit texture uses more VRAM than a 16-bit texture, and anything that's bigger means slower. The more memory that gets touched by the hardware, the slower it runs. That general rule can be applied to anything in your code.

The only excuse for using a 32-bit texture in QuickDraw 3D is that you need the 8 bits of alpha in the alpha channel of the texture. A 16-bit texture only has 1 bit of alpha which may not be sufficient for your particular application. In Nanosaur, there is only one 32-bit texture: the shadow. I used a 32-bit texture to do nice alpha blending of the shadow and the ground, but that's the only one – everything else is 16-bit.

MIPMAPS VS. PIXMAPS

In case you don't know what a mipmap is, here's the gist of it. A mipmap is a shrunken copy of a texture map which is used when a model is far away from the camera. A texture can have several mipmaps which get smaller and smaller. As the model gets farther away from the camera, the smaller mipmaps get used. This can sometimes create a nice blurring effect as the object moves farther away, but other times it just looks muddy.

Figure 2.1



Mipmaps are smaller copies of the original texture map

Ready for something completely confusing? Good, now get this: Before QuickDraw 3D 1.5, textures were always stored as Pixmaps (`TQ3Pixmap`). Pixmaps always generated mipmaps of the texture and you had no choice but to see your models rendered with mipmaps. As of QuickDraw 3D 1.5, we can now encase our textures in Mipmaps (`TQ3Mipmap`). Whenever we don't want our model to be rendered with mipmaps, we use the Mipmap instead of the Pixmap. Sounds completely backward, doesn't it? Use a Mipmap if we don't want mipmaps? Well, it may seem completely confusing at first, but there's a very valid reason for this confusion.

The new Mipmap structure lets you, the programmer, create your own mipmaps to use when rendering the model. The old Pixmaps told QuickDraw 3D to generate the mipmaps automatically. The nice thing about Mipmaps is that in addition to being able to assign your own mipmapped textures, you can also choose how many mipmaps to use. If you don't want any mipmaps at all, just the original texture, then you only apply one mipmap to the Mipmap object: the original texture. See, it really does make sense after all!

The following function shows how we can take the image in a GWorld and turn it into a Mipmap which contains only one texture:

```
/******************** GWORLD TO MIPMAP ********************/
//
```

```
// Creates a mipmap from an existing GWorld
//
// NOTE: Assumes that GWorld is 16bit!!!!
//
// INPUT: pGWorld = pointer to the gworld
//        mipmap = pointer to Mipmap structure to fill out.
//
// OUTPUT: mipmap = new mipmap holding texture image
//

void MyGWorldToMipmap(GWorldPtr pGWorld, TQ3Mipmap *mipmap)
{
unsigned long      pictMapAddr;
PixMapHandle       hPixMap;
unsigned long      pictRowBytes;
long          width, height;
short            depth;

          /* GET GWORLD INFO */

  hPixMap = GetGWorldPixMap(pGWorld);
  depth = (**hPixMap).pixelSize;

  pictMapAddr = (unsigned long)GetPixBaseAddr(hPixMap);
  pictRowBytes = (unsigned long)(**hPixMap).rowBytes & 0x3fff;
  width = ((**hPixMap).bounds.right - (**hPixMap).bounds.left);
  height = ((**hPixMap).bounds.bottom - (**hPixMap).bounds.top);


       /* MAKE MIPMAP */

  mipmap->image = Q3MemoryStorage_New((unsigned char *) pictMapAddr,
                                      pictRowBytes * height);
  if (mipmap->image == nil)
    DoError("\pQ3MemoryStorage_New Failed!");

  mipmap->useMipmapping = kQ3False;
  if (depth == 16)
    mipmap->pixelType = kQ3PixelTypeRGB16;
  else
    mipmap->pixelType = kQ3PixelTypeRGB32;

  mipmap->bitOrder = kQ3EndianBig;
  mipmap->byteOrder = kQ3EndianBig;
  mipmap->reserved = nil;
  mipmap->mipmaps[0].width = width;
  mipmap->mipmaps[0].height = height;
  mipmap->mipmaps[0].rowBytes = pictRowBytes;
  mipmap->mipmaps[0].offset = 0;
}
```

In the above code, we are really just filling out the `TQ3Mipmap` data structure with the needed information. Note that there is a parameter called `useMipmapping`. Make sure you set this to false if you don't really want to use mipmaps. The last few lines effectively setup mipmap #0 which is the only texture map in this Mipmap.

It's fairly simple stuff and it's not really much different than setting up an old Pixmap structure. The difference is that your textures will now use only half as much VRAM, usually look better, and put less computing burden on QuickDraw 3D.

There is, however, one thing going for using multiple mipmaps in a texture. Smaller textures render faster - we learned that a few pages ago. Having mipmaps in certain cases will increase the performance of the 3D hardware. This really only applies to large textures. If you have a 256x256 texture map, but your object is so far away that its only 50 pixels wide on the screen, then you really should be using mipmaps because the hardware will be able to draw the triangles a little faster when the texture is smaller.

TEXTURE QUALITY

Also new in QuickDraw 3D 1.5 is the ability to set the texture rendering quality. Actually, the function call exists, but doesn't actually work yet. Nonetheless, the next version of QuickDraw 3D will fix the problem, therefore, we should discuss it.

The function which is supposed to let you set the texture quality is:

```
TQ3Status Q3InteractiveRenderer_SetRAVETextureFilter(TQ3RendererObject,
                                                     RAVEtextureFilterValue);
```

You simply pass in a reference to the current renderer object and a texture filtering value which is defined in `Rave.h`:

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

The meanings of these values is up to your particular 3D hardware to decide, but the general rule is that `kQATextureFilter_Fast` means that no filtering is done to the texture for maximum performance, `kQATextureFilter_Mid` means that bi-linear or tri-linear filtering is

performed on the texture when the texture is close to the camera, and `kQATextureFilter_Best` means that bi-linear or tri-linear filtering is always performed on the textures.

Needless to say, the better the quality, the longer it takes to render. For maximum performance, always set the filter mode to `kQATextureFilter_Fast`, but `kQATextureFilter_Mid` will give you much nicer looking images.

DRAW CONTEXT OPTIMIZATIONS

Rendering speed is dictated mainly by what you are rendering, but it is also affected by where you are rendering.

BIT-DEPTH

Once again, bit-depth comes into play. This is so obvious that it's hardly worth mentioning, but here goes... Rendering performance will be substantially increased if your monitor is set to 16-bits per pixel (thousands of colors) versus having it set to 32-bits (millions of colors).

ALIGNMENT & WIDTH

For optimal performance with 3D accelerators, it's best to be rendering into a window which is on a 32 byte boundary (the size of a PowerPC cache line) and is some multiple of 32 bytes wide. If you are rendering into a window which the user can arbitrarily move around and resize, then you really don't have any control over this, but if your application takes over the entire screen or has static windows, then try to adhere to the 32-byte rule. It'll improve rendering performance by some small, probably unnoticeable amount.

Note that if you are rendering in 16-bits per pixel video mode then 32-bytes is only 16 pixels, therefore, you should place your 3D

window on 16 pixel boundaries and make them multiples of 16 pixels wide.

If your application cannot guarantee 32 byte alignment, then you should at least attempt 8-byte alignment and width. Keeping the view bounds at an 8-byte boundary will ensure that blitting from the back-buffer to the front buffer will be aligned such that the PowerPC can use its floating point double registers to copy the pixels. This is generally 2x faster than using other methods to copy the pixels.

OPTIMIZING GROUPS

Group Objects are great! They make organizing geometries, transforms, and attributes really easy. The only problem with groups is that they are a hierarchical system which has performance issues to be aware of.

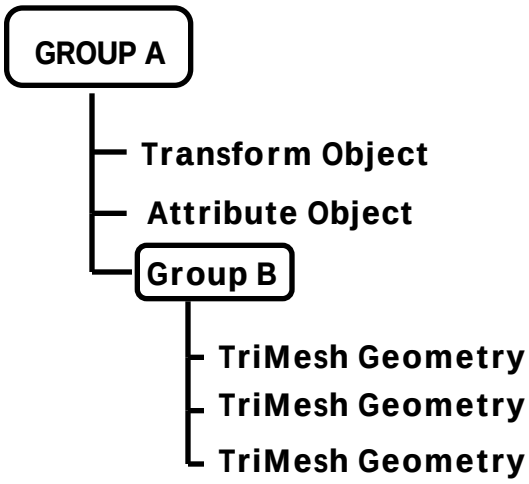
HIERARCHIES & STATE INFORMATION

Each time QuickDraw 3D encounters a group object, it pushes a lot of "state information" onto a stack before traversing into that group. When the objects inside the group have been processed (including any sub-groups), QuickDraw 3D must pop that state information back off the stack.

The state information which QuickDraw 3D pushes onto the stack can consist of anything from geometry attributes, to rendering information, to the currently active matrix. The amount of data needed to be pushed and popped is significant and you really want to avoid it whenever possible.

For example, see what happens when the following group hierarchy is submitted to QuickDraw 3D:

Figure 2.2



1. QuickDraw 3D sees that it has encountered a group object (Group A), so it saves all of the current state information on the stack.

2. The Transform and Attribute objects are processed which effectively change the current "state".

3. Group B is encountered which once again causes the current state information to be pushed onto the stack.

4. Each TriMesh inside Group B is processed.

5. Group B is done, so we pop the state information off of the stack.

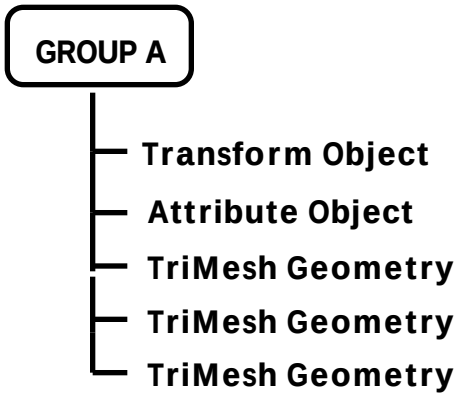
6. Group A is also done, so we pop the state information off the stack again.

7. We have now traversed through Group A's hierarchy and the all of QuickDraw 3D's state information has been preserved.

OPTIMIZING THE HIERARCHY

As you can see from the above example, having hierarchical group objects can be expensive, therefore, you should avoid them whenever possible. We should remove the TriMeshes from the unnecessary Group B and leave them at the end of Group A:

Figure 2.3



When submitted for rendering, this single non-hierarchical group object will look exactly the same as the more complex version in Figure 3.1, but it will require less CPU overhead to accomplish it.

Sometimes, however, you may want to use a group simply for the sake of organizing data. In Figure 2.2, we had all of the TriMeshes grouped into Group B. This served no purpose other than to organize the data nicely. Luckily, the QuickDraw 3D designers realized this and they have given us the ability to set a group as "inline." Making a group inline simply means that QuickDraw 3D will not push and pop the state information when it enters and exits a group. It allows you to organize data in groups while avoiding the unnecessary overhead of saving and restoring the state information.

To set a Display Group as inline all you need to do is change the group's "state bits":

```
TQ3DisplayGroupState theState;

Q3DisplayGroup_GetState(theGroup, &theState);
```

```
theState |= kQ3DisplayGroupStateMaskIsInline;
Q3DisplayGroup_SetState(theGroup, theState);
```

The above code first gets the group's current state bits by calling `Q3DisplayGroup_GetState`. We then activate the inline bit by ORing in `kQ3DisplayGroupStateMaskIsInline`. Finally, to update the group's state bits, we simply call `Q3DisplayGroup_SetState`.

It is still more optimal to not have any unnecessary groups in your hierarchy, but if you really need them then try to make as many of them inline as possible to improve performance. Just remember that an inline group does not preserve any state information, so transforms and attributes can get mangled in your hierarchy if you are not careful.

If we were to define Group A as inline we would be in big trouble. Since group A contains a transform object, the active transformation matrix would be set to some arbitrary value upon exiting the group. This can cause all further objects and groups to be transformed incorrectly. As a rule of thumb, only inline groups which contain nothing but geometry objects. Avoid inlining groups containing attributes and/or transforms.

MATHEMATICAL OPTIMIZATIONS

Earlier I showed how it was better to write your own transform function rather than to call one of QuickDraw 3D's built in math functions. In general, you will always want to write your own matrix multiples, vector transforms, etc. Writing your own functions will usually result in faster code since matrices do not need to be reloaded each time you use them, and a compiler can generate better code when functions are inlined rather than called with branches. Also, many of the QuickDraw 3D mathematical functions have error checking overhead which can really eat into the performance.

The only downside to not calling QuickDraw 3D's mathematical functions is that future version of QuickDraw 3D may make use of new hardware capabilities for doing faster calculations. Despite this,

my philosophy is “write it your way and if the technology changes, update it.”

EXTENDED FLOATING POINT

All of the PowerPC chips except the 601 have the extended floating point opcodes which can make doing certain calculations much faster. Some of these extended opcodes work well in doing 3D computations and are described as follows.

Reciprocal Square Root & Newton-Raphson Refinement

When normalizing a vector we have to calculate a reciprocal square root. The usual way to do this is with code like the following:

```
    number = 1.0/sqrt(temp);
```

This is rather slow. The `sqrt` function call takes between 50-100 cycles to execute and the divide takes another 18 cycles. On the PowerPC, we can do the same calculation in 15 cycles using an extended floating point opcode called `frsqrte`.

Actually, `frsqrte` takes 4 cycles to execute, but it only returns an “estimate” of the result (`frsqrte = floating-point reciprocal square root estimate`). This estimate only has an accuracy of a few bits which is not accurate enough for a 3D engine in most cases. Luckily, there is something known as Newton-Raphson refinement which can take this reciprocal square root estimate and refine it to a very accurate value in just a few instructions.

The code for Newton-Raphson refinement looks like this:

```
float  isqrt,temp1,temp2,result;

isqrt = __frsqrte(num);
```

```
temp1 = num * -.5f;
temp2 = isqrt * isqrt;
temp1 *= isqrt;
isqrt *= 1.5f;
result = temp1 * temp2 + isqrt;
```

I'm not going to even bother trying to explain how the Newton-Raphson refinement works because I honestly don't know how it works. It's one of those things you find deep in a math book where no sane person should ever venture. Just be happy that it works and don't ask why.

Reciprocal Estimate

Another nice opcode which exists in the extended floating point instruction set is the reciprocal estimate (`fres`). This opcode, like the reciprocal square root estimate opcode, also returns an estimate value, but this estimate is accurate to one part in 256 which is not accurate enough to launch the Space Shuttle, but in many cases it's accurate for a 3D engine.

There's a catch, however. On the PowerPC 603's and 604's, the `fres` opcode takes just as long as a floating point divide - 18 cycles. All it saves you is the cost of getting the 1.0 floating point value into a register to do the divide. On the new PowerPC chips starting with the PowerPC 750 (the "G3"), however, the `fres` opcode only takes 10 cycles and is therefore significantly faster than a floating point divide. It's unlikely that you will gain much performance by using `fres` to calculate reciprocals since only brand new machines will benefit from it, so use it sparingly.

Code to use `fres` looks like this:

```
float  myReciprocal;
myReciprocal = __fres(someFloat);
```

SUMMARY

In this chapter we learned how to perform several optimization tricks to QuickDraw 3D applications:

- The single most important thing learned in this chapter was manual object culling. This will give most 3D applications a huge speed boost!
- Paying attention to texture size will help improve performance since powers of 2 are preferred by all 3D hardware accelerators.
- Being careful how you construct hierarchical groups will also improve performance.
- Never construct multiple layers of groups unless it's absolutely necessary, and try to use inline groups whenever possible.
- Using the extended floating point opcodes will speed up some mathematical calculations such as reciprocals and square roots.