Making Cool QuickDraw 3D Applications!

By
Brian Greenstone
Apple Computer, Inc.

Introduction

QuickDraw 3D is perhaps the best-designed API that Apple Computer has ever created. It provides a way for the average programmer who is not well versed in 3D programming to create complex 3D scenes in very little time with very little effort. The API is so robust that even beginner programmers should have no problem diving into this otherwise convoluted and unstandardized technology we all know as "3D Graphics". A recent survey of 25 QuickDraw 3D developers showed that the average developer gave the QuickDraw 3D API an 8.8 out of 10 where 0 was awful, 5 was average and 10 was great.

At first glance QuickDraw 3D may seem overwhelming (the QuickDraw 3D 1.5 Technical Reference is almost 1400 pages long!), but keep in mind that you'll probably never use 70% of what's in the book. The reality of the situation is that to use QuickDraw 3D and to use it well, you only need to be familiar with a small subset of the functions that QuickDraw 3D provides.

This document is going to focus on that small subset of API functions which are needed to create fast and efficient 3D worlds. We will not be discussing the slow and difficult to use geometries such as NURBS, but rather we will focus mainly on one geometry in particular: the TriMesh. This is the data structure which is the easiest to work with and also provides the maximum rendering performance, especially with 3D accelerator hardware.

There are many QuickDraw 3D applications available today, but very few of the programmers who wrote them knew how to write the code in such a way as to achieve maximum performance. I've seen some 3DMF geometry files created by these applications which render up to 13 times faster after they have been reoptimized with the techniques discussed in this book. It's not that these weren't good programmers, it's just that there is very little information available to developers to teach them what works best in QuickDraw 3D.

Many people have asked me if I really think QuickDraw 3D is “fast.” Well, yes, it’s “fast”, but “fast” is a relative term. QuickDraw 3D will never be as fast as a custom 3D engine which you might write for a specific task such as a game or a modeling application. No general purpose API is ever as fast as an engine built for a specific task, but QuickDraw 3D can come very, very close. I believe that QuickDraw 3D can come to 90-95% the speed of a custom 3D engine in most cases. I’ve thought about writing my own 3D engine for several years now, but as QuickDraw 3D has gotten better and better, I’ve found that it’s simply not worth the expense of writing such an engine when QuickDraw 3D does everything I’d ever need and it only costs me a small percentage of relative speed.

This document is not meant to be a 3D tutorial, nor a tutorial on QuickDraw 3D. I am going to assume that you already have a basic understanding of how 3D and QuickDraw 3D work. This will save hundreds of pages which would only duplicate information found in dozens of other 3D programming books found at any book store. Instead, this book will teach you how to code QuickDraw 3D such that you get the maximum possible performance out of your applications. It will also show you how to perform various 3D tasks such as calculating splines, doing rudimentary collision detection, and creating QuickTime movies with 3D tracks. By the time you finish reading this documentation you will know all there is to know about writing super-fast QuickDraw 3D applications.

Topic 1:

The TriMesh Geometry

WHAT IS THE TRIMESH?

TriMesh is your friend. If your primary concern is speed you will want to use TriMesh geometries since they will render fast. The TriMesh geometry type is a very low-level geometry which was introduced in QuickDraw 3D 1.5. Before TriMesh existed, we used less optimized geometry types like Mesh or TriGrid to build our 3D models. These other geometry types are easier and more flexible to work with than TriMesh, but generally do not give you as much performance as the TriMesh. In addition, 3DMF files containing Mesh geometries take a lot longer to load than 3DMF files containing TriMeshes.

TriMesh geometry is very streamlined and the data can be passed to hardware accelerators in whole without being broken down into its individual triangles. Most 3D accelerators can process TriMeshes around two times faster than they can a stream of individual triangles.

The TriMesh Data Structures

TQ3TriMeshData

Simply put, a TriMesh is just a bunch of parallel arrays which define all of the points and attributes in a model. The main data structure looks like this:

```
typedef struct TQ3TriMeshData
{
   TQ3AttributeSet         triMeshAttributeSet;

   unsigned long           numTriangles;
   TQ3TriMeshTriangleData  *triangles;

   unsigned long           numTriangleAttributeTypes;
   TQ3TriMeshAttributeData *triangleAttributeTypes;

   unsigned long           numEdges;
   TQ3TriMeshEdgeData      *edges;

   unsigned long           numEdgeAttributeTypes;
   TQ3TriMeshAttributeData *edgeAttributeTypes;

   unsigned long           numPoints;
   TQ3Point3D              *points;

   unsigned long           numVertexAttributeTypes;
   TQ3TriMeshAttributeData *vertexAttributeTypes;

   TQ3BoundingBox          bBox;
} TQ3TriMeshData;
```

Unlike most of the other geometries in QuickDraw 3D, there are no support functions which help you add faces, vertices, or attributes to a TriMesh. You get to build all of the data by hand, therefore, it is important to really understand the `TQ3TriMeshData` structure.

The first record, `triMeshAttributeSet`, is simply a reference to a regular QuickDraw 3D Attribute Set object. This attribute set will contain all of the attributes to apply to the entire TriMesh such as its color or texture map.

`numTriangles` determines how many triangles are in the TriMesh, and `triangles` points to an array of triangle definitions (see below) which you supply.

`numTriangleAttributeTypes` determines how many types of attributes the triangles have, and `triangleAttributeTypes` points to an array which contains all of the attribute data. All of the triangles in a TriMesh have the same types and quantities of attributes, but the value of each attribute can differ from triangle to triangle. In other words, if one triangle has a face normal attribute, then they all have face normal attributes. Actually, the only triangle attribute

which we will ever want to include in our TriMeshes is a face normal attribute. I'll go into more detail about triangle and vertex attributes later, but suffice to say that you will never want to assign anything but face normals to the triangles.

`numEdges` is used for defining edges on your TriMesh. This is only needed if the fill style you're using to render is set to `kQ3FillStyleEdges`. Since you're probably not going to use edge rendering for a fast, interactive, 3D application, we'll always leave `numEdges` and `numEdgeAttributeTypes` set to 0. Also be sure to set the `edges` and `edgeAttributeTypes` pointers to nil.

`numPoints` is the number of vertices in the TriMesh, and `points` points to an array of 3D points (`TQ3Point3D`) containing the coordinates of all the vertices.

`numVertexAttributeTypes` and `vertexAttributeTypes` are like their counterparts `numTriangleAttributeTypes` and `triangleAttributeTypes`. These records define the attributes you wish to assign to each vertex. The only attributes we'll need to apply to our vertices are vertex normals and texture uv coordinates.

`bBox` is the bounding box encapsulating all of the points in the TriMesh. QuickDraw 3D provides a utility function called `Q3BoundingBox_SetFromPoints3D` which can be used to calculate the correct bounding box based on the points in the `points` array.

> It is critical that you calculate this correctly! Do not even consider setting `bBox.isEmpty` to true! This may result in a serious performance hit. Also, make sure to never ever create a bounding box smaller than what it should be. If there are vertices which lie outside of the bounding box then your application is destined to eventually crash. Be very diligent about generating a correct bounding box for each TriMesh.

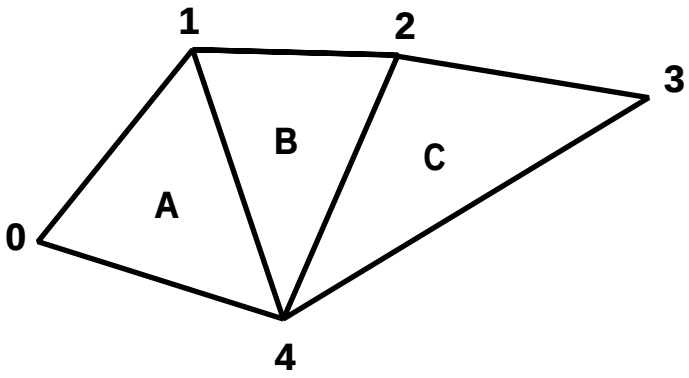
Triangles

As described above, `triangles` points to an array of triangle definitions. A triangle definition is a simple data structure which looks like this:

```
  typedef struct TQ3TriMeshTriangleData
  {
     unsigned long    pointIndices[3];
  } TQ3TriMeshTriangleData;
```

Since the points are kept in the `points` array, all that is needed to define a triangle are three indices into the `points` list. So, suppose we have following geometry:

Figure 1.0

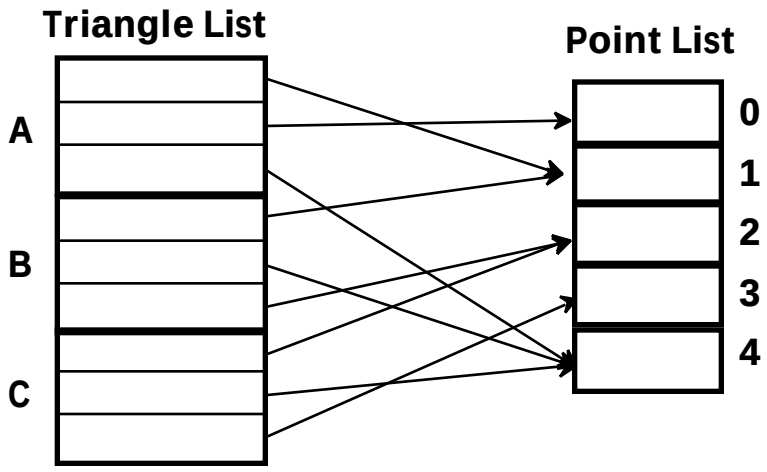


A geometry made of 3 triangles (A, B, anc C) and 5 points (0..4)

The triangles are thus built in the TriMesh as:

```
TQ3TriMeshData          myTriMesh;
TQ3TriMeshTriangleData  triangles[3] =
{
  1,0,4,      // triangle A
  1,4,2,      // triangle B
  2,4,3       // triangle C
};

myTriMesh.numTriangles = 3;
myTriMesh.triangles = &triangles[0];
```

Figure 1.1



Graphical representation of the relation between triangles and the point list.

TriMesh Attribute Arrays

Setting up attribute arrays for faces and vertices is a little strange at first because it doesn't work like anything else in QuickDraw 3D. It's actually a bit messy, but it makes sense.

Remember that `numTriangleAttributeTypes` determines how many types of attributes we need for the faces of the TriMesh. Since the only face attribute we will ever want to apply to a TriMesh is a face normal, we can set this value to 1. The pointer `triangleAttributeTypes` simply points to a single `TQ3TriMeshAttributeData` structure which has the following form:

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

The `attributeType` parameter is set to `kQ3AttributeTypeNormal` since we want to assign normals to the faces.

`data` points to an array of values for the specified attribute type. Since our attribute type is `kQ3AttributeTypeNormal` this data pointer points to an array of vectors (`TQ3Vector3D`).

> There must be exactly as many vectors in the array as there are triangles in the model. This way there is exactly 1 vector for each triangle - no more, no less.

`attributeUseArray` is used for custom attributes so always set this to nil since we don't want to mess with those.

The code to set up these attributes might look like the following:

```
TQ3TriMeshData          myTriMesh;
TQ3TriMeshAttributeData attribData;
TQ3Vector3D             vectorArray[NUM_TRIANGLES];

       /* SET MAIN TRIMESH STRUCT */

  myTriMesh.numTriangles = NUM_TRIANGLES;

  myTriMesh.numTriangleAttributeTypes = 1;
  myTriMesh.triangleAttributeTypes = &attribData;


       /* SET ATTRIBUTE STRUCT */

  attribData.attributeType = kQ3AttributeTypeNormal;
  attribData.data = &vectorArray[0];
  attribData.attributeUseArray = nil;
```

Setting normals for each of the vertices is almost completely identical to the above code, but very often we will also need to apply UV texture mapping coordinates to each vertex. As with the faces, there must be a 1:1 correlation between the number of points and the number of attribute values for each attribute type, therefore, the normal and uv arrays must have as many entries as there are points in the model.

Figure 1.2

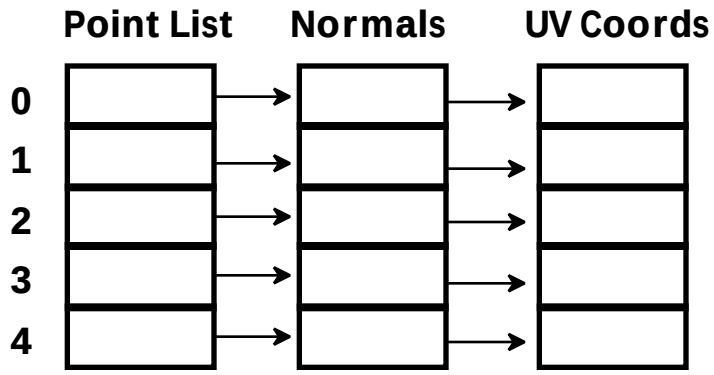


Diagram showing the parallel correlations among the point list, normal list, and UV list.

The following code shows how to setup the normal and uv attributes for the vertices in a TriMesh:

```
TQ3TriMeshData          myTriMesh;
TQ3TriMeshAttributeData attribData[2];
TQ3Vector3D             vectorArray[NUM_VERTICES];
TQ3Param2D              uvArray[NUM_VERTICES];

       /* SET MAIN TRIMESH STRUCT */

  myTriMesh.numPoints = NUM_ VERTICES;

  myTriMesh.numVertexAttributeTypes = 2;
  myTriMesh.vertexAttributeTypes = &attribData;


       /* SET ATTRIBUTE STRUCT */

  attribData[0].attributeType = kQ3AttributeTypeNormal;
  attribData[0].data = &vectorArray[0];
  attribData[0].attributeUseArray = nil;

  attribData[1].attributeType = kQ3AttributeTypeSurfaceUV;
  attribData[1].data = &uvArray [0];
  attribData[1].attributeUseArray = nil;
```

Building the Whole TriMesh

Now let's see how to build the TriMesh shown in Figure 1.0.

```
/*************** BUILD MY TRIMESH ***************/
//
// INPUT: textureAttrib = reference to attribute set containing
//                        the texture shader to apply to the
//                        TriMesh.
//
// OUTPUT: a reference to the new TriMesh geometry object
//

TQ3GeometryObject BuildMyTriMesh(TQ3AttributeSet *textureAttrib)
{
TQ3TriMeshData          myTriMeshData;
TQ3TriMeshAttributeData vertexAttribs[2],faceAttribs;
TQ3GeometryObject       myTriMeshObject;

TQ3Vector3D vertexNormals[5] =
{
  x0,y0,z0,
  x1,y1,z1,
  x2,y2,z2,
  x3,y3,z3,
  x4,y4,z4
};

TQ3Vector3D faceNormals[3] =
{
  x0,y0,z0,
  x1,y1,z1,
  x2,y2,z2
};

TQ3Param2D uvArray[5] =
{
  u0,v0,
  u1,v1,
  u2,v2,
  u3,v3,
  u4,v4
};

TQ3Point3D points[5] =
{
  x0,y0,z0,
  x1,y1,z1,
  x2,y2,z2,
  x3,y3,z3,
  x4,y4,z4
};

TQ3TriMeshTriangleData  triangles[3] =
{
  1,0,4,      // triangle A
  1,4,2,      // triangle B
  2,4,3       // triangle C
};
```

```
        /* BUILD MAIN TRIMESH DATA STRUCTURE */

    myTriMeshData.triMeshAttributeSet = textureAttrib;

    myTriMeshData.numTriangles = 3;
    myTriMeshData.triangles = &triangles[0];

    myTriMeshData.numTriangleAttributeTypes = 1;
    myTriMeshData.triangleAttributeTypes = &faceAttribs;

    myTriMeshData.numEdges = 0;
    myTriMeshData.edges = nil;
    myTriMeshData.numEdgeAttributeTypes = 0;
    myTriMeshData.edgeAttributeTypes = nil;

    myTriMeshData.numPoints = 5;
    myTriMeshData.points = &points[0];

    myTriMeshData.numVertexAttributeTypes = 2;
    myTriMeshData.vertexAttributeTypes = &vertexAttribs[0];


        /* CALCULATE BOUNDING BOX */

    Q3BoundingBox_SetFromPoints3D(&myTriMeshData.bBox, &points[0],
                              5, sizeof(TQ3Point3D));


        /* CREATE FACE ATTRIBUTES */

    faceAttribs.attributeType = kQ3AttributeTypeNormal;
    faceAttribs.data = &faceNormals[0];
    faceAttribs.attributeUseArray = nil;


        /* CREATE VERTEX ATTRIBUTES */

    vertexAttribs[0].attributeType = kQ3AttributeTypeNormal;
    vertexAttribs[0].data = &vertexNormals[0];
    vertexAttribs[0].attributeUseArray = nil;

    vertexAttribs[1].attributeType = kQ3AttributeTypeSurfaceUV;
    vertexAttribs[1].data = &uvArray[0];
    vertexAttribs[1].attributeUseArray = nil;


      /* MAKE THE TRIMESH GEOMETRY OBJECT */

    myTriMeshObject = Q3TriMesh_New(&myTriMeshData);
    if (myTriMeshObject == nil)
      DoError("\pQ3TriMesh_New failed!");


    return(myTriMeshObject);
}
```

When Q3TriMesh_New is called all of the data in the various TriMesh data structures and arrays gets copied into QuickDraw 3D's internal structures. Any further modifications to myTriMeshData or the attribute structures will have no effect on the new TriMesh object we have created. The only way to change the settings of this TriMesh is to call Q3TriMesh_SetData which will update the object with the latest values contained in the data structures.

Object References & Memory

When the TriMesh object is created, the reference count of the triMeshAttributeSet is increased by 1 since that attribute object is now included in the new TriMesh. Making a call to Q3TriMesh_GetData to get all of the data in a TriMesh object will increase the reference count of the attribute set again. Because of this action, it is very important that you properly dispose of TriMesh data obtained from a call to Q3TriMesh_GetData. Calling Q3TriMesh_Empty data will properly decrement the attribute set's reference count and will dispose of all other memory allocated by Q3TriMesh_GetData.

It is important to realize that Q3TriMesh_GetData allocates memory and copies the TriMesh's data into that memory. The pointers contained in the main TriMesh data structure do not point to data actually being used by QuickDraw 3D to represent the TriMesh. The pointers point to copies of that data, therefore, modifying the data will have no effect until Q3TriMesh_SetData is called to update the TriMesh.

Failure to call Q3TriMesh_Empty will result in memory leaks and incorrect reference counts to any assigned attribute sets.

MAKING EFFICIENT TRIMESHES

Just because you can build a TriMesh doesn't mean that your 3D application will run fast. I've seen a lot of people who create arbitrary TriMeshes and expect them to be blazingly fast, but this is not how it works. To make QuickDraw 3D burn rubber and scream

like a demon, you need to build TriMeshes in a particular way by following some basic rules and principles:

One Material Per TriMesh

The most important rule to making fast and efficient TriMeshes is to only apply one "material" per TriMesh. In QuickDraw 3D there is no such thing as a "material" per se, but for our purposes a material is a combination of attributes which define how a surface looks. These attributes include texture shaders, colors, specular and diffuse values, etc. So, when we create a TriMesh object, we never ever want to have more than one material assigned to that TriMesh. Never build a single TriMesh with multiple textures or multiple colors. This will kill any performance you ever hoped to gain by using TriMeshes.

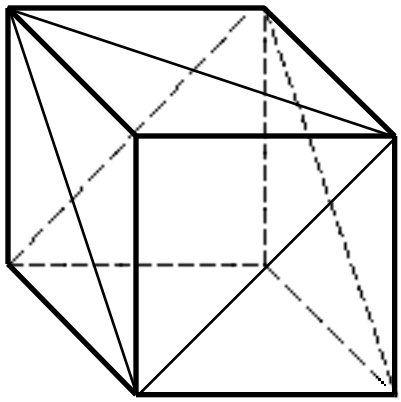
You may be wondering "How do I build my airplane model which has six different texture maps if I'm being told to only apply one material per TriMesh?" The answer is that you must build your airplane model from six different TriMeshes - one for each texture map. The reason for doing this is because QuickDraw 3D, RAVE, and the 3D accelerator cards function much faster when they are given large streams of triangles which all have common attributes. If you assign a texture shader attribute to each individual triangle in a TriMesh, don't expect to get very good rendering performance at all. You should only apply a texture shader or color attribute to the TriMesh's main attribute set, but never ever to the individual triangles or vertices.

Earlier, I told you that the only attribute you will ever want to apply to a triangle is a face normal. Never waver from this rule because a face normal is the only attribute you can assign to a triangle in a TriMesh which will not hinder performance. The same goes for vertex attributes. Only vertex normals and vertex u/v texture mapping coordinates should be used in a TriMesh. As long as you stick to the "one material per TriMesh rule" you'll be in good shape.

Watch out for Duplicate Data

For as much as I have praised the wonderful TriMesh, it is not without flaws. It does have one fundamental flaw which can cause performance problems and there is no good way around it. The best way to explain the issue is to use a simple example. Suppose we want to model a cube:

Figure 1.3



A cube constructed from 12 triangles and 8 points

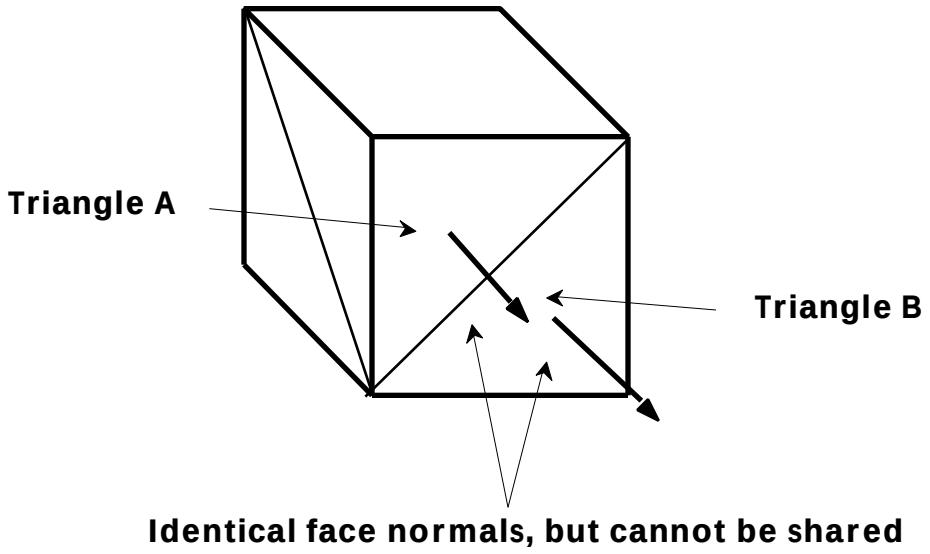
This cube is made up of 12 triangles, 8 points, and 6 normals, right? Wrong! There is no way to represent this cube in such a way using the TriMesh, and here's why:

1. Face and Vertex attributes cannot be shared in any way, therefore, we end up with two independent arrays of normals: one for the triangles and one for the vertices. QuickDraw 3D must transform both lists of normals independently even though they contain identical values.

2. Because TriMeshes are based on the concept of parallel arrays of data, we end up with even more duplicate data per vertex and per triangle. For example, the two front faces on the cube have the same face normal, but because the attribute array is parallel to the triangle array, each triangle has to have it's own copy of the normal. Same goes for the vertices of each triangle.

The three vertices of a triangle in the cube should share the same normal, but the parallel arrays of attributes makes this impossible, thus we end up with three copies of the same normal.

Figure 1.4

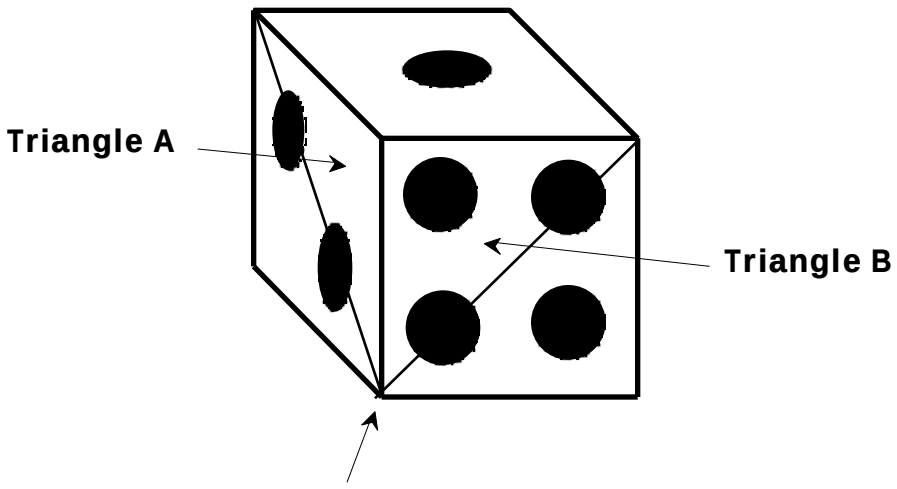


Triangle A and B have identical face normals which cannot be shared, so two copies are needed.

3. The same problem applies to vertex u/v coordinates. Even though a triangle on the left of the cube may share a vertex coordinate with a triangle on the front, the u/v coordinates for that vertex will probably be different for each triangle, thus the point cannot be shared by the two triangles. The result is a duplicate copy of the point.

   Suppose each face of the cube has a different texture assigned to it (say we're making a model of a die). Remember that you should only have one material per TriMesh. This means that each side of the cube needs to be a separate TriMesh, therefore, it would take six different TriMeshes to represent this model. Even if we broke the one material per TriMesh rule, we'd still have the problem of vertices having different u/v coordinates depending on which triangle was using it.

Figure 1.5



Vertices cannot share common points if the u/v values are not identical.

The result of this inability to share duplicate data is that it takes 24 points, 12 face normals, and 24 vertex normals to build this TriMesh. Not a very efficient way to represent a simple cube, eh?

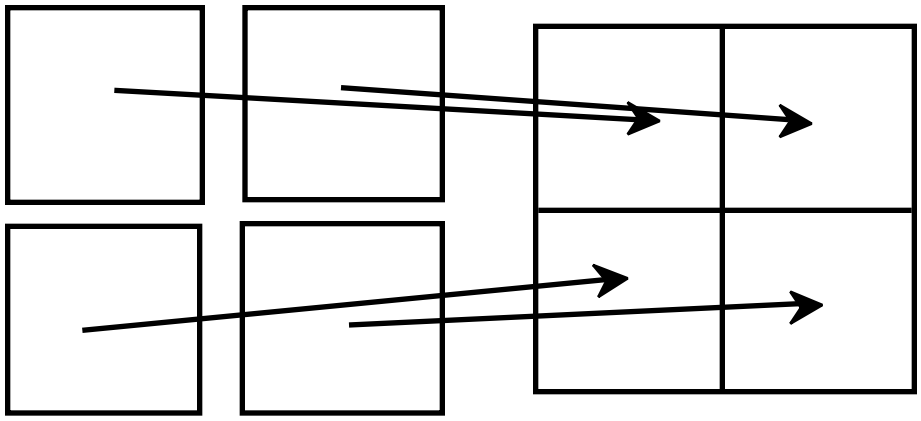
Working Around The Restrictions

When I was first told about the above problems with TriMeshes I figured that was the final nail in the coffin for TriMesh. I couldn't understand how I was supposed to build anything under those kinds of conditions. Luckily, I found that just about anything in the universe has a work-around and even though there's no "perfect" solution to these problems, there are "acceptable" solutions. Additionally, the cube is a sort of worst-case example. Most real-world models don't suffer this severity of the problem.

Merging Texture Maps

If you have an airplane model which uses 4 different texture maps, there's no need to create 4 different TriMeshes to build it. It makes much more sense to try to merge all 4 textures into one bigger texture map. The simple way to do this is shown in Figure 1.6.

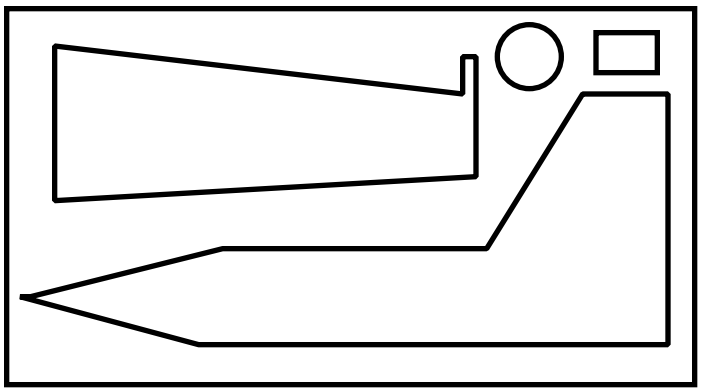
Figure 1.6



Combine 4 separate textures into one big texture to that a single TriMesh can be built

The more complex way to do this is to fill in the "black" space in a texture with other sub-textures. For example, the following single texture map actually contains multiple textures which we've wedged into what was the black space in the largest of the original textures:

Figure 1.7



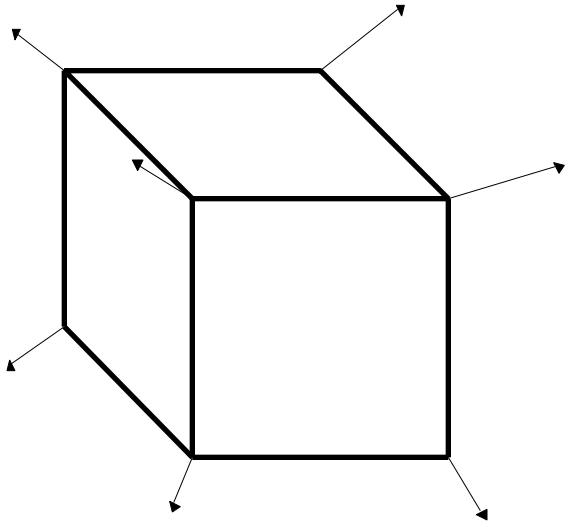
We put the wing, wheel, and windshield texture in the "black-space" of the body texture.

These solutions have their problems, but if maximum speed is your top concern then these problems will seem trivial. The first problem is that a large map may have a more difficult time fitting into VRAM if VRAM is running low on your 3D accelerator card. Secondly, if you are using Bi-Linear or Tri-Linear texture mapping then you may get texture bleeding. This occurs at the edges of a texture map where the pixels are smoothed with the pixels adjacent to it. If the adjacent pixels are from another texture map, then you may get some bleed through. To avoid this, just keep a margin in between your merged textures.

Smoothing Models

If the above cube model was smoothed such that the vertex normals were identical for each triangle using that vertex, then vertices could be shared.

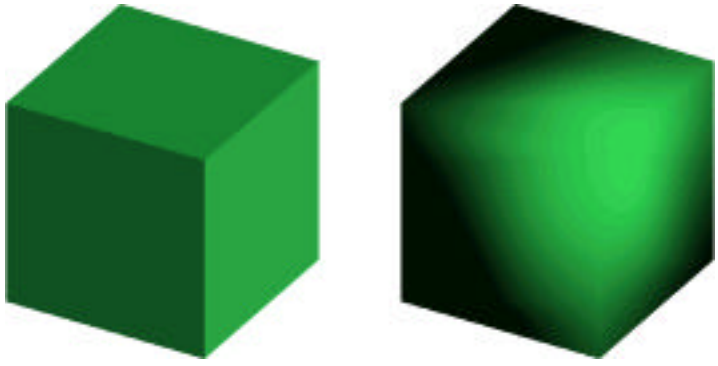
Figure 1.8



When triangles share vertices, the normals get averaged.

The cube no longer has the “hard” edges that we wanted, but it does share common vertices with common points and normals which improves performance dramatically. Now we can build the TriMesh from 8 points & vertex normals, and 12 face normals.

Figure 1.9



The cube on the left shows the inefficient TriMesh with lots of duplicate data. The cube on the right

doesn't have the hard edges, but it's much more efficient.

So, the simple rule here is to avoid having hard edges in your models. Hard edges equate to duplicate vertices which slow down performance.

You may think that this really sucks, but realize that for rendering organic models, this works great. The dinosaur models in Nanosaur have no hard edges and use one gigantic texture map. They form incredibly optimal TriMeshes and looked great!

Figure 1.10

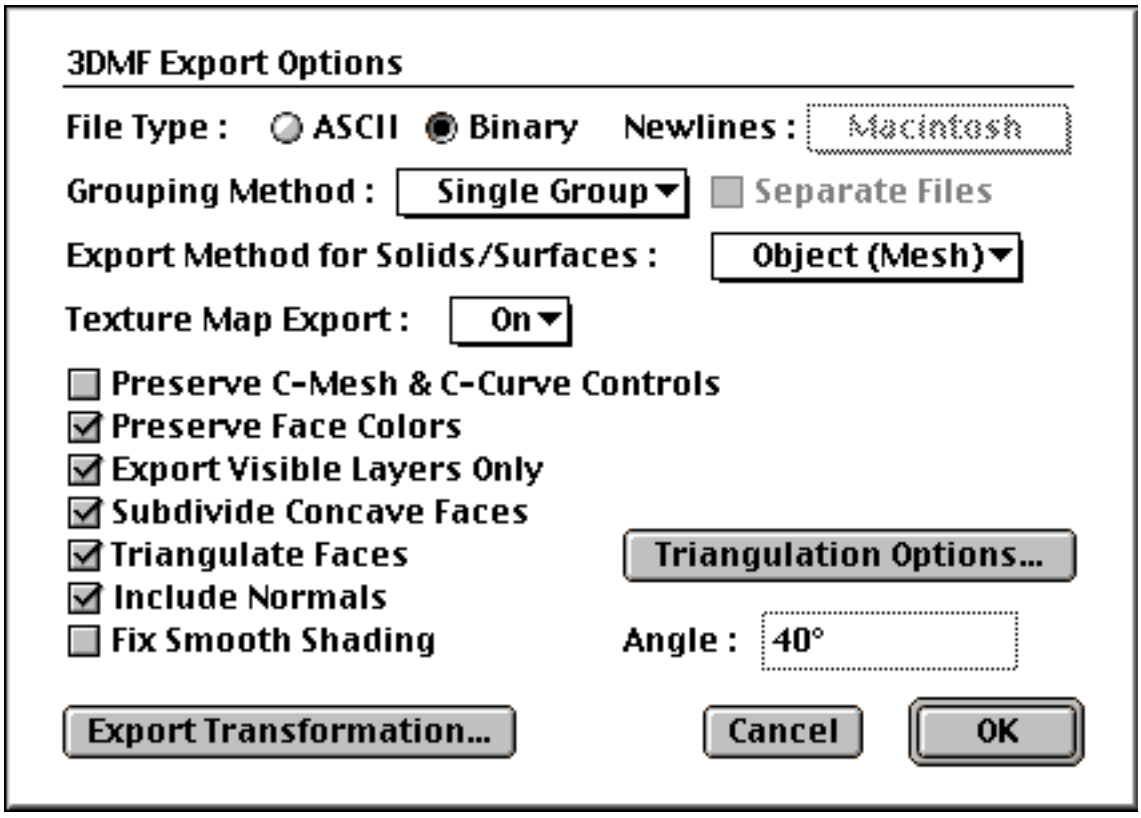
This model is entirely smooth shaded and is made from a single, highly optmized TriMesh.

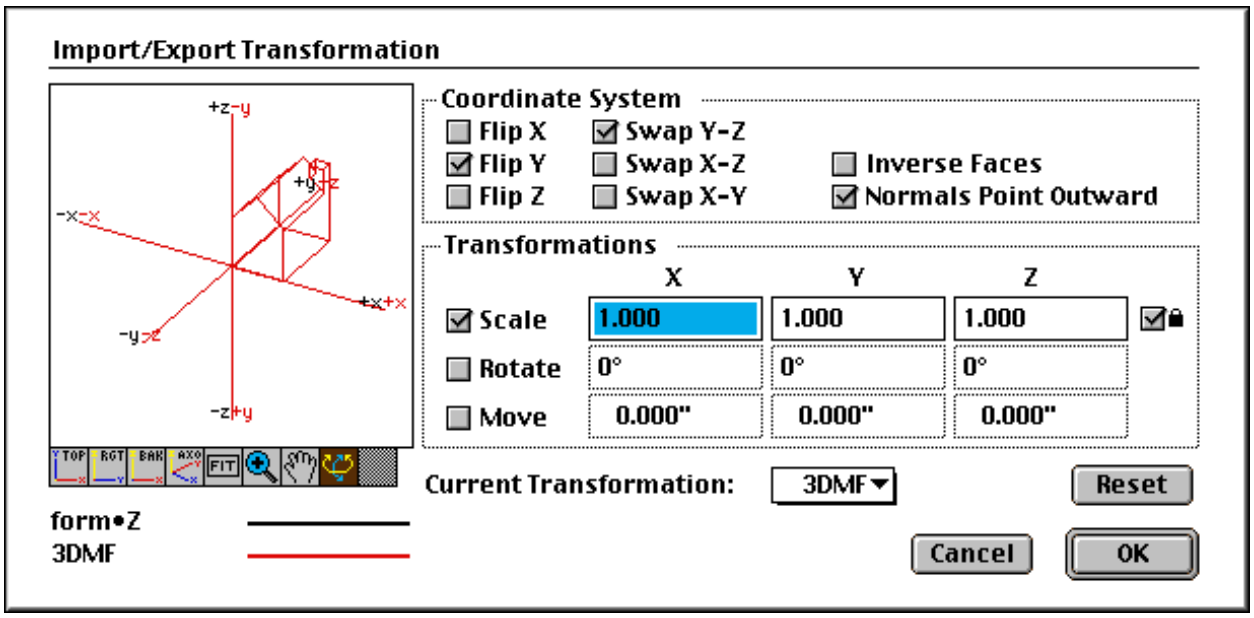
If you absolutely must create models with hard edges then just be aware that you may get less performance that you expect. Always make the best attempt to share vertices in a model to get the best performance.

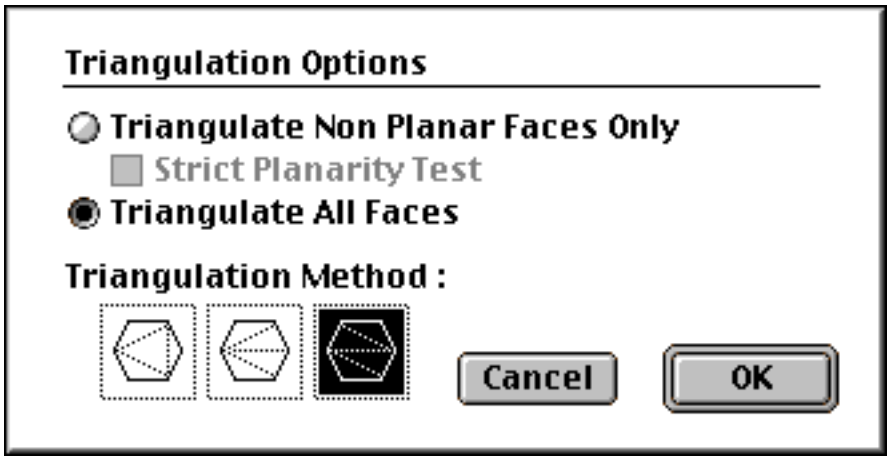
Form*Z & 3DMF Optimizer

Exporting 3DMF Files from Form*Z

For building highly efficient 3DMF models, I like to use Form*Z. In addition to being a fantastic 3D modeling application for creating low polygon count geometries, Form*Z output's fairly clean 3DMF files. Its output is a little buggy at times, but if you use the right export settings it works great! The following images show what settings you should use in Form*Z when exporting a 3DMF file:







These settings work great about 99% of the time, but occasionally the exported model will have inverted faces. It seems that the solution to this is to just "tweak" the texture mapping coordinates of any objects whose faces are flipped and then re-export the 3DMF file. Usually, this will cause the bad faces to magically correct themselves.

You'll note that I recommend you export the geometry as Mesh and not TriMesh. I've had problems with the TriMesh export in Form*Z and I've found Mesh to be much more reliable. Not to worry, however, because 3DMF Optimizer takes care of converting those meshes into TriMeshes.

Also note that I turn “off” the Flip X option and turn “on” the Flip Y option. For some reason, Form*Z always defaults to the wrong settings. If you use their defaults, your object will be inverted along the z-axis when you view it in a QuickDraw 3D application, therefore, make sure you remember to change these checkboxes when you export your models.

One other problem you may have with Form*Z are the vertex normals. If you have a model with some smoothed geometry and some non-smoothed geometry, you’re out of luck. Seems that when you go to export the model to 3DMF, Form*Z either smoothes the entire thing or none of it. If you have the Fix Smooth Shading option activated then you get non-smoothed models, otherwise, the entire model will be smoothed whether you wanted it to be or not.

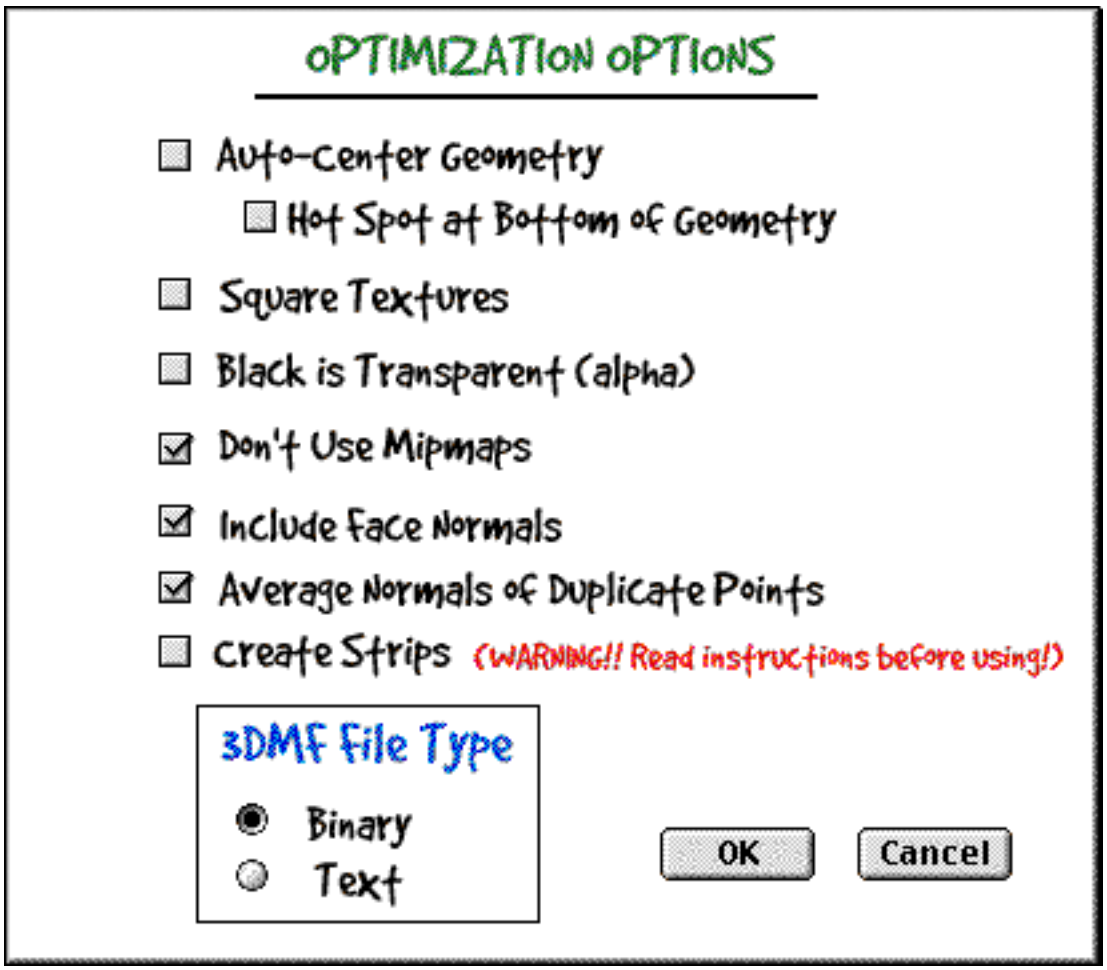
Form*Z does export transparency attributes on transparent geometry, but it does not use the transparency value in the material you have assigned to it. Rather, it uses the Transparency on/off value which you can apply to a model on an object by object basis. The transparency value applied is 50% and unfortunately there is no way to modify that.

Using 3DMF Optimizer

Once you have a 3DMF file which you created in Form*Z or any of the other 3D modelers, you should always process it with 3DMF Optimizer. This tool parses a 3DMF file, optimizes its contents, and converts all geometry into TriMeshes. The 3DMF file output by 3DMF Optimizer is as optimal as you can possibly make it.

In general, 3DMF Optimizer speeds up rendering of a model by 2 to 3x and it decreases file sizes and load times by 4-10x. Some of the more “offensive” 3DMF files get speed-ups in the range of 5-13x!!!

3DMF Optimizer has a nice Options dialog which lets you determine many aspects of the optimizing process. In general, I recommend that you keep the settings as they are in the following figure unless you have a specific need to change them:



These settings will generate the fastest and smallest 3DMF file possible. This tool can do a lot of great things with 3DMF files and it is constantly being updated. A demo is available on the Pangea Software web site at http://www.realtime.net/~pangea.

STRIPS & FANS OPTIMIZATIONS

There is one more TriMesh optimization which may speed up your application: Strips and Fans. Be warned that as of this writing, this optimization actually has no effect. RAVE directly supports Strips and Fans, but QuickDraw 3D does not. QuickDraw 3D does, however, support the TriMesh (obviously), and if a smart 3D accelerator card driver checks arbitrary TriMeshes for Strips and Fans, then this optimization will work for you. Unfortunately, I do not believe that

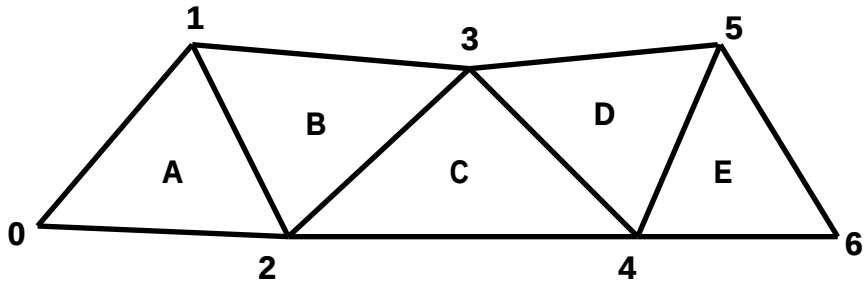
any of the 3D card drivers currently make such a test. Also note that the strip and fan optimization really does only apply to 3D accelerator cards.

It is unlikely that QuickDraw 3D will ever directly support Strips and Fans, but you never know. This section is going to talk about Strips and Fans in the off chance that it eventually is added to QuickDraw 3D's internal workings, but keep in mind that currently the only way to make use of Strips and Fans is to write code directly to RAVE instead of QuickDraw 3D.

Strips

A "strip" or "fan" simply refers to the way in which triangles and vertices are ordered in a TriMesh. The idea is to be able to represent a triangle by only 1 vertex instead of 3. "How can this be done?" you ask. Look at the following mesh:

Figure 2.11



A TriMesh which is "stripped"

As this TriMesh is processed, we work from triangle A through triangle E. When it passes triangle A to the 3D hardware, it needs to pass vertices 0, 1, and 2.

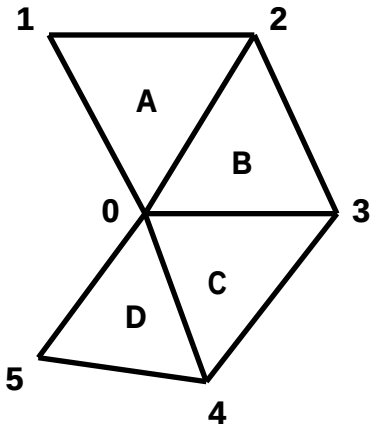
Next, we do triangle B, but because vertices 1 and 2 were already sent to the hardware for triangle A, we only need to send vertex 3 to

define triangle B. When this happens, the 3D hardware assumes that you want to use the last two vertices from the previous triangle plus the one new vertex to define the new triangle. So, to draw triangle C, QuickDraw 3D only needs to send vertex 4 to the hardware because the hardware automatically knows to use vertex 2 and 3 from the previous triangle. Following this pattern you can see that triangle D only needs to send vertex 5 and so on and so on.

Fans

Fans are very similar to Strips, but they revolve around a central vertex as shown here:

Figure 2.12



Here, triangle A is drawn by passing vertices 0, 1, and 2 to the 3D hardware. Next, to draw triangle B, only vertex 3 needs to be passed since the hardware will use vertices 0 and 2 from the previous triangle.

Triangle C will now use vertices 0 and 3 from triangle B, and triangle D will use vertices 0 and 4 from triangle C, and so on.

Don’t Get Too Excited

Don’t get too excited about using Strips and Fans to build your TriMesh. Writing an algorithm to efficiently generate long streams of triangles like this is very difficult. Not only do you have to submit adjacent triangles one after another, but the vertex list being used for these triangles must be linearly incremental. In other words, the submitted vertices must be in order such as 0,1,2,3,4, etc. or 104,105,106, etc. Taking an arbitrary 3D model and getting the data into this kind of order is extremely difficult and often impossible to do with any degree of efficiency.

Like I said earlier, the current version of QuickDraw 3D does not even recognize strips or fans. The only benefit you will get from strips and fans is if you are writing directly to RAVE in which case you can pass your data to the hardware as strips or fans and get a substantial speedup. There is a higher chance that some 3D accelerator drivers will automatically detect Strip and Fan patterns in a TriMesh than the chance of having direct Strip and Fan support in QuickDraw 3D. Note that if a 3D driver recognizes a Strip or Fan in a TriMesh that the vertices do not need to be sequentially ordered. The driver will simply check if the first two vertices of the new triangle match the last two vertices of the previous triangle, and if so, it knows that it is a Strip.

Should QuickDraw 3D ever support strips and fans, you should also note that the vertex ordering of every other triangle switches direction. In figure 2.11 the first triangle’s vertices are ordered clockwise (0,1, and 2), but the second triangle is ordered counter-clockwise (1,2 and 3). Then the third triangle is clock wise again (2,3 and 4). If you have backface removal turned on in QuickDraw 3D, then you’ll need to make sure that your face normals also alternate to cancel out the changes in vertex ordering. Otherwise, your TriMesh will be drawn with every other triangle removed via backface removal.

EDGE GENERATION

As mentioned earlier in this chapter, you should set `numEdges` to 0 in the `TriMeshData` structure because this information is not needed for rendering triangles. Having this data only increases the file size and memory usage. However, there are many times where edge rendering comes in very useful. If there are edges assigned to your TriMesh, the Wireframe renderer will use those edges to display the model. Otherwise, the Wireframe renderer shows a true wireframe of every edge of every triangle in the model – not a very nice thing to look at. Rendering with edges usually displays a much cleaner image, and I use edge mode extensively in many of my 3D tools.

The easy (and incorrect) way to generate edges is just to assume that each side of a triangle is an edge. Don't do this! An edge is a side of a triangle which is not adjacent to any other co-planar triangles. Correctly generating edges for a TriMesh is a fairly easy process which consists of parsing the triangle data and looking for adjacent triangles whose face normals are not identical.

The following code generates edge data for the input TriMesh object:

```
#define kMaxEdges  2000

/****************** CALC TRIMESH EDGES *********************/

void CalcTriMeshEdges(TQ3GeometryObject theTriMesh)
{
TQ3TriMeshData   triMeshData;
unsigned long    faceA,numFaces,faceB,m;
long             inda[3],indb[3];
TQ3Vector3D      faceNormalA,faceNormalB,v1,v2;
TQ3Status        status;
TQ3Point3D       *pointList,a[3], b[3],pa1,pa2,pb1,pb2;
TQ3TriMeshEdgeData edgeData[kMaxEdges];
short            numEdges = 0,e1,e2;
Boolean          edgeOnSpace[3];
TQ3TriMeshTriangleData  *faceList;

          /* GET TRIMESH DATA */

  status = Q3TriMesh_GetData(theTriMesh, &triMeshData);
  if (status == kQ3Failure)
     DoError("\pCalcTriMeshEdges: Q3TriMesh_GetData failed!");

  numFaces = triMeshData.numTriangles;          // get # faces
  faceList = triMeshData.triangles;             // point to face list
```

```
pointList = triMeshData.points;              // point to points


     /****************************/
     /* SCAN EACH FACE FOR EDGES */
     /****************************/

for (faceA = 0; faceA < numFaces; faceA++)
{
     /* GET 3 VERTS OF THIS FACE */

   inda[0] = faceList[faceA].pointIndices[0];
   inda[1] = faceList[faceA].pointIndices[1];
   inda[2] = faceList[faceA].pointIndices[2];

   a[0] = pointList[inda[0]];
   a[1] = pointList[inda[1]];
   a[2] = pointList[inda[2]];

   edgeOnSpace[0] = true;    // assume nothing adjacent on this edge
   edgeOnSpace[1] = true;
   edgeOnSpace[2] = true;


          /* CALC FACE NORMAL */

   v1.x = a[0].x - a[1].x;
   v1.y = a[0].y - a[1].y;
   v1.z = a[0].z - a[1].z;
   v2.x = a[2].x - a[1].x;
   v2.y = a[2].y - a[1].y;
   v2.z = a[2].z - a[1].z;
   Q3Vector3D_Cross(&v1, &v2, &faceNormalA);


        /* CHECK EACH FACE AGAINST ALL OTHERS */

   for (faceB = 0; faceB < numFaces; faceB++)
   {
     if (faceB == faceA)          // dont compare against self
       continue;

        /* GET 3 VERTS FOR OTHER FACE */

     indb[0] = faceList[faceB].pointIndices[0];
     indb[1] = faceList[faceB].pointIndices[1];
     indb[2] = faceList[faceB].pointIndices[2];

     b[0] = pointList[indb[0]];
     b[1] = pointList[indb[1]];
     b[2] = pointList[indb[2]];

            /* CALC FACE NORMAL */

     v1.x = b[0].x - b[1].x;
     v1.y = b[0].y - b[1].y;
     v1.z = b[0].z - b[1].z;
     v2.x = b[2].x - b[1].x;
```

```
      v2.y = b[2].y - b[1].y;
      v2.z = b[2].z - b[1].z;
      Q3Vector3D_Cross(&v1, &v2, &faceNormalB);

           /*************************/
           /* SCAN 3 EDGES FOR MATCH */
           /*************************/

      for (e1 = 0; e1 < 3; e1++)
      {
        pa1 = a[e1];                    // get 2 points of edge
        if (e1 == 2)
          pa2 = a[0];
        else
          pa2 = a[e1+1];


        for (e2 = 0; e2 < 3; e2++)
        {
          pb1 = b[e2];                  // get 2 points of edge
          if (e2 == 2)
            pb2 = b[0];
          else
            pb2 = b[e2+1];


                /* COMPARE BOTH ENDPOINTS FOR MATCH */

          if ((ComparePoints(&pa1,&pb1,0.01) &&
            ComparePoints(&pa2,&pb2,0.01)) ||
            ComparePoints(&pa1,&pb2,0.01) &&
            ComparePoints(&pa2,&pb1,0.01))
          {
               /***************/
               /* GOT A MATCH */
               /***************/
               //
               // we check face normals here (and not earlier)
               // b/c we still want to know if a face has an
               // adjacent match since empty space indicates an edge.
               //

            edgeOnSpace[e1] = false;

               /* CHECK IF THIS EDGE PREVIOUSLY DETECTED */

            if (faceB >= faceA)
              continue;

            /* IF FACE NORMALS MATCH (OR CLOSE ENOUGH), THEN SKIP */

            if (CompareVectors(&faceNormalA, &faceNormalB, 0.01))
              continue;


                   /* ADD EDGE TO LIST */

            edgeData[numEdges].pointIndices[0] = inda[e1];
```

```
              if (e1 == 2)
                edgeData[numEdges].pointIndices[1] = inda[0];
              else
                edgeData[numEdges].pointIndices[1] = inda[e1+1];

              edgeData[numEdges].triangleIndices[0] = faceA;
              edgeData[numEdges].triangleIndices[1] = faceB;

              numEdges++;

              if (numEdges >= kMaxEdges)
                DoError("\pCalcTriMeshEdges: numEdges >= kMaxEdges ");
            }
          }    // e2
        }      // e1
    }          // face2

            /************************************/
            /* NOW CHECK FOR EDGES ON EMPTY SPACE */
            /************************************/

    for (m = 0; m < 3; m++)
    {
      if (edgeOnSpace[m])
      {
        edgeData[numEdges].pointIndices[0] = inda[m];
        if (m == 2)
          edgeData[numEdges].pointIndices[1] = inda[0];
        else
          edgeData[numEdges].pointIndices[1] = inda[m+1];

        edgeData[numEdges].triangleIndices[0] = faceA;
        edgeData[numEdges].triangleIndices[1] = faceA;
        numEdges++;
        if (numEdges >= 2000)
          DoFatalAlert("\pCalcTriMeshEdges: m-numEdges >= 2000");
      }
    }
  }          // face1



      /* UPDATE TRIMESH DATA */

  if (numEdges > 0)
  {
    triMeshData.numEdges = numEdges;
    triMeshData.edges = &edgeData[0];

    Q3TriMesh_SetData(theTriMesh,&triMeshData);
  }


      /* CLEANUP */

  Q3TriMesh_Empty(&triMeshData);
}
```

```
/************* COMPARE POINTS ******************/
//
// Returns true if input points are close enough based
// on tolerance value.
//

Boolean ComparePoints(TQ3Point3D *p1, TQ3Point3D *p2,
                      float tolerance)
{
float  dx,dy,dz;

  dx = fabs(p1->x - p2->x);
  dy = fabs(p1->y - p2->y);
  dz = fabs(p1->z - p2->z);

  if ((dx <= tolerance) && (dy <= tolerance) && (dz <= tolerance))
    return(true);

  return(false);
}


/********** COMPARE VECTORS ********************/
//
// Returns true if input vectors are close enough based
// on tolerance value.
//

Boolean CompareVectors(TQ3Vector3D *p1, TQ3Vector3D *p2,
                       float tolerance)
{
float  dx,dy,dz;

  dx = fabs(p1->x - p2->x);
  dy = fabs(p1->y - p2->y);
  dz = fabs(p1->z - p2->z);

  if ((dx <= tolerance) && (dy <= tolerance) && (dz <= tolerance))
    return(true);

  return(false);
}
```

The code is a little complex because of the multiple nested loops, but the logic is simple. We compare each triangle against all other triangles. If two triangles share a common side then we see if the face normals are the same. If the face normals are different, then we assume that the shared side is a visible edge and we generate edge data for it. When no triangle shares a side with the current triangle, then this side also becomes an edge which we want displayed.

The two utility functions `ComparePoints` and `CompareVectors` determines if the input data are “close enough” to be considered a match.

SUBMITTING TRIMESHES

There is one additional trick you can do with TriMeshes to get a little more performance:

> When submitting your TriMeshes and if you are using the QuickDraw 3D Interactive Renderer, try to submit your largest TriMesh first.

The reason for this lies in the way that the Interactive Renderer manages memory. When a TriMesh is submitted for rendering, the Interactive Renderer allocates enough temporary memory to work with that TriMesh. If the next submitted TriMesh in the same rendering loop is larger than the previous TriMesh, then the Interactive Renderer has to reallocate a larger block of temporary memory to work with.

So, if you submit the largest TriMesh first, then all subsequent smaller TriMeshes will already have enough temporary memory to work with and the Interactive Renderer will not need to do any new memory allocation. Depending on your specific circumstances, you may see up to a 3-5% speed boost if you use this optimization.

SUMMARY

In this chapter we learned about the TriMesh geometry type which is new to QuickDraw 3D 1.5. TriMesh is the preferred geometry type if you want the maximum speed in your 3D applications.

To make sure your TriMesh geometries are built for maximum performance, follow these rules:

1. Only use one material per TriMesh.
2. Apply only face normal attributes to triangles.
3. Apply only vertex normals and vertex u/v coordinate attributes to points.
4. Smooth your models so that vertices will be shared.
5. If possible, attempt to construct Strips and Fans in your TriMeshes so that hardware acceleration will be improved.
6. Try to submit your largest TriMesh first to improve the Interactive Renderer's memory management.

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

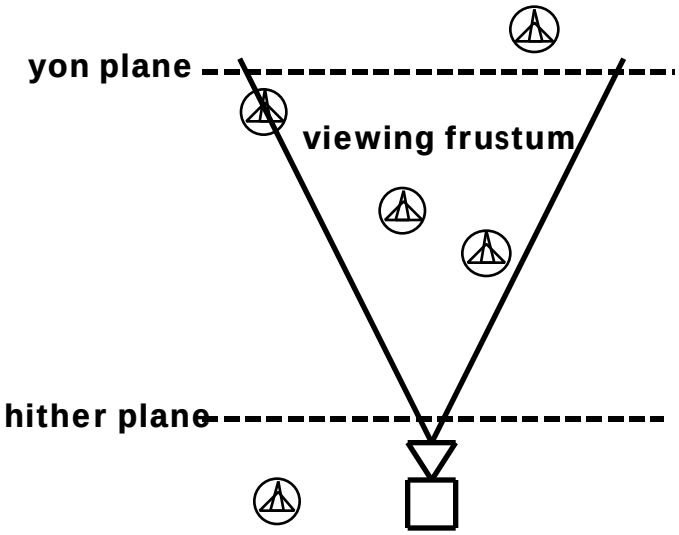
With a little optimization I'm about to show you, we can cull the above-described airplane in only 2 transforms instead of 40. Even if we have a very complex airplane made up of 100 different TriMeshes, we can still cull the model with only 2 transforms; not the 800 that QuickDraw 3D would require. When 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.

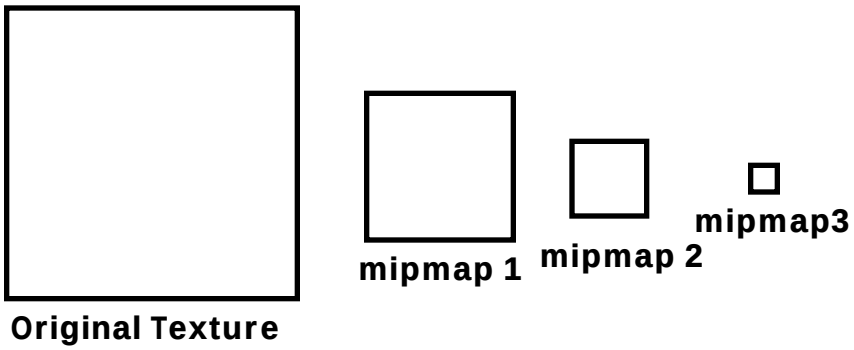
A 32-bit texture uses more VRAM than a 16-bit texture, and anything that's bigger means slower. The more memory that gets touched by the hardware, the slower it runs. That general rule can be applied to anything in your code.

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.

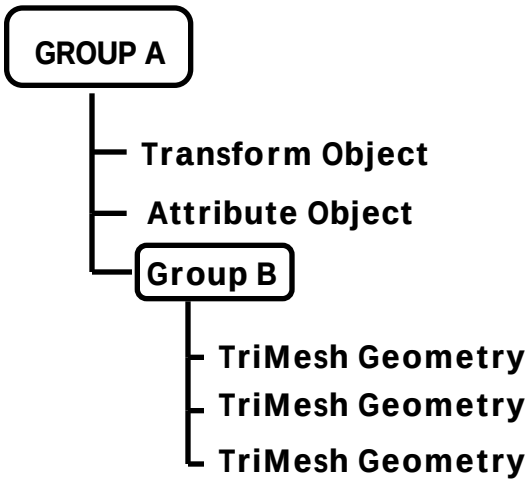
HIERARCHIES & STATE INFORMATION

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.

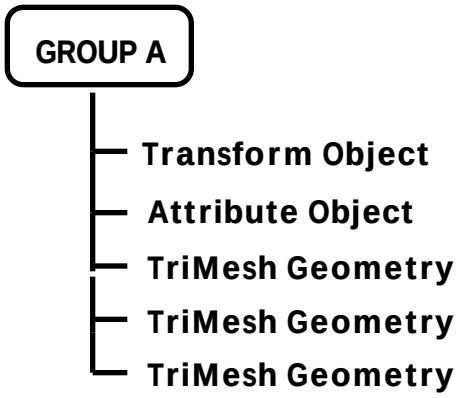
6. Group A is also done, so we pop the state information off the stack again.

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.

Topic 3:

Cool Algorithms

QuickDraw 3D does everything needed to create and render 3D scenes, but it does not have functions to do a lot of application-specific stuff like collision detection and special effects. This chapter contains many different algorithms which I have found to be very useful in the QuickDraw 3D applications I have written.

LINE INTERSECT PLANE

When doing 3D collision detection it is often necessary to calculate the intersection of a line and a plane, or to determine if a line segment intersects a plane. QuickDraw 3D does not have a built-in function for doing this, so we need to do a little extra coding on our own.

THE PLANE EQUATION OF A TRIANGLE

Once again, I'm going to avoid getting into the mathematics of a function here in this document, but suffice to say that the plane equation of a triangle is just a face normal and a constant. These four values are all that is needed to represent the plane of a triangle in mathematical terms.

The following code will calculate the plane equation of the input triangle:

```
/************** CALC PLANE EQUATION OF TRIANGLE ***************/
//
```

```
// INPUT: plane  = pointer to structure to store plane equation into.
//        p3..p1 = pointers to the 3 points in a triangle (clockwise)
//

void CalcPlaneEquationOfTriangle(TQ3PlaneEquation *plane,
                                 TQ3Point3D *p3,
                                 TQ3Point3D *p2,
                                 TQ3Point3D *p1)
{
float  pq_x,pq_y,pq_z;
float  pr_x,pr_y,pr_z;
float  p1x,p1y,p1z;

       /* CALC 2 EDGE VECTORS */

  p1x = p1->x;                           // get point #1
  p1y = p1->y;
  p1z = p1->z;

  pq_x = p1x - p2->x;                    // calc vector pq
  pq_y = p1y - p2->y;
  pq_z = p1z - p2->z;

  pr_x = p1->x - p3->x;                  // calc vector pr
  pr_y = p1->y - p3->y;
  pr_z = p1->z - p3->z;


       /* CALC CROSS PRODUCT FOR THE FACE'S NORMAL */

  plane->normal.x = (pq_y * pr_z) - (pq_z * pr_y);
  plane->normal.y = ((pq_z * pr_x) - (pq_x * pr_z));
  plane->normal.z = (pq_x * pr_y) - (pq_y * pr_x);


       /* MAKE SURE FACE NORMAL IS NORMALIZED */

  Q3Vector3D_Normalize(&plane->normal,&plane->normal);


     /* CALC DOT PRODUCT FOR PLANE CONSTANT */

  plane->constant =  ((plane->normal.x * p1x) +
                      (plane->normal.y * p1y) +
                      (plane->normal.z * p1z));
}
```

Most of the above code is simply calculating the face normal of the triangle. If you already have the face normal of the triangle, then you don't need to recalculate it and you only need to calculate the plane constant at the end of the function.

TESTING THE INTERSECTION

Determining the intersection coordinate of a line segment and a plane is fairly simple and occurs in two basic steps:

1. If both endpoints of the line segment are on the same side of the plane, then no intersection could have occurred.

2. Calculate the intersection coordinate.

The code to do this is as follows:

```
/************ INTERSECT PLANE & LINE SEGMENT ***************/
//
// Returns TRUE if the input line segment intersects the plane.
//
// INPUT:   plane     = pointer to plane equation
//          v1x/y/z   = coords of line segment endpoint #1
//          v2x/y/z   = coords of line segment endpoint #2
//
// OUTPUT:  outPoint  = pointer to point to receive data
//

Boolean IntersectionOfLineSegAndPlane(TQ3PlaneEquation *plane,
                            float v1x, float v1y, float v1z,
                            float v2x, float v2y, float v2z,
                            TQ3Point3D *outPoint)
{
int    a,b;
float  r;
float  nx,ny,nz,planeConst;
float  vBAx, vBAy, vBAz, dot, lam;


     /*****************************************/
     /* SEE IF LINE SEGMENT CROSSES PLANE AT ALL */
     /*****************************************/

       /* GET PLANE EQUATION DATA */

  nx = plane->normal.x;
  ny = plane->normal.y;
  nz = plane->normal.z;
  planeConst = plane->constant;


     /* DETERMINE SIDENESS OF VERT1 */

  r = -planeConst;
  r += (nx * v1x) + (ny * v1y) + (nz * v1z);
  a = (r < 0.0f) ? 1 : 0;
```

```
    /* DETERMINE SIDENESS OF VERT2 */

  r = -planeConst;
  r += (nx * v2x) + (ny * v2y) + (nz * v2z);
  b = (r < 0.0f) ? 1 : 0;

    /* SEE IF LINE CROSSES PLANE (INTERSECTS) */

  if (a == b)
    return(false);


    /*****************************************************/
    /* LINE INTERSECTS, SO CALCULATE INTERSECTION POINT */
    /*****************************************************/

         /* CALC LINE SEGMENT VECTOR BA */

  vBAx = v2x - v1x;
  vBAy = v2y - v1y;
  vBAz = v2z - v1z;


      /* DOT PRODUCT OF PLANE NORMAL & LINE SEGMENT VECTOR */

  dot = (nx * vBAx) + (ny * vBAy) + (nz * vBAz);


      /* IF VALID, CALC INTERSECTION POINT */

  if (dot != 0.0f)
  {
    lam = planeConst;

              // calc dot product of plane normal & 1st vertex
    lam -= (nx * v1x) + (ny * v1y) + (nz * v1z);

              // divide by previous dot for scaling factor
    lam /= dot;

              // calc intersect point
    outPoint->x = v1x + (lam * vBAx);
    outPoint->y = v1y + (lam * vBAy);
    outPoint->z = v1z + (lam * vBAz);
    return(true);
  }

  /* DOT == 0, THUS LINE IS PARALLEL TO PLANE SO NO INTERSECTION */

  else
    return(false);
}
```

REFLECTION MAPPING

Reflection mapping is often referred to as Environment mapping. In case you are unfamiliar with the term, it is a method of putting environmental reflections onto 3D models. For example, if you have a chrome ball floating in outer space, you would probably want the reflections of the stars and planets visible on the chrome sphere. Or if you have a shiny spoon on a table in a room, you'd probably want a reflected image of the room mapped onto the spoon.

In a high-end rendering package, reflections like this would be calculated via raytracing or some other complex algorithm. Since doing real-time raytracing for reflections is out of the question with today's technology, we have to resort to a much faster, yet much less accurate method for generating reflections.

There are basically two types of reflection mapping. The first type of reflection mapping uses six different environment texture maps, one for each "side" of a scene (front, back, top, bottom, left, and right). The second type only requires one texture map to represent the scene, and the calculations are very simple and can easily be done in real-time with QuickDraw 3D.

GENERATING THE REFLECTION MAP

The first step to performing reflection mapping is to generate an actual texture map to use. This is by far the most difficult part of the process. Writing the code to do the mapping is much easier than building a good texture.

Textures for reflection maps are not simple photographs of an ocean, room, or space. The textures we need to use are the reflections of a simple photograph on the surface of a chrome ball. In other words, an environment/reflection map of an ocean does not look like this....

Figure 3.0

A typical image of the ocean

... rather, it looks like this...

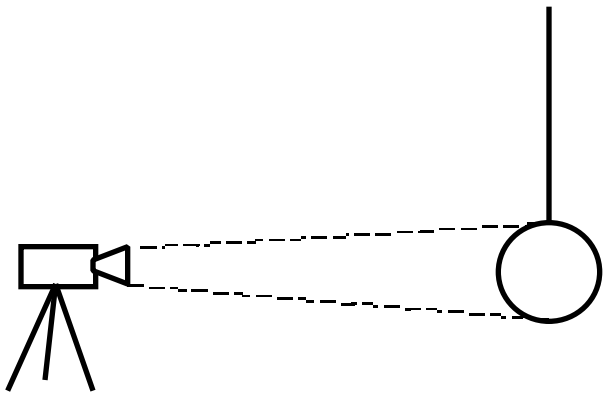
Figure 3.1

The same image converted into a reflection map

The texture map is the image of a chrome sphere which is reflecting our source image.

If you really want to generate an image like this correctly, the best way to do it is to find yourself a gigantic chrome ball which you can aim a camera at and take a picture of. I've seen some strange roadside art stores which sell large chrome Christmas ornaments about two feet in diameter - these will work well. Just take the chrome ball and hang it from a string inside a room, stand back, zoom into the ball and take a picture. This should generate a perfect reflection map of the room. The only hitch is that you might see you and your camera in the image unless you've camouflaged yourself.

Figure 3.2



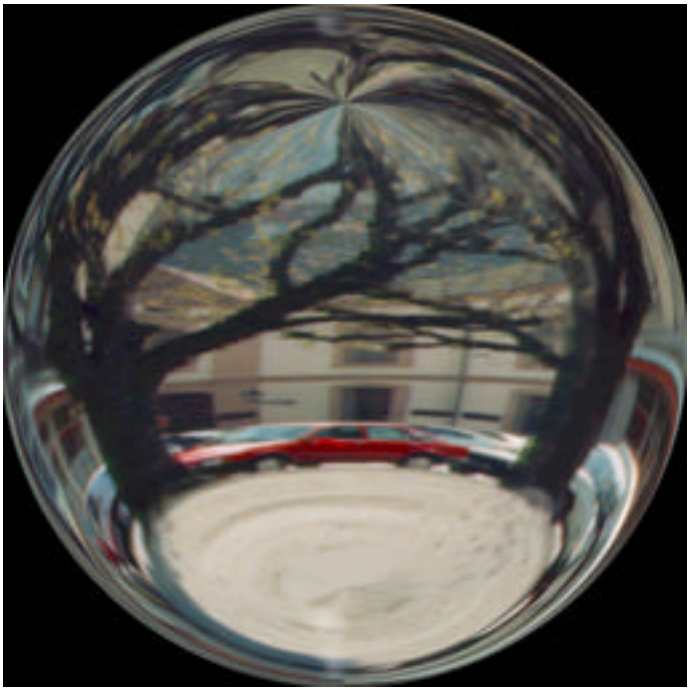
Take a close-up picture of a chrome ball hanging froma string makes perfect reflection maps.

For those of us who don't have the luxury of owning a giant chrome ball, we must find other ways to make our reflection maps. The only other way I've found to generate these is to actually model a chrome ball in a 3D rendering package and render it. Unfortunately, most 3D packages seem to have difficulty in correctly calculating the reflection of an environment on a chrome sphere. Taking a 2D image and treating it like a 3D environment is something of a mathematical hack and as a result, it is very difficult to correctly render a chrome sphere. 3D renderers will generate a chrome sphere which "looks" correct, but is actually very wrong.

Personally, I like to use Infini-D to render my chrome spheres. I've found it to be a very accurate renderer for such things. If you are using a different renderer you should be careful. If your reflection mapped models don't look right in QuickDraw 3D, it's probably because your renderer didn't generate a very accurate reflection map.

Once your map has been created, make sure that you crop out any black space around the edges. The edge of the chrome sphere must make contact with the edges of the texture map. Also make sure that the background around the sphere is black.

Figure 3.3



A completed reflection map with edges cropped.

Even Infini-D doesn't do a perfect job of rendering our sphere. Notice the "pinching" of the texture at the top-center of the sphere. You wouldn't see this on a real chrome ball, but that artifacting is what happens when we try to build a chrome ball with a 3D modeler. Other renderers pinch much worse or distort the image unrealistically.

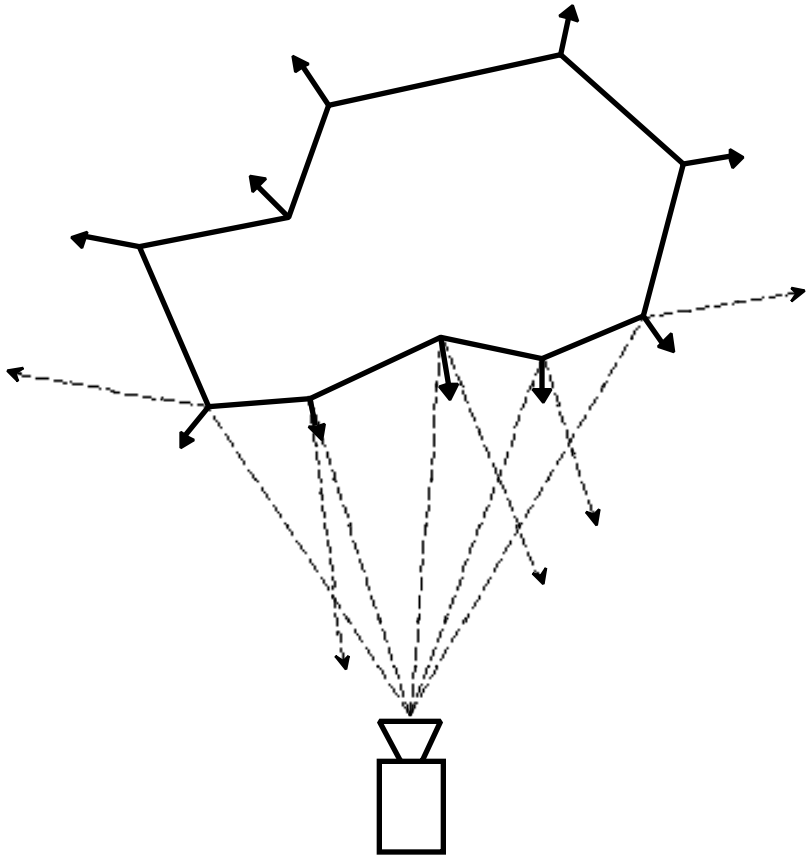
REFLECTION CODE

Like I said before, generating the reflection map is the hardest part. Now that we're past that, let's move on to the code.

The algorithm is simple:

- Calculate a vector from the camera to a vertex
- Reflect the vector off of that vertex
- Use the normalized reflected vector's x and y values as u and v texture coordinates.

Figure 3.4



Reflecting vectors off of the model.

Here is code which shows how this is done:

```
/***************** CALC REFLECTION UV COORDS **************/
//
// INPUT: numVertices    = # vertices to process
//        points         = pointer to vertex point list
//        normals        = pointer to vertex normals list
//        uvs            = pointer to uv coordinate list to store data
//

void CalcReflectionUVs(int numVertices, TQ3Point3D *points,
                       TQ3Vector3D *normals, TQ3Param2D *uvs)
{
float   camX,camY,camZ;
float   eyeVectorX,eyeVectorY,eyeVectorZ;
int     vertNum;

     /* GET CURRENT CAMERA COORDINATES */
```

```
  camX = gCamCoord.x;
  camY = gCamCoord.y;
  camZ = gCamCoord.z;


         /* CALC UV COORDINATE FOR EACH VERTEX */

  for (vertNum = 0; vertNum < numVertecies; vertNum++)
  {
           /* CALC VECTOR FROM CAMERA TO VERTEX */

       eyeVectorX = points[vertNum].x - camX;
       eyeVectorY = points[vertNum].y - camY;
       eyeVectorZ = points[vertNum].z - camZ;


           /* REFLECT VECTOR AROUND VERTEX NORMAL */

       ReflectVector(eyeVectorX, eyeVectorY, eyeVectorZ,
                   &normals[vertNum],&reflectedVector);


             /* CALC UV FROM REFLECTION VECTOR */

       uvs[vertNum].u = (reflectedVector.x * .5f) + .5f;
       uvs[vertNum].v = (-reflectedVector.y * .5f) + .5f;
  }
}


/********************** REFLECT VECTOR ************************/
//
// Given a view vector and a vertex normal vector, this function
// returns the reflected vector.
//
// INPUT: viewX/Y/Z   = view vector from camera to vertex
//        normal      = pointer to vertex normal
//        out         = pointer where to store reflected vector.
//
//

void ReflectVector(float viewX, float viewY, float viewZ,
                   TQ3Vector3D *normal, TQ3Vector3D *out)
{
float       dotProduct
TQ3Vector3D *reflected;

    /* COMPUTE DOT PRODUCT OF VERTEX NORMAL AND VIEW VECTOR */
    //
    // this calculates the angle between the two vectors
    //

  dotProduct = normal->x * viewX;
  dotProduct += normal->y * viewY;
  dotProduct += normal->z * viewZ;
```

```
    /* DOUBLE THE ANGLE TO CAUSE IT TO REFLECT AROUND VERTEX NORMAL */

  dotProduct += dotProduct;


    /* COMPUTE THE REFLECTED VECTOR & NORMALIZE */

  reflected.x = normal->x * dotProduct - viewX;
  reflected.y = normal->y * dotProduct - viewY;
  reflected.z = normal->z * dotProduct - viewZ;

  Q3Vector3D_Normalize(&reflected.x, out);
}
```

Most of the work is done by the `ReflectVector` function. This function takes care of reflecting the view vector around the vertex normal. The reflected vector has x and y values between -1.0 and 1.0 which does not match the valid uv coordinate range of 0.0 to 1.0, therefore, we adjust the x and y values to fit in 0.0 to 1.0 with this code in `CalcReflectionUVs`.

```
      uvs[vertNum].u = (reflectedVector.x * .5f) + .5f;
      uvs[vertNum].v = (-reflectedVector.y * .5f) + .5f;
```

REFLECTION MAPPING TRIMESHES

You now have the basic algorithm for reflection mapping vertices, but now I'd like to present the code for reflection mapping an actual TriMesh.

We cannot use retained mode for submitting these TriMeshes for rendering. In order to correctly calculate the reflection mapping we need to know the world space coordinates of all of the vertices. The only way to get these values is to transform the object by hand.

The following code takes any QuickDraw 3D object and processes any data contained in it:

```
/****************** REFLECTION MAP MY OBJECT ******************/

TQ3Matrix4x4   gWorkMatrix;
```

```
void ReflectionMapMyObject(TQ3Object obj)
{
       /* INIT WORK MATRIX TO IDENTITY MATRIX */

  Q3Matrix4x4_SetIdentity(&gWorkMatrix);

       /* PROCESS EVERYTHING INSIDE THIS OBJECT */

  CalcEnvMap_Recurse(thisNodePtr->BaseGroup);
}


/****************** CALC ENV MAP_RECURSE *********************/
//
// Recusively parses the input object looking for
// important data to process.
//
// INPUT:   obj = some QD3D object
//

static void CalcEnvMap_Recurse(TQ3Object obj)
{
TQ3Matrix4x4     transform;
TQ3GroupPosition position;
TQ3Object        object,baseGroup;
TQ3Matrix4x4     stashMatrix;


         /*******************************/
         /* SEE IF ACCUMULATE TRANSFORM */
         /*******************************/

  if (Q3Object_IsType(obj,kQ3ShapeTypeTransform))
  {
    Q3Transform_GetMatrix(obj,&transform);
    Q3Matrix4x4_Multiply(&transform,&gWorkMatrix,&gWorkMatrix);
  }
         /*********************************/
         /* SEE IF FOUND TRIMESH GEOMETRY */
         /*********************************/

  else
  if (Q3Object_IsType(obj,kQ3ShapeTypeGeometry))
  {
    if (Q3Geometry_GetType(obj) == kQ3GeometryTypeTriMesh)
      EnvironmentMapTriMesh(obj);
  }

       /****************************/
       /* SEE IF RECURSE SUB-GROUP */
       /****************************/

  else
  if (Q3Object_IsType(obj,kQ3ShapeTypeGroup))
  {
    baseGroup = obj;
    stashMatrix = gWorkMatrix;           // push matrix
```

```
    Q3Group_GetFirstPosition(obj, &position);
    while(position != nil)              // scan all objects in group
    {
         /* GET OBJECT REFERENCE FROM GROUP */

       Q3Group_GetPositionObject (obj, position, &object);
       if (object != nil)
       {
            /* RECURSE THIS OBJ */

          CalcEnvMap_Recurse(object);

          /* DISPOSE OF OUR REFERENCE TO THIS SUB-GROUP */

          Q3Object_Dispose(object);
       }

          /* GET NEXT OBJECT IN THE GROUP */

       Q3Group_GetNextPosition(obj, &position);
    }
    gWorkMatrix = stashMatrix;          // pop matrix
  }
}
```

The above code starts by initializing an identity matrix. Since we're going to be transforming the geometry by hand, we need to accumulate a transform matrix to use. The recursive code, parses through any groups and when a transform object is encountered the work matrix is updated. When a TriMesh is encountered, it calls a function which applies reflection mapping to it:

```
/************* ENVIRONMENT MAP TRIMESH ***********/
//
// Transforms the TriMesh by the current transform matrix
// and then applies reflection mapping to the uv texture
// coordinates before submitting it for rendering.
//

static void EnvironmentMapTriMesh(TQ3Object theTriMesh)
{
TQ3Status        status;
unsigned long    numVertecies,vertNum, numFaces, faceNum;
TQ3Point3D       *vertexList;
TQ3TriMeshData   triMeshData;
TQ3TriMeshAttributeData *attribList,*faceAttribList;
short            numVertAttribTypes,a, numFaceAttribTypes;
TQ3Vector3D      *normals,reflectedVector;
TQ3Param2D       *uvList;
float            camX,camY,camZ;
float            eyeVectorX,eyeVectorY,eyeVectorZ;
float            M00, M01, M02;
```

```
float            M10, M11, M12;
float            M20, M21, M22;
float            x,y,z;
TQ3Matrix4x4   tempm,invTranspose;



        /********************/
        /* GET TRIMESH INFO */
        /********************/

  Q3TriMesh_GetData(theTriMesh, &triMeshData);
  numVertecies       = triMeshData.numPoints;
  numFaces           = triMeshData.numTriangles;
  vertexList         = triMeshData.points;
  numVertAttribTypes = triMeshData.numVertexAttributeTypes;
  attribList         = triMeshData.vertexAttributeTypes;
  numFaceAttribTypes = triMeshData.numTriangleAttributeTypes;
  faceAttribList     = triMeshData.triangleAttributeTypes;

        /* FIND UV ATTRIBUTE LIST */

  for (a = 0; a < numVertAttribTypes; a++)
  {
    if ((attribList[a].attributeType == kQ3AttributeTypeSurfaceUV) ||
      (attribList[a].attributeType == kQ3AttributeTypeShadingUV))
    {
      uvList = attribList[a].data;       // point to list of normals
      goto got_uv;
    }
  }
  DoError("\pThis TriMesh doesnt have a texture map!");

got_uv:

       /*******************************/
       /* TRANSFORM THE BOUNDING BOX */
       /*******************************/

  Q3Point3D_To3DTransformArray(&triMeshData.bBox.min, &gWorkMatrix,
                               &triMeshData.bBox.min, 2,
                               sizeof(TQ3Point3D), sizeof(TQ3Point3D));


        /***************************/
        /* TRANSFORM VERTEX COORDS */
        /***************************/

  Q3Point3D_To3DTransformArray(vertexList, &gWorkMatrix, vertexList,
                              numVertecies, sizeof(TQ3Point3D),
                              sizeof(TQ3Point3D));


        /***************************/
        /* TRANSFORM VERTEX NORMALS */
        /***************************/

        /* CALC INVERSE-TRANSPOSE MATRIX */
```

```
Q3Matrix4x4_Invert(&gWorkMatrix, &tempm);
Q3Matrix4x4_Transpose(&tempm,&invTranspose);

       /* LOAD MATRIX INTO REGISTERS */

M00 = invTranspose.value[0][0];
M01 = invTranspose.value[0][1];
M02 = invTranspose.value[0][2];
M10 = invTranspose.value[1][0];
M11 = invTranspose.value[1][1];
M12 = invTranspose.value[1][2];
M20 = invTranspose.value[2][0];
M21 = invTranspose.value[2][1];
M22 = invTranspose.value[2][2];


     /* FIND THE VERTEX NORMALS ATTRIBS */

for (a = 0; a < numVertAttribTypes; a++)
{
   if (attribList[a].attributeType == kQ3AttributeTypeNormal)
   {
      normals = attribList[a].data;

         /* TRANSFORM & NORMALIZE ALL NORMALS */

      for (vertNum = 0; vertNum < numVertecies; vertNum++)
      {
         x = normals[vertNum].x;
         y = normals[vertNum].y;
         z = normals[vertNum].z;

         normals[vertNum].x = x * M00 + y * M10 + z * M20;
         normals[vertNum].y = x * M01 + y * M11 + z * M21;
         normals[vertNum].z = x * M02 + y * M12 + z * M22;

         Q3Vector3D_Normalize(&normals[vertNum],&normals[vertNum]);
      }
      break;
   }
}


         /*************************/
         /* TRANSFORM FACE NORMALS */
         /*************************/

         /* FIND FACE NORMAL ATTRIBS */

for (a = 0; a < numFaceAttribTypes; a++)
{
   if (faceAttribList[a].attributeType == kQ3AttributeTypeNormal)
   {
      normals2 = faceAttribList[a].data;

         /* TRANSFORM ALL FACE NORMALS */
```

```
      for (faceNum = 0; faceNum < numFaces; faceNum++)
      {
         x = normals2[faceNum].x;
         y = normals2[faceNum].y;
         z = normals2[faceNum].z;

         normals2[faceNum].x = x * M00 + y * M10 + z * M20;
         normals2[faceNum].y = x * M01 + y * M11 + z * M21;
         normals2[faceNum].z = x * M02 + y * M12 + z * M22;
         Q3Vector3D_Normalize(&normals2[vertNum],&normals2[vertNum]);
      }
      break;
   }
  }

   /*****************************/
   /* CALC UVS FOR EACH VERTEX */
   /*****************************/

  CalcReflectionUVs(numVertices, &vertexList[0],
                    &normals[0],&uvList[0]);


   /*********************************/
   /* SUBMIT TRIMESH FOR RENDERING  */
   /*********************************/

  Q3TriMesh_Submit(&triMeshData, gMyViewObject);
}
```

This code does essentially what QuickDraw 3D does internally when a TriMesh is submitted for rendering. It applies the current transform matrix to the bounding box, points, and normals. Once we have the world-space coordinates for the vertices, we calculate the vertex uv's for the reflection mapping using code we discussed earlier.

Note that I used the QuickDraw 3D function `Q3Point3D_To3DTransformArray` to transform the TriMesh's points, but I wrote my own function to transform the vertex and face normals. I had to do this because for some reason QuickDraw 3D does not have a `Vector3D` transform array function. Luckily, however, QuickDraw 3D does supply functions for calculating the inverse-transform of a matrix. You must calculate the inverse transform of a matrix before transforming normals because the original matrix may contain scaling and translating information which you do not want to apply to a vector transform. The inverse-transform of a matrix will yield a matrix which only rotates - the scale and translate information has been cancelled out.

That's pretty much it. You now know how to quickly reflection map a TriMesh geometry in QuickDraw 3D.

"OVERLAYS"

For the purpose of this discussion, an overlay is considered to be an image which appears at the same place on the screen regardless of the camera's orientation. An example of an overlay is the health bar in Weekend Warrior or the overhead-map in Nanosaur:

Figure 3.5

A screenshot from Nanosaur which shows the map overlay.

Normally, putting such things on the screen would be trivial if you were using a custom 3D engine or RAVE directly. You would simply draw the polygons to the screen coordinates you specify. With

QuickDraw 3D, however, this is impossible. You cannot tell QuickDraw 3D to draw geometry in screen coordinates, only in world coordinates. So, to make this work we need to know the world-space coordinates to place an object such that it will be drawn at the desired screen coordinate.

Calculating the world-space coordinate to put an overlay is a straightforward task. Here are the steps:

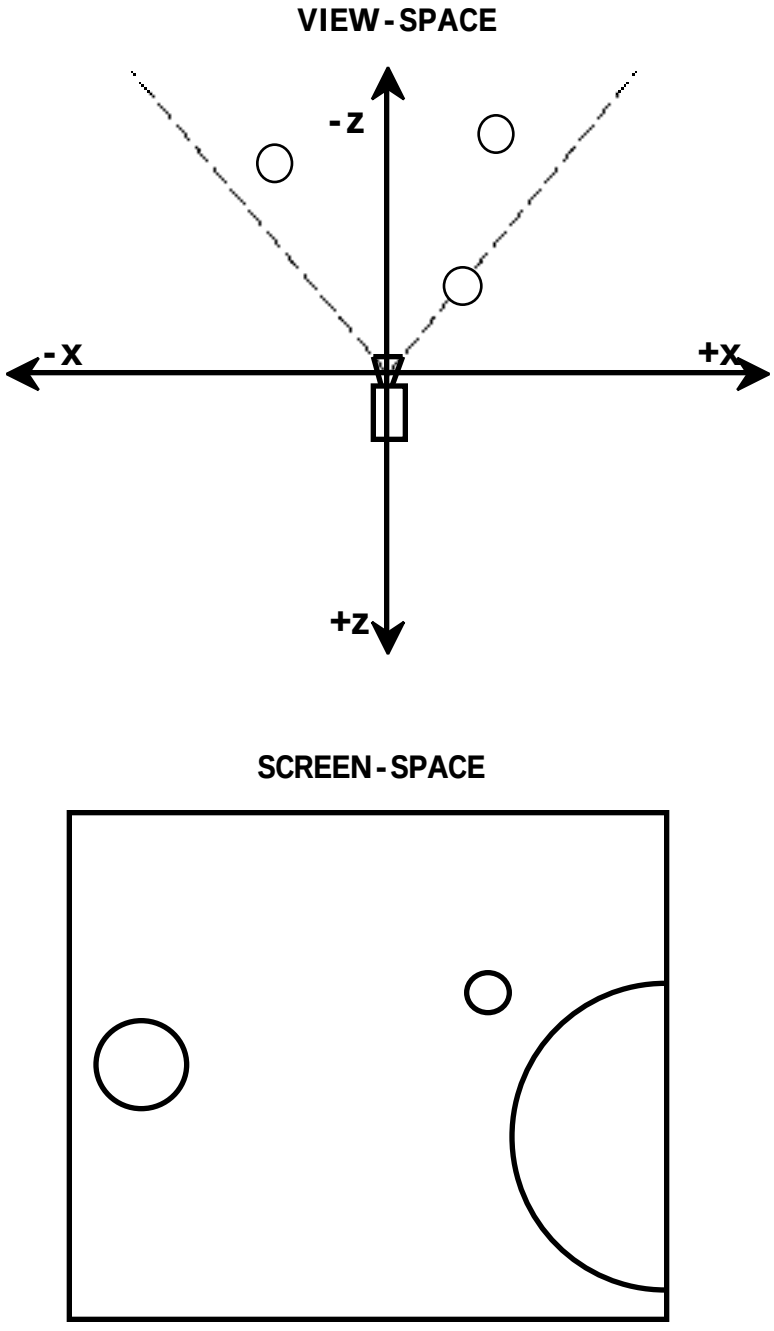
1. Calculate the inverse of the camera's `world->view` matrix which gives you a `view->world` matrix.

2. Place the overlay at the desired view-space coordinate.

3. Transform the overlay to world space with the `view->world` matrix you calculated in step 1.

View space is the closest thing to screen space that we can use because there is no way to calculate a frustum->world matrix from the camera's world->frustum matrix. That matrix simply can't be inverted correctly, but luckily the world->view matrix can be inverted just fine. You'll need to do some trial and error to get the right view space coordinates for your overlays. Your z value should be just a tiny bit less than your camera's hither value because we want the overlays to be as close to the camera lens as possible. The x and y values will vary depending on where on the screen you want your overlay to appear, and the camera's fov will cause the x, y values to vary.

Figure 3.6



The relation between view-space and screen-space.

```
/******** CALC OVERLAY TRANSFORM MATRIX **********/

void CalcOverlayTransformMatrix(TQ3Point3D *viewSpaceCoord,
                                TQ3Matrix4x4 *outMatrix)
{
```

```
TQ3Matrix4x4   matrix;
TQ3Matrix4x4   worldToView,viewToWorld;

       /* CALCULATE VIEW->WORLD MATRIX */

  Q3Camera_GetWorldToView(viewPtr->cameraObject, &worldToView);
  Q3Matrix4x4_Invert(&worldToView, &viewToWorld)


     /* PUT OVERLAY AT DESIRED VIEW-SPACE COORDS */

  Q3Matrix4x4_SetTranslate(&matrix,
                           viewSpaceCoord->x,
                           viewSpaceCoord->y,
                           viewSpaceCoord->z);


     /* TRANSFORM TO WORLD-SPACE COORDINATES */

  Q3Matrix4x4_Multiply(&matrix,&viewToWorld, outMatrix);
}
```

As you can see, the code is very simple. The only tough part about doing overlays is figuring out what view-space coordinates to use, but that's just a matter of hacking at it to find values that work. When I do this, I usually start at x = 0, y = 0, z = HITHER-1.0 which should put the overlay in the middle of the screen. Then I gradually adjust the x and y values until the overlay is where I want it to be. Increasing x moves it right, decreasing x moves it left. Increasing y moves it up, decreasing y moves it down. In general, the x and y values will go from about -10 to +10 with 0 being the center of the screen. These values will vary depending on your camera's fov and the z coordinate you have assigned to the overlay.

You may also want to apply scaling and/or rotation to you overlays. The map overlay in Nanosaur used rotation to spin the map around. The following is a modified version of the above code which shows how to also apply scaling and rotation to your overlay.

```
/******** CALC OVERLAY TRANSFORM MATRIX 2 **********/

void CalcOverlayTransformMatrix2(TQ3Point3D *viewSpaceCoord,
                                 TQ3Vector3D *scale,
                                 TQ3Vector3D *rotation,
                                 TQ3Matrix4x4 *outMatrix)
{
TQ3Matrix4x4   matrix,matrix2,matrix3;
TQ3Matrix4x4   worldToView,viewToWorld;
```

```
      /* CALCULATE VIEW->WORLD MATRIX */

  Q3Camera_GetWorldToView(viewPtr->cameraObject, &worldToView);
  Q3Matrix4x4_Invert(&worldToView,& viewToWorld)


      /* APPLY SCALE */

  Q3Matrix4x4_SetScale(&matrix, scale->.x, scale->y, scale->.z);


      /* APPLY ROTATION */

  Q3Matrix4x4_SetRotate_XYZ(&matrix2, rotation->x, rotation->y,
                            rotation->z);
  Q3Matrix4x4_Multiply(&matrix,&matrix2,&matrix3);


    /* TRANSLATE TO DESIRED VIEW-SPACE COORD */

  Q3Matrix4x4_SetTranslate(&matrix2, viewSpaceCoord->x,
                          viewSpaceCoord->y, viewSpaceCoord->z);
  Q3Matrix4x4_Multiply(&matrix3,&matrix2,&matrix);


    /* TRANSFORM TO WORLD-SPACE COORDINATES */

  Q3Matrix4x4_Multiply(&matrix,&viewToWorld, outMatrix);
}
```

SPLINES

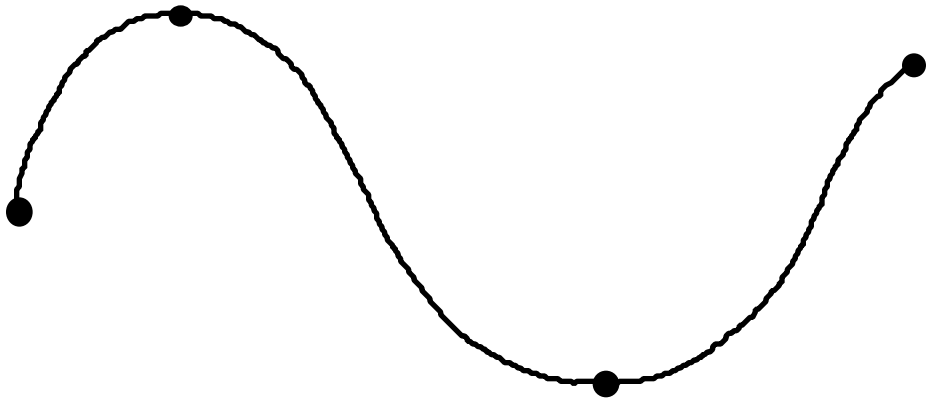
The number one technical question I've been asked by QuickDraw 3D developers is "how did you do the track in Gerbils?" The answer is splines. Splines are a wonderful thing and are very easy to generate. They're good for generating rollercoaster track as in Gerbils, but they're also good for creating smooth animation paths which you can move objects or the camera along.

SPLINE TYPES

There have been several books written only about splines. These books are very mathematically oriented and I've never been able to understand a single page of them. I don't know why mathematicians always need to take something simple and make it incomprehensible.

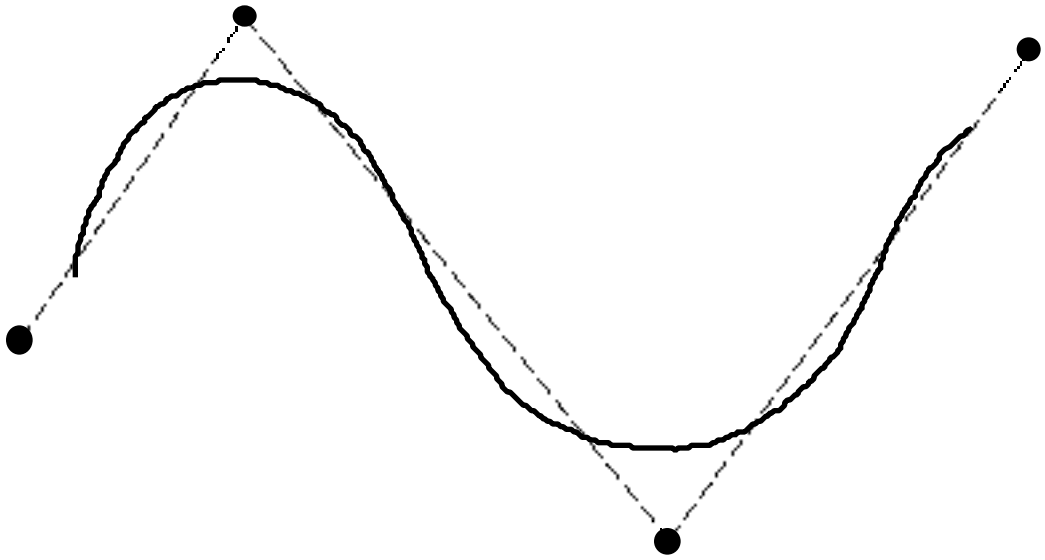
For our purposes, there are two types of splines: b-splines and cubic splines. B-splines are faster to generate than cubic splines, but b-splines do not pass through their control points whereas cubic splines do.

Figure 3.7



A cubic spline passes through it's control points

Figure 3.8



A b-spline does not pass through it's control points

A spline is just a series of points along a curve which are calculated based on the coordinates of "control points". I'm not going to explain the math behind these calculations since it's really not important how it works. The important thing is that it does work.

CALCULATING A B-SPLINE

To build a spline all you need is an array of control points and a function to interpolate the curve points between them. The following code shows how to generate a b-spline for the input array of control points:

```
#define   kMaxSplinePoints 5000

/********************** CALC SPLINE CURVE **********************/
//
// INPUT: numControlPoints =  # control points to build spline from
//        controlPoints    =  ptr to list of control points
//        numSubDivs       =  num spline points to calculate between each
//                            pair of control points
//
// OUTPUT:  numSplinePoints     =  total # spline points generated
//          theSpline           =  pointer to array holding all of
```

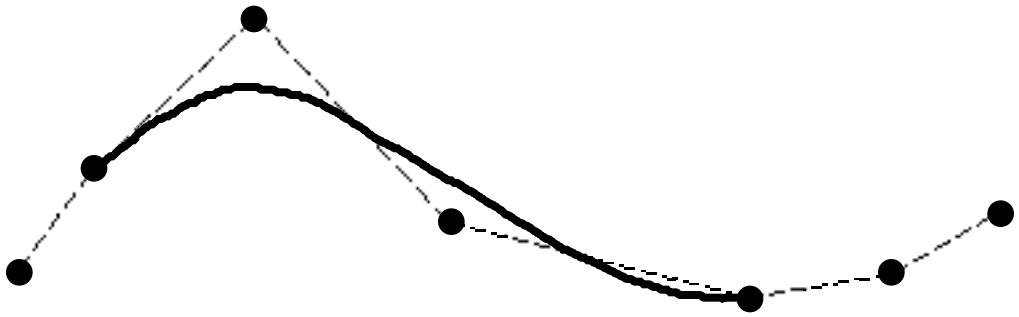
```
//                                the spline points
//

void CalcSplineCurve(int numControlPoints, TQ3Point3D *controlPoints,
                  float numSubDivs, int *numSplinePoints,
                  TQ3Point3D **theSpline)
{
float  t,tSquared,tCubed,a,b,c,d,incVal;
long   subCount;


      /* ALLOC MEMORY FOR SPLINE DATA */

  *theSpline = (TQ3Point3D *)NewPtr(sizeof(TQ3Point3D) *
                                 kMaxSplinePoints);
  if (theSpline == nil)
     DoError("\pCant allocate spline memory");

  *numSplinePoints = 0;            // start # points at zero

  incVal = 1.0/numSubDivs;         // calc fractional increment value


        /*****************************/
        /* SCAN THRU CONTROL POINTS */
        /*****************************/
        //
        // Start on 2nd control point and end on 3rd to last.
        //

  for (cp=1; cp < (numControlPoints-2); cp++)
  {
     for (t=0, subCount=0; subCount < numSubDivs; t+=incVal,subCount++)
     {
           /* MAKE SURE WE DON'T OVERFLOW SPLINE ARRAY */

      if (*numSplinePoints >= kMaxSplinePoints)
        DoError("\pToo many spline points!  Overflowed array!");


           /* DO MAGICAL CALCULATIONS */

      tSquared  = t*t;
      tCubed    = tSquared*t;
      a      = (-0.166*tCubed) + (0.5*tSquared) - (0.5*t) + 0.166;
      b      = (0.5*tCubed) - tSquared + 0.666;
      c      = (-0.5*tCubed) + (0.5*tSquared) + (0.5*t+0.166);
      d      = 0.166*tCubed;


          /* INTERPOLATE A POINT ON THE SPLINE */

      (*theSpline)[*numSplinePoints].x =       // calc x
                  (a * controlPoints[cp-1].x) +
                  (b * controlPoints[cp].x) +
                  (c * controlPoints[cp+1].x) +
                  (d * controlPoints[cp+2].x);
```

```
      (*theSpline)[*numSplinePoints].y =       // calc y
                  (a * controlPoints[cp-1].y) +
                  (b * controlPoints[cp].y) +
                  (c * controlPoints[cp+1].y) +
                  (d * controlPoints[cp+2].y);

      (*theSpline)[*numSplinePoints].z =       // calc z
                  (a * controlPoints[cp-1].z) +
                  (b * controlPoints[cp].z) +
                  (c * controlPoints[cp+1].z) +
                  (d * controlPoints[cp+2].z);

      *numSplinePoints++;                      // inc size of spline
    }
  }
}
```

We start by allocating memory to hold the gigantic array of interpolated spline points. The input value `numSubDivs` determines how many spline points we want to calculate between control points. The more subdivisions, the finer resolution the spline will be.

To calculate the actual spline, we simply process through the control points and perform the necessary calculations. Note that the first and last two control points are only used to start and end the curvature of the spline, but no spline points are interpolated for them.

Figure 3.9



There is no spline data generated for the first and last two control points

CALCULATING A CUBIC SPLINE

The code to build a cubic spline is much more complex than the code for b-splines, but the benefit of the cubic spline is that the curve will pass through each control point which makes it much easier to predict the shape of the curve. Additionally, cubic splines don't require three "extra" control points like b-splines do; all of the control points in a cubic spline will be part of the final spline.

```
/***************** CALCULATE CUBIC SPLINE *********************/
//
// INPUT: numControlPoints =  # control points to build spline
//        controlPoints    =  array of control points
//        numSubDivs       =  # of spline points to interpolate
//                            between two control points
//
// OUTPUT:  outNumPoints   =  # points in spline
//          outPoints      =  ptr to array of spline points
//

void CalculateCubicSpline(int numControlPoints,
                          TQ3Point3D *controlPoints, int numSubDivs
                          int *outNumPoints, TQ3Point3D **outPoints)
{
TQ3Point3D    **space;
TQ3Point3D    *a, *b, *c, *d;
TQ3Point3D    *h0, *h1, *h2, *h3, *hi_a;
short         imax;
float         t, dt;
TQ3Point3D    *v,*splinePoints;
long          i,i1,numSplinePoints;
OSErr         iErr;
long          bufferSize;


      /* MUST BE AT LEAST 4 CONTROL POINTS */

  if (numControlPoints < 4)
  {
     *outNumPoints = 0;
     *outPoints = nil;
     return;
  }


      /* MAKE SURE ARRAY IS LARGE ENOUGH FOR DATA */

  bufferSize = (numControlPoints * numSubDivs) + numControlPoints;
  *outPoints = splinePoints = NewPointer(bufferSize *
                                  sizeof(TQ3Point3D));
  if (splinePoints == nil)
     DoError("\pCannot allocate spline memory");
```

```
        /* ALLOCATE 2D ARRAY FOR CALCULATIONS */

space = (TQ3Point3D **) AllocPtr(8 * sizeof(TQ3Point3D *));
space[0] = (TQ3Point3D *) AllocPtr(8 * numControlPoints *
                                   sizeof(TQ3Point3D));
for (i = 1; i < 8; i++)
   space[i] = space[i-1] + numControlPoints;



   /************************************************/
   /* DO MAGICAL SPLINE CALCULATIONS ON CONTROL PTS */
   /************************************************/

h0 = space[0];
h1 = space[1];
h2 = space[2];
h3 = space[3];

a = space[4];
b = space[5];
c = space[6];
d = space[7];

for (i = 0; i < numControlPoints; i++)  // copy control pts into array
   d[i] = controlPoints[i];

for (i = 0, imax = numControlPoints - 2; i < imax; i++)
{
   h2[i].x = h2[i].y = h2[i].z = 1;
   h3[i].x = 3 *(d[i+ 2].x - 2 * d[i+ 1].x + d[i].x);
   h3[i].y = 3 *(d[i+ 2].y - 2 * d[i+ 1].y + d[i].y);
   h3[i].z = 3 *(d[i+ 2].z - 2 * d[i+ 1].z + d[i].z);
}
h2[numControlPoints - 3].x =
h2[numControlPoints - 3].y =
h2[numControlPoints - 3].z = 0;


a[0].x = a[0].y = a[0].z = 4;
h1[0].x = h3[0].x / a[0].x;
h1[0].y = h3[0].y / a[0].y;
h1[0].z = h3[0].z / a[0].z;
for (i = 1, i1 = 0, imax = numControlPoints - 2; i < imax; i++, i1++)
{
   h0[i1].x = h2[i1].x / a[i1].x;
   a[i].x = 4 - h0[i1].x;
   h1[i].x = (h3[i].x - h1[i1].x) / a[i].x;

   h0[i1].y = h2[i1].y / a[i1].y;
   a[i].y = 4 - h0[i1].y;
   h1[i].y = (h3[i].y - h1[i1].y) / a[i].y;

   h0[i1].z = h2[i1].z / a[i1].z;
   a[i].z = 4 - h0[i1].z;
   h1[i].z = (h3[i].z - h1[i1].z) / a[i].z;
}
```

```
b[numControlPoints - 3] = h1[numControlPoints - 3];

for (i = numControlPoints - 4; i >= 0; i--)
{
   b[i].x = h1[i].x - h0[i].x * b[i+ 1].x;
   b[i].y = h1[i].y - h0[i].y * b[i+ 1].y;
   b[i].z = h1[i].z - h0[i].z * b[i+ 1].z;
}

for (i = numControlPoints - 2; i >= 1; i--)
   b[i] = b[i - 1];

b[0].x = b[numControlPoints - 1].x =
b[0].y = b[numControlPoints - 1].y =
b[0].z = b[numControlPoints - 1].z = 0;
hi_a = a + numControlPoints - 1;

for (; a < hi_a; a++, b++, c++, d++)
{
   c->x = ((d+1)->x - d->x) -(2 * b->x + (b+1)->x) / 3;
   a->x = ((b+1)->x - b->x) / 3;

   c->y = ((d+1)->y - d->y) -(2 * b->y + (b+1)->y) / 3;
   a->y = ((b+1)->y - b->y) / 3;

   c->z = ((d+1)->z - d->z) -(2 * b->z + (b+1)->z) / 3;
   a->z = ((b+1)->z - b->z) / 3;
}

   /***********************************/
   /* NOW CALCULATE THE SPLINE POINTS */
   /***********************************/

a = space[4];
b = space[5];
c = space[6];
d = space[7];
v = splinePoints;

numSplinePoints = 0;
for (; a < hi_a; a++, b++, c++, d++)
{
   dt = 1.0 / numSubDivs;
   for (t = 0; t < 1; t += dt)
   {
          /* SAVE SPLINE POINT */

     v->x = ((a->x * t + b->x) * t + c->x) * t + d->x;
     v->y = ((a->y * t + b->y) * t + c->y) * t + d->y;
     v->z = ((a->z * t + b->z) * t + c->z) * t + d->z;
     v++;
     numSplinePoints++;         // inc # points
   }
}
*v++ = *d;                      // add final point
numSplinePoints++;
```

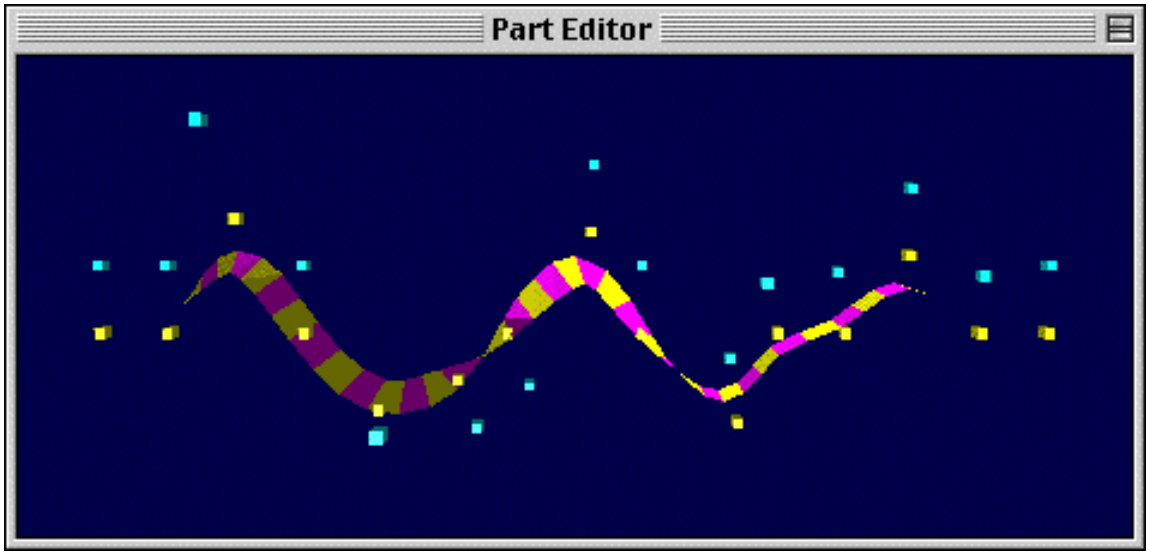
```
    /*************/
    /* ALL DONE */
    /*************/

  *outNumPoints = numSplinePoints;  // pass back # points in spline

  DisposePtr((Ptr)space[0]);        // dispose of the 2D array
  DisposePtr((Ptr)space);
}
```

As you can see, the code for generating a cubic-spline is significantly more complex. If 20% of the above code makes sense to you then you're doing better than most people. I've said it before and I'll say it again: it's not important that you understand why this code works, just be happy that it does and know how to use it. Knowing how to use a tool is much more important than knowing how the tool works.

GENERATING A ROLLERCOASTER TRACK

So, we come back to the question of how I built the rollercoaster tracks in Gerbils. Gerbils actually has a hidden "Part Editor" built into the application. This part editor will let you create your own sections of track and playing with this editor will help you understand the explanation that is to follow. To activate the hidden Part Editor in Gerbils, simply add Menu resource ID 128 to the MBAR resource. That's it! The Part Editor menu will appear when you run Gerbils and you will be able to see how it works.

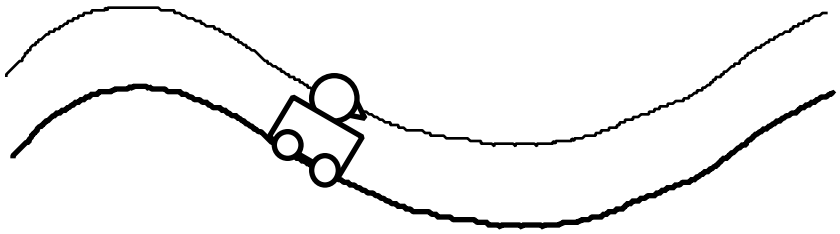
Figure 3.10



The Gerbils “Part Editor”

Technically, the track was a single spline in 3D space, but a spline is just a line - it isn’t a “track”. How do you know which way is up on a line? You can’t, that’s why it actually takes two splines to represent the rollercoaster track: a base spline and a “head” spline. The base spline is the spline that is the track, and the head spline is the spline where the rider’s head would be. These two splines run in parallel along the course of the track.

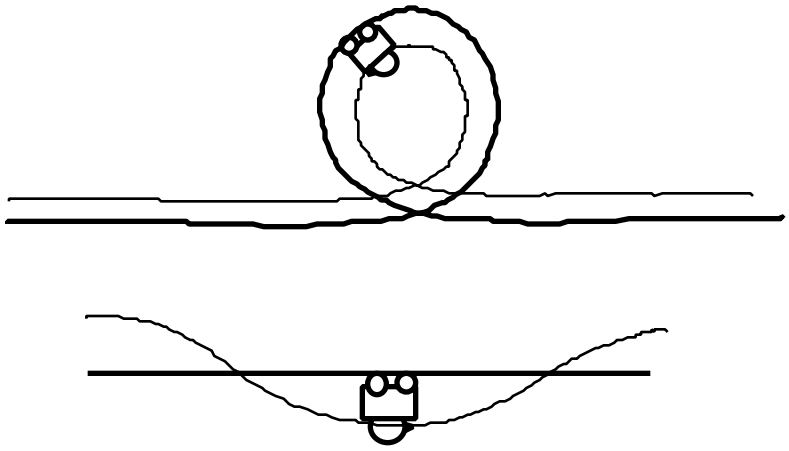
Figure 3.11



The track is made of the base and head splines.

The head spline is necessary so that we know which way is up. This allows us to correctly do corkscrews and loops which would be impossible to represent with just a single spline.

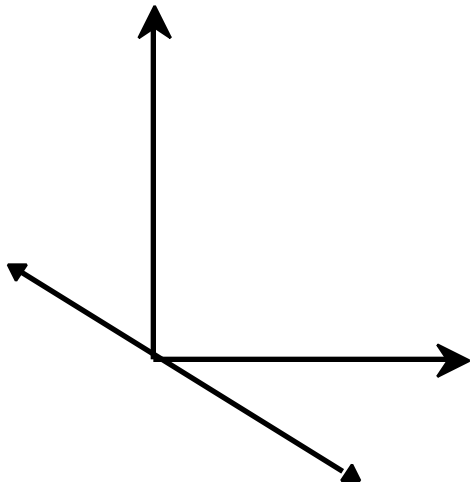
Figure 3.12



Two figures showing how the head spline controls the orientation of the rollercoaster for a loop and a corkscrew

The crossbars and rails on the track are easily generated by taking the cross product of the up-vector and the forward vector along these two splines. This will give us a vector from the base spline to the right rail. By negating this vector we get a vector to the left rail. Then it's just a matter of creating the geometry for the track.

Figure 4.13



The rails are found by taking the cross-product of
the up and forward vectors.

This is the basic principle behind generating the rollercoaster track, and the only other thing you need to know is how to align the rollercoaster cart (or Gerbil) to the track as it moves down it.

For starters, all the cart is doing is following the base spline. There's no math involved here, it's just a matter of indexing into the gigantic array of spline points. The more spline points you have interpolated, the smoother the cart will move down the track. In Gerbils, there were tens of thousands of interpolated points making up the spline, thus movement along the track was very smooth. The only tricky thing is keep thing the cart aimed correctly.

Luckily, there is a neat little matrix trick which lets us align the cart without hardly any effort at all. Using an up vector and a forward vector, we can just plug our vector values directly into a matrix. Using this matrix to transform the cart geometry will cause it to align to the direction of the track.

The following code shows how to build a matrix to align the cart to the track direction. This code comes in very useful in other 3D functions where you need to align geometry to an arbitrary vector.

```
/************ BUILD ALIGNMENT MATRIX ***************/

void BuildAlignmentMatrix(TQ3Vector3D *up, TQ3Vector3D *forward,
                          TQ3Point3D *coord, TQ3Matrix4x4 *theMatrix)
{
TQ3Vector3D theXAxis;

     /* SET THE TRANSLATE VALUES */

  theMatrix->value[3][0] = coord->x;
  theMatrix->value[3][1] = coord->y;
  theMatrix->value[3][2] = coord->z;



       /* SET FORWARD VECTOR */

  Q3Vector3D_Normalize(forward, forward);
  theMatrix->value[2][0] = forward->x;
  theMatrix->value[2][1] = forward->y;
  theMatrix->value[2][2] = forward->z;


       /* SET UP VECTOR */

  Q3Vector3D_Normalize(up, up);
  theMatrix->value[1][0] = up->x;
  theMatrix->value[1][1] = up->y;
  theMatrix->value[1][2] = up->z;


     /* CALC & SET THE X-AXIS VECTOR */
     //
     // Cross product of up & forward vector gives us
     // the X-axis vector
     //

  Q3Vector3D_Cross(up, forward, &theXAxis);
  Q3Vector3D_Normalize(&theXAxis,&theXAxis);

  theMatrix->value[0][0] = theXAxis.x;
  theMatrix->value[0][1] = theXAxis.y;
  theMatrix->value[0][2] = theXAxis.z;


       /* SET REMAINING CELLS */

  theMatrix->value[0][3] =
  theMatrix->value[1][3] =
  theMatrix->value[2][3] = 0;
  theMatrix->value[3][3] = 1;

}
```

We directly plug the forward-vector into the 3rd row of the matrix. The up-vector gets plugged into the 2nd row of the matrix, and the x-axis vector gets plugged into the 1st row. The x-axis vector is the only thing that we have to calculate, but it's pretty simple. We just need a vector which is perpendicular to the plane of the forward and up vectors, and a simple cross product calculation gives us this.

SUMMARY

This chapter discussed some algorithms which can improve the "coolness" of your QuickDraw 3D applications.

- The algorithm for finding the intersection of a line and a plane is useful for doing certain types of collision detection.
- Reflection / Environment mapping is a very cool effect for applying a chrome texture map onto an object.
- Overlays are trivial to do in a custom 3D engine, but with QuickDraw 3D they require a little bit of hacking to get them to work. They will allow you to position geometry relative to the camera such that it always appears at the same place on the screen regardless of the camera's orientation.
- Splines are lots of fun and this chapter discussed how to generate both b-splines and cubic splines. Building a 3D rollercoaster track is rudimentary once you know how to build a 3D spline.

Topic 4:

3D QuickTime Movies

3D ON THE WEB

3D on the Web these days is usually done by pre-rendering animating gif's or QuickTime movies, and VRML really hasn't taken off mainly because it's too difficult to create VRML data and no one can seem to get VRML viewers to install and remain stable. But if all you want to do is put a 3D model on a web site and apply some basic animation to it, then QuickTime and QuickDraw 3D is definitely the way to go. It's stable, everyone has it (especially with QuickTime 3.0 which comes with QuickDraw 3D by default), and creating the data is very simple. Unlike VRML, however, 3D models in a QuickTime movie are not interactive... yet. The QuickTime movie simply plays back a pre-recorded sequence. Nonetheless, this is still better than pre-rendering a 3D animation because there is no image quality loss when you enlarge the playback window, plus the files are much smaller and faster to download from a web site.

I'm not going to turn this chapter into a QuickTime tutorial, but I will cover the basic information you need to know in order to create QuickTime movies which contain animating 3D models. QuickTime movies let you create many different types of media tracks, and one of those track types is a 3D Track. Another type of track is the "tween" track. A tween track is essentially the animation track, and it let's you interpolate the orientation of a 3D model over a period of time.

Unfortunately, creating a 3D movie is a fairly complex process with a lot of steps. I'll try to explain what each step is doing, but if you don't understand everything, don't worry about it. As long as you

can use the code, it isn't completely critical that you understand why it works.

STEP 1: WRITING A 3DMF FILE

The 3D data contained in a QuickTime movie is actually a 3DMF file. Your movie can either reference an actual 3DMF file to use (an alias to the file) or you can embed the 3DMF file binary data in the QuickTime movie. For our purposes, we will want to embed the data in the movie so that it is all self-contained.

First we are going to write a function called `Make3DMovie`. This is the function to which you pass the QuickDraw 3D geometry you want in the QuickTime movie:

```
/***************** MAKE 3D MOVIE ******************/
//
// INPUT: myModel =   reference to 3D data we want to put
//                    in the QT movie.
//        myView  =   reference to a View object to use
//                    for various things.
//

void Make3DMovie(TQ3Object myModel, TQ3ViewObject myView)
{
FSSpec                spec;
short                 tempFolderVRefNum;
long                  tempFolderDirID;
TQ3ViewHintsObject    viewHintsObj;

  /*************************/
  /* INITIALIZE SOME STUFF */
  /*************************/

     /* CREATE A VIEW HINTS OBJECT BASED ON THIS VIEW */

viewHintsObj = Q3ViewHints_New(myView);


       /****************/
       /* SAVE TO 3DMF */
       /****************/

  /* WE'RE GOING TO PUT TEMPORARY 3DMF FILE INTO TEMP FOLDER */

  FindFolder(kOnSystemDisk,kTemporaryFolderType,kCreateFolder,
         &tempFolderVRefNum,&tempFolderDirID);
```

```
      /* CREATE FSSPEC FOR TEMP 3DMF FILE */

  FSMakeFSSpec(tempFolderVRefNum, tempFolderDirID, "\ptemp.3dmf",
              &spec);


        /* SAVE THE 3DMF FILE */

  SaveMy3DMFModel(myModel, viewHintsObj, myView, &spec);


      /*************************/
      /* BUILD QUICKTIME MOVIE */
      /*************************/

  RecordMyMovie(&spec);


      /************/
      /* CLEANUP */
      /************/

  Q3Object_Dispose(viewHintsObj);         // nuke the view hints object
  FSpDelete(&spec);                       // nuke the temp 3dmf file
}
```

The first thing this code does is create a View Hints object. This is critical since this object will tell QuickTime where to put the camera, what background color to use, etc. Without a view hints object, you will probably get a big, blank QuickTime movie.

Next, we go ahead and write the 3DMF file to a temporary file in the Temporary Files folder in the System folder. This 3DMF file is only used while creating the QuickTime movie, thus, we delete it at the end when we are done.

In case you are not familiar with the process of saving a 3DMF file, here is the code for SaveMy3DMFModel:

```
/*************** SAVE MY 3DMF MODEL **************/
//
// INPUT: theModel      = ref to model to save to file
//        viewHintsObj  = ref to view hints object to put in file
//        myView        = ref to View object
//        myFSSpec      =  fsspec of file to save to.
//

void Save3DMFModel(TQ3Object theModel, TQ3ViewHintsObject viewHintsObj,
                TQ3ViewObject myView, FSSPec myFSSpec)
```

```
{
TQ3FileObject    fileObj;
TQ3Object        theModel;
TQ3Status        myStatus;
TQ3ViewStatus    myViewStatus;
OSErr            iErr;
TQ3StorageObject myStorageObj;

    /*****************************/
    /* CREATE THE NEW 3DMF FILE */
    /*****************************/

  FSpDelete(&myFSSpec);                    // delete any old one
  iErr = FSpCreate(&myFSSpec, 'ttxt', '3DMF', smSystemScript);
  if (iErr)
    DoError(“\pFSpCreate failed!”);


      /************************/
      /* CREATE A FILE OBJECT */
      /************************/

  /* CREATE NEW STORAGE OBJECT WHICH IS THE 3DMF FILE */

  myStorageObj = Q3FSSpecStorage_New(myFSSpec);
  if (myStorageObj == nil)
    DoError(“\pQ3FSSpecStorage failed!”);


      /* CREATE NEW FILE OBJECT */

  fileObj = Q3File_New();
  if (fileObj == nil)
    DoError(“\pQ3File_New failed!”);


      /* SET THE STORAGE FOR THE FILE OBJECT */

  if (Q3File_SetStorage(fileObj, myStorageObj) == kQ3Failure)
    DoError(“\pQ3File_SetStorage failed!”);

  Q3Object_Dispose(myStorageObj);


  /*****************************/
  /* OPEN 3DMF FILE FOR WRITING */
  /*****************************/

  myStatus = Q3File_OpenWrite(fileObj, kQ3FileModeNormal);
  if (myStatus != kQ3Success)
    DoError(“\pQ3File_OpenWrite failed!”);


    /***********************/
    /* WRITE THE 3DMF FILE */
    /***********************/

      /* START WRITING PROCESS */
```

```
  myStatus = Q3View_StartWriting(myView,fileObj);
  if ( myStatus != kQ3Success )
     DoError(“\pQ3View_StartWriting failed!”);

          /* SUBMIT LOOP */
     do
     {
        Q3Object_Submit(viewHintsObj, myView);
        Q3Geometry_Submit(theModel, myView);
        myViewStatus = Q3View_EndWriting(myView);
     } while (myViewStatus == kQ3ViewStatusRetraverse);


     /***********/
     /* CLEANUP */
     /***********/

  Q3File_Close(fileObj);
  Q3Object_Dispose(fileObj);
}
```

STEP 2: CREATING THE MOVIE FILE

After saving the 3DMF file to the Temporary Items folder, we call `RecordMyMovie` to actually create a QuickTime movie from the 3D data:

```
/********************* RECORD MY MOVIE ***************************/
//
// INPUT: fsSpec =  3DMF file to include in movie.
//

void RecordMyMovie(FSSpec *fsSpec)
{
Movie       theMovie,movie2;
short       resRefNum ;
short       resId = 0;
OSErr       iErr;
StandardFileReply  reply;
FSSpec      *specPtr,tempSpec;
Handle      tempH;
short       tempFolderVRefNum;
long        tempFolderDirID;

          /*******************************/
          /* LET USER SET FILE SAVE INFO */
          /*******************************/

  StandardPutFile("\pSave Movie As...",”\pMy3DMovie.mov”,&reply);
  if (!reply.sfGood)
     return;
  specPtr = &reply.sfFile;
```

```
     /*********************************/
     /* CREATE NEW SCRATCH MOVIE FILE */
     /*********************************/

   /* ONCE AGAIN, WE'LL SAVE THIS TO THE TEMP FOLDER */

FindFolder(kOnSystemDisk,kTemporaryFolderType,kCreateFolder,
            &tempFolderVRefNum,&tempFolderDirID);

FSMakeFSSpec(tempFolderVRefNum, tempFolderDirID, "\ptemp.mov",
            &tempSpec);


       /* CREATE THE SCRATCH MOVIE */

iErr = CreateMovieFile (&tempSpec,
                        'TVOD',
                        smCurrentScript,
                        createMovieFileDeleteCurFile,
                        &resRefNum,
                        &theMovie);
if (iErr)
   DoError("\pCreateMovieFile failed!");


       /**********************/
       /* SET MOVIE TO LOOP */
       /**********************/

tempH = NewHandleClear(sizeof(long));
AddUserData(GetMovieUserData(theMovie), tempH, 'LOOP');
DisposeHandle(tempH);


     /*********************************/
     /* ADD THE 3D DATA TO THE MOVIE */
     /*********************************/

CreateMyMovie3DTrack(theMovie, fsSpec);


     /*****************************************/
     /* NOW TURN SCRATCH MOVIE INTO REAL MOVIE */
     /*****************************************/

movie2 = FlattenMovieData(theMovie, flattenAddMovieToDataFork,
                          specPtr, 'TVOD', smCurrentScript,
                          createMovieFileDeleteCurFile);
if (movie2 == nil)
   DoError("\pFlattenMovieData failed!");


     /*************************/
     /* CLOSE THE SCRATCH FILE */
     /*************************/
```

```
  iErr = CloseMovieFile(resRefNum);
  if (iErr)
     DoFatalAlert("\pRecordNewMovie: CloseMovieFile failed!");


       /***********/
       /* CLEANUP */
       /***********/

  DisposeMovie(movie2);
  DisposeMovie(theMovie);
  DeleteMovieFile(&tempSpec);
}
```

This function outlines the basic process of creating the QuickTime movie. We need to build our movie, so we create a scratch file in the Temporary Items folder to work with. The function `CreateMyMovie3DTrack` will add the 3D data to the movie and is listed below. Once our scratch movie is created, we call `FlattenMovieData` which essentially merges the QuickTime data with the 3DMF data and allows us to create a new movie which has the 3D data embedded in it.

Once we've flattened the movie, the final movie file has been created. All we do from here is close a few files, dispose of the movie data, and delete the scratch movie file.

STEP 3: THE 3D & TWEEN TRACKS

So, the easy part is over. Let the fun begin! Creating the 3DMF file and the movie file was easy, but adding the 3D and animation data to the movie is a very messy process. It all makes sense, but it's cumbersome so take it slow.

```
#define   kMovieWidth    300       // dimensions of the movie to make
#define   kMovieHeight   300

#define   kMovieTimeScale  100     // # samples per second in movie

#define   kStartScale    1.0       // scale from start to end value
#define   kEndScale      2.0

#define   kStartRotY     0.0       // rot on y-axis from start to end
#define   kEndRotY       (kQ3Pi * 2)

#define   kStartX     0.0          // move from start to end coords
```

```
#define   kStartY     0.0
#define   kStartZ     -300.0
#define   kEndX       0.0
#define   kEndY       0.0
#define   kEndZ       300.0


#define   kDataCompressorType zlibDataCompressorSubType

enum                              // enums to help me out
{
  TRANS_ID_SCALE = 1,
  TRANS_ID_ROTY,
  TRANS_ID_MOVE
};

/************** CREATE MY MOVIE 3D TRACK ***************/
//
// Creates the movie video track and adds our animation to it.
//

void CreateMyMovie3DTrack(Movie theMovie, FSSpec *the3DMFFile)
{
Track       theTrack,tweenTrack;
Media       theMedia,tweenMedia;
OSErr       iErr;
ThreeDeeDescriptionHandle sampleDescription = nil;
long        eof;
short       fref;
TQ3Vector3D tweenTranslate;
long        referenceIndex1, referenceIndex2, referenceIndex3;
QTAtomContainer inputMap;
QTAtomContainer tweenSample;
TQ3Vector3D tweenScale;
TQ3RotateTransformData tweenRotate;
Handle      the3DMFData,compressed3DMFData = nil,dataToUse;

       /*********************************/
       /* READ IN THE 3DMF BINARY DATA */
       /*********************************/

  FSpOpenDF(the3DMFFile, fsRdPerm, &fref);      // open 3DMF file
  GetEOF(fref, &eof);                           // get size of file
  the3DMFData = NewHandleClear(eof);            // get mem for it
  HLock(the3DMFData);
  FSRead(fref, &eof, *the3DMFData);             // read the data
  FSClose(fref);                                // close the file


     /*********************************************/
     /* MAKE A SAMPLE DESCRIPTION FOR THE 3D MEDIA */
     /*********************************************/

  sampleDescription = (ThreeDeeDescriptionHandle)NewHandleClear(
                        sizeof(ThreeDeeDescription));
  (**sampleDescription).descSize = sizeof(ThreeDeeDescription);

         /***********************/
```

```
        /* CREATE THE 3D TRACK */
        /***********************/

theTrack = NewMovieTrack (theMovie,
            FixRatio(kMovieWidth,1),
            FixRatio(kMovieHeight,1),
            kNoVolume);

if (theTrack == nil)
   DoError("\pNewMovieTrack failed!");


        /*************************/
        /* CREATE THE TRACK MEDIA */
        /*************************/

theMedia = NewTrackMedia (theTrack,
                        ThreeDeeMediaType,
                        kMovieTimeScale,
                        0,nil);
if (theMedia == nil)
   DoError("\pNewTrackMedia failed!");


          /************************/
          /* COMPRESS THE 3D DATA */
          /************************/

iErr = CompressMyHandle(the3DMFData, &compressed3DMFData);
if (iErr)
   dataToUse = the3DMFData;
else
{
   (**sampleDescription).decompressorType = kDataCompressorType;
   dataToUse = compressed3DMFData;
}

          /*****************************/
          /* ADD THE 3DMF DATA AS MEDIA */
          /*****************************/

iErr = BeginMediaEdits(theMedia);
if (iErr)
   DoError("\pBeginMediaEdits failed!");

iErr = AddMediaSample(theMedia,
                          dataToUse,
                          0,
                          GetHandleSize(dataToUse),
                          kMovieTimeScale*5,
                          (SampleDescriptionHandle)sampleDescription,
                          1,            // one sample
                          0,            // self-contained samples
                          nil);
if (iErr)
   DoError("\pAddMediaSample failed!");

EndMediaEdits(theMedia);
```

```
            /*********************************/
            /* ADD THE 3D MEDIA TO THE TRACK */
            /*********************************/

InsertMediaIntoTrack(theTrack, 0, 0, GetMediaDuration(theMedia),
                     kFix1);
DisposeHandle((Handle)sampleDescription);



            /******************/
            /* DO TWEEN TRACK */
            /******************/

            /* MAKE TWEEN TRACK */

tweenTrack = NewMovieTrack(theMovie, 0, 0, 0);


            /* NEW TWEEN MEDIA */

tweenMedia = NewTrackMedia(tweenTrack, TweenMediaType,
                           gMovieTimeScale, nil, 0);


          /* MAKE EMPTY TWEEN SAMPLE */

iErr = QTNewAtomContainer(&tweenSample);
if (iErr)
   DoError("\pQTNewAtomContainer failed!");


        /*************************/
        /* ADD TWEEN ENTRY: SCALE */
        /*************************/

            /* SET ENDING SCALE VALUE */

tweenScale.x =
tweenScale.y =
tweenScale.z = kEndScale;
AddMyTweenEntryToSample(tweenSample, TRANS_ID_SCALE,
                     kTweenType3dScale, &tweenScale,
                     sizeof(tweenScale));

            /* SET INITIAL SCALE VALUE */

tweenScale.x =
tweenScale.y =
tweenScale.z = kStartScale;
SetMyTweenEntryInitialConditions(tweenSample, TRANS_ID_SCALE,
                                 &tweenScale, sizeof(tweenScale));

          /*****************************/
          /* ADD TWEEN ENTRY: ROTATE-Y */
          /*****************************/
```

```
       /* SET ENDING ROTATE VALUE */

tweenRotate.axis = kQ3AxisY;
tweenRotate.radians = kEndRotY;
addTweenEntryToSample(tweenSample, TRANS_ID_ROTY, kTweenType3dRotate,
                    &tweenRotate, sizeof(tweenRotate));

       /* SET STARTING ROTATE VALUE */

tweenRotate.axis = kQ3AxisY;
tweenRotate.radians = kStartRotY;
setTweenEntryInitialConditions(tweenSample, TRANS_ID_ROTY,
                           &tweenRotate, sizeof(tweenRotate));


     /*******************************/
     /* ADD TWEEN ENTRY: TRANSLATE */
     /*******************************/


          /* SET ENDING COORDINATE */

tweenTranslate.x = kEndX;
tweenTranslate.y = kEndY;
tweenTranslate.z = kEndZ;
addTweenEntryToSample(tweenSample, TRANS_ID_MOVE,
                    kTweenType3dTranslate, &tweenTranslate,
                    sizeof(tweenTranslate));


       /* SET STARTING COORDINATE */

tweenTranslate.x = kStartX;
tweenTranslate.y = kStartY;
tweenTranslate.z = kStartZ;
setTweenEntryInitialConditions(tweenSample, TRANS_ID_MOVE,
                          &tweenTranslate, sizeof(tweenTranslate));


     /*********************************/
     /* MAKE TWEEN SAMPLE DESCRIPTION */
     /*********************************/

sampleDescription = (ThreeDeeDescriptionHandle)NewHandleClear(
                   sizeof(SampleDescription));
(**sampleDescription).descSize = sizeof(SampleDescription);


     /*************************************/
     /* ADD TWEEN SAMPLE TO THE TWEEN MEDIA */
     /*************************************/

BeginMediaEdits(tweenMedia);

iErr = AddMediaSample(tweenMedia, tweenSample, 0,
                   GetHandleSize(tweenSample), kMovieTimeScale*5,
                  (SampleDescriptionHandle)sampleDescription, 1,
```

```
                    0, nil);
if (iErr)
   DoError("\pAddMediaSample failed!");

EndMediaEdits(tweenMedia);


          /**********************************/
          /* ADD TWEEN MEDIA INTO THE TRACK */
          /**********************************/

InsertMediaIntoTrack(tweenTrack, 0, 0, GetMediaDuration(tweenMedia),
                     kFix1);

QTDisposeAtomContainer(tweenSample);    // throw away a few things
DisposeHandle((Handle)sampleDescription);


        /************************************************/
        /* CREATE REFERENCES BETWEEN 3D AND TWEEN TRACKS */
        /************************************************/


AddTrackReference(theTrack, tweenTrack, kTrackModifierReference,
                  &referenceIndex1);
AddTrackReference(theTrack, tweenTrack, kTrackModifierReference,
                  &referenceIndex2);
AddTrackReference(theTrack, tweenTrack, kTrackModifierReference,
                  &referenceIndex3);


   /***********************************/
   /* CREATE INPUT MAP FOR THE 3D TRACK */
   /***********************************/

QTNewAtomContainer(&inputMap);


     /* ADD ENTRIES TO INPUT MAP */
     //
     // NOTE:  This order seems to determine how transforms
     // are applied.  Do transform *then* rotate to get
     //  a rot->trans concatenation.  In other words,
     // do in reverse order that you want transforms
     // applied.
     //

AddMyTweenEntryToInputMapEntry(inputMap, referenceIndex1,
                     TRANS_ID_MOVE, kTrackModifierType3d4x4Matrix,
                     nil);
AddMyTweenEntryToInputMapEntry (inputMap, referenceIndex2,
                     TRANS_ID_ROTY, kTrackModifierType3d4x4Matrix,
                     nil);
AddMyTweenEntryToInputMapEntry (inputMap, referenceIndex3,
                     TRANS_ID_SCALE, kTrackModifierType3d4x4Matrix,
                     nil);
```

```
      /* ATTACH INPUT MAP TO 3D MEDIA HANDLER */

  SetMediaInputMap(theMedia, inputMap);


             /***********/
             /* CLEANUP */
             /***********/

  QTDisposeAtomContainer(inputMap);
  DisposeHandle(the3DMFData);
  if (compressed3DMFData != nil)
     DisposeHandle(compressed3DMFData);
}
```

Okay, the above function is a monster, I know. The general flow of events goes like this:

1. Load the 3DMF file's binary data into memory.
2. Create the 3D track & 3D media
3. Compress the 3D data.
4. Assign the 3D data to the media.
5. Assign the media to the track.
6. Create the Tween track & media.
7. Create a tween "sample".
8. Add scale, rotate, and translate information to the sample.
9. Add the sample to the tween media.
10. Add the tween media to the tween track.
11. Create references between the 3D and Tween tracks.
12. Cleanup

When you look at it broken down like this, it starts to make a little more sense. You can get more information on creating tween tracks

by downloading the “Tween Media Handler” documentation off of the QuickTime web site at www.quicktime.apple.com.

The code above added a scale, y-axis rotate, and a translate tween to the movie, but we are not limited to this. We could have done just a rotate, or just a translate tween. We could have also added x and z-axis rotates as well. The header file Movies.h contains enums for all of the possible tween types. There are quite a few of them and if you really get into it, you can make your 3D QuickTime movies do some really cool stuff!

ADDING TWEEN ENTRIES

There are still three functions to write which are needed by the CreateMyMovie3DTrack function above. These functions are responsible for actually adding tween data to samples.

```
/************** ADD MY TWEEN ENTRY TO SAMPLE ****************/
//
// Adds a tween to the sample.
//

OSErr AddMyTweenEntryToSample(QTAtomContainer tweenSample, long tweenID,
                              long tweenType, void *tweenData,
                              long tweenDataSize)
{
OSErr err;
QTAtom tweenAtom;

       /* CREATE ENTRY FOR THIS TWEEN IN THE SAMPLE */

  err = QTInsertChild(tweenSample, kParentAtomIsContainer, kTweenEntry,
                      tweenID, 0, 0, nil, &tweenAtom);
  if (err)
     return(err);


       /* DEFINE THE TYPE OF IT */

  err = QTInsertChild(tweenSample, tweenAtom, kTweenType, 1, 0,
                      sizeof(tweenType), &tweenType, nil);
  if (err)
     return(err);


       /* DEFINE THE DATA FOR IT */
```

```
   err = QTInsertChild(tweenSample, tweenAtom, kTweenData, 1, 0,
                       tweenDataSize, tweenData, nil);

   return(err);
}
```

```
/********* SET TWEEN ENTRY INITIAL CONDITIONS ***********/
//
// Sets the 1st tween value in the sample.
//

OSErr SetMyTweenEntryInitialConditions(QTAtomContainer tweenSample,
                                       long tweenID, void *initialData,
                                       long initialDataSize)
{
QTAtom tweenAtom;

            /* LOOK-UP THE TWEEN ENTRY */

   tweenAtom = QTFindChildByID(tweenSample, kParentAtomIsContainer,
                               kTweenEntry, tweenID, nil);
   if (!tweenAtom)
      return(paramErr);


            /* ADD THE INITIAL DATA OFFSET */

   return(QTInsertChild(tweenSample, tweenAtom, 'icnd', 1, 0,
                       initialDataSize, initialData, nil));
}
```

```
/*********** ADD TWEEN ENTRY TO INPUT MAP ENTRY ************/

OSErr AddMyTweenEntryToInputMapEntry(QTAtomContainer inputMap,
                                     long referenceIndex,
                                     long tweenID, long tweenType,
                                     char *name)
{
OSErr err;
QTAtom inputAtom;

          /* ADD AN INPUT ENTRY TO THE INPUT MAP */

   err = QTInsertChild(inputMap, kParentAtomIsContainer,
                       kTrackModifierInput, referenceIndex, 0, 0,
                       nil, &inputAtom);
   if (err)
      return(err);


          /* SET THE TYPE OF THE MODIFIER INPUT */
```

```
          //
          // note: for 3D, this is almost always
          // kTrackModifierType3d4x4Matrix
          //

  err = QTInsertChild(inputMap, inputAtom, kTrackModifierType, 1, 0,
                      sizeof(tweenType), &tweenType, nil);
  if (err)
     return(err);

          /* SET THE SUB INPUT ID (ID OF THE TWEEN ENTRY) */

  err = QTInsertChild(inputMap, inputAtom, kInputMapSubInputID, 1, 0,
                      sizeof(tweenID), &tweenID, nil);
  if (err)
     return(err);


            /* DEFINE THE NAME */

  if (name)
  {
     long nameLen = 1;
     Ptr p = name;

     while (*p++)
        nameLen++;

     err = QTInsertChild(inputMap, inputAtom, kTrackModifierInputName,
                         1, 0,  nameLen, name, nil);
     if (err)
        return(err);
  }
  return(noErr);
}
```

The complexity of this code may seem a bit overwhelming at first, but once you start working with it you will begin to understand how it works. Soon you'll be able to modify it to do all sorts of complex animations by adding multiple tweens in the tween track. You can make a model fly in, turn around and fly away. Or you can make a model spin like mad while rapidly scaling in and out.

COMPRESSING THE 3D DATA

In the function `CreateMyMovie3DTrack` we called another function named CompressMyHandle which took our 3DMF binary data and compressed it. This feature is only available in QuickTime 3.0 or newer, therefore, do not compress 3D data for QuickTime movies which need to playback correctly with older versions of QuickTime.

QuickTime 3.0 has several compressors and decompressors built into it, including data compressors for compressing any arbitrary block of data. Technically we're using the Component Manager to do this, but the functionality is by way of QuickTime 3.0.

Here is the code which does the compression:

```
/*************** COMPRESS MY HANDLE *****************/
//
// INPUT: srcHandle = handle to source data to compress
// OUTPUT: compressedSrcData = handle to compressed data
//

OSErr CompressMyHandle(Handle srcHandle, Handle *compressedSrcData)
{
OSErr            err = noErr;
ComponentInstance  dataCompressor = nil;
unsigned long    srcSize = GetHandleSize(srcHandle);
unsigned long    compressSize;
unsigned long    whoCares;


      /* INIT HANDLE IN CASE WE CAN'T DO COMPRESSION */

  *compressedSrcData = nil;


        /* OPEN THE COMPRESSOR COMPONENT */

  err = OpenADefaultComponent(DataCompressorComponentType,
                        kDataCompressorType, &dataCompressor);
  if (err)
     goto bail;


      /***********************************************/
      /* GET SIZE OF COMPRESSION BUFFER & ALLOCATE ONE */
      /***********************************************/

  err = DataCodecGetCompressBufferSize(dataCompressor, srcSize,
                                       &compressSize);
  if (err)
     goto bail;

  *compressedSrcData = NewHandle(compressSize + sizeof(long));
  if (err = MemError())
     goto bail;

  HLockHi(srcHandle);
  HLockHi(*compressedSrcData);
```

```
            /**********************/
            /* DO THE COMPRESSION */
            /**********************/

  err = DataCodecCompress(dataCompressor, *srcHandle, srcSize,
                          **compressedSrcData + sizeof(long),
                          compressSize - sizeof(long), &compressSize,
                          &whoCares);
  if (err)
     goto bail;


          /* PUT SIZE INTO 1ST LONG OF DATA */

  ***(long ***)compressedSrcData = EndianS32_NtoB(srcSize);


            /***********/
            /* CLEANUP */
            /***********/

  HUnlock(*compressedSrcData);
  HUnlock(srcHandle);

       /* RESIZE HANDLE TO FIT EXACTLY RIGHT */

  SetHandleSize(*compressedSrcData, compressSize + sizeof(long));

bail:
  if (dataCompressor)
     CloseComponent(dataCompressor);

  return err;
}
```

The compressor type values are found in `QuickTimeComponents.h` and you should generally use `zlibDataCompressorSubType` with 3D data.

SUMMARY

Creating a QuickTime movie with a 3D track is a fairly complex thing to do (relative to doing other things with QuickDraw 3D), but once you get used to it, you'll find that you can create some really cool QuickTime movies.

It's safe to say that QuickDraw 3D with QuickTime is the easiest way to get 3D content onto the web.

Interactive Media Resources Include:

- **Interactive Media Guidebooks**
- **Market Research Reports**
- **Survival Guides—Technical "How To" Guides**
- **Comarketing Opportunities**
- **Special Discounts**

Apple Developer Connection

As Apple technologies such as QuickTime, ColorSync, and AppleScript continue to expand Macintosh as the tool of choice for content creators and interactive media authors, the Apple Developer Connection continues its commitment to provide creative professionals with the latest technical and marketing information and tools.

Interactive Media Resources

Whether looking for technical guides from industry experts or for market and industry research reports to help make critical business decisions, you'll find them on the Interactive Media Resources page.

Apple Developer Connection Programs and Products

ADC programs and products offer easy access to technical and business resources for anyone interested in developing for Apple platforms worldwide. Apple offers three levels of program participation serving developer needs .

Membership Programs

Online Program—Developers gain access to the latest technical documentation for Apple technologies as well as business resources and information through the Apple Developer Connection web site.

Select Program—Offers developers the convenience of technical and business information and resources on monthly CDs, provides access to prerelease software, and bundles two technical support incidents.

Premier Program—Meets the needs of developers who desire the most complete suite of products and services from Apple, including eight technical support incidents and discounts on Apple hardware.

Standalone Products

Apple offers many standalone products that allow developers to choose their own level of support from Apple or enhance their Select or Premier Program membership. Choose from the following products and begin enjoying the benefits today.

Developer Connection Mailing—Subscribe to the Apple Developer Connection Mailing for the latest in development tools, system software, and more.

Technical Support—Purchase technical support and work directly with Apple's Worldwide Developer Technical Support engineers.

Apple Developer Connection News—Stay connected to Apple and developer-specific news by subscribing to our free weekly e-mail newsletter, *Apple Developer Connection News*. Each newsletter contains up-to-date information on topics such as Mac OS, Interactive Media, Hardware, Apple News and Comarketing Opportunities.

Macintosh Products Guide

The most complete guide for Macintosh products! Be sure to list your hardware and software products in our *free* online database at http://www.macsoftware.apple.com

http://developer.apple.com/media

The ultimate source for creative professionals.