# 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...

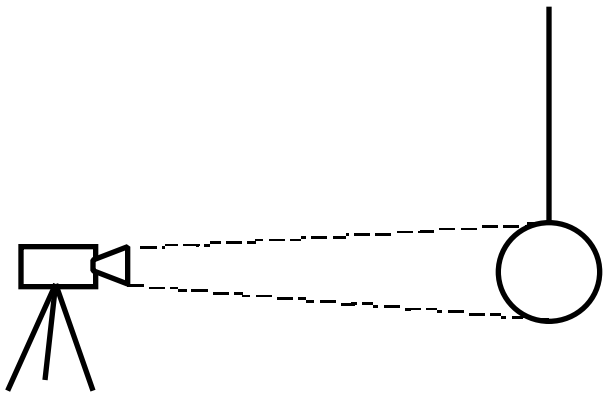
Figure 3.1

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



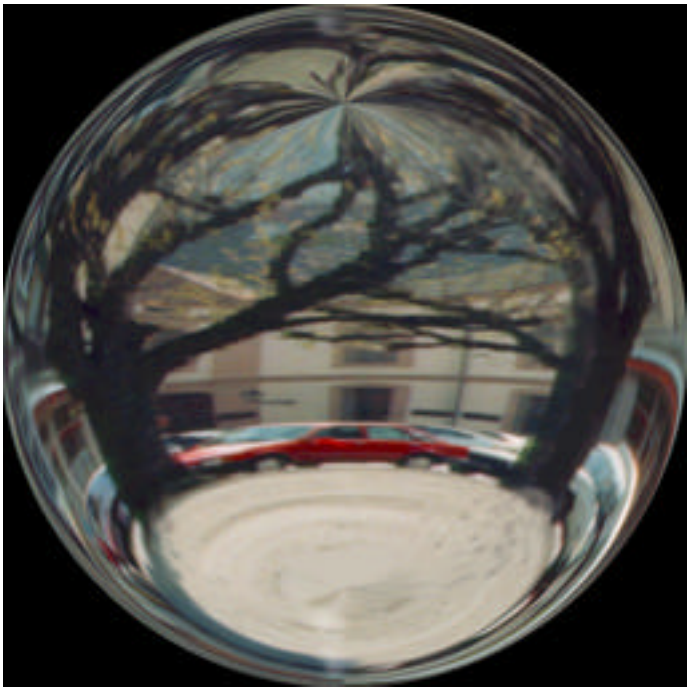
Take a close-up picture of a chrome ball hanging froma string makes perfect reflection maps.

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.

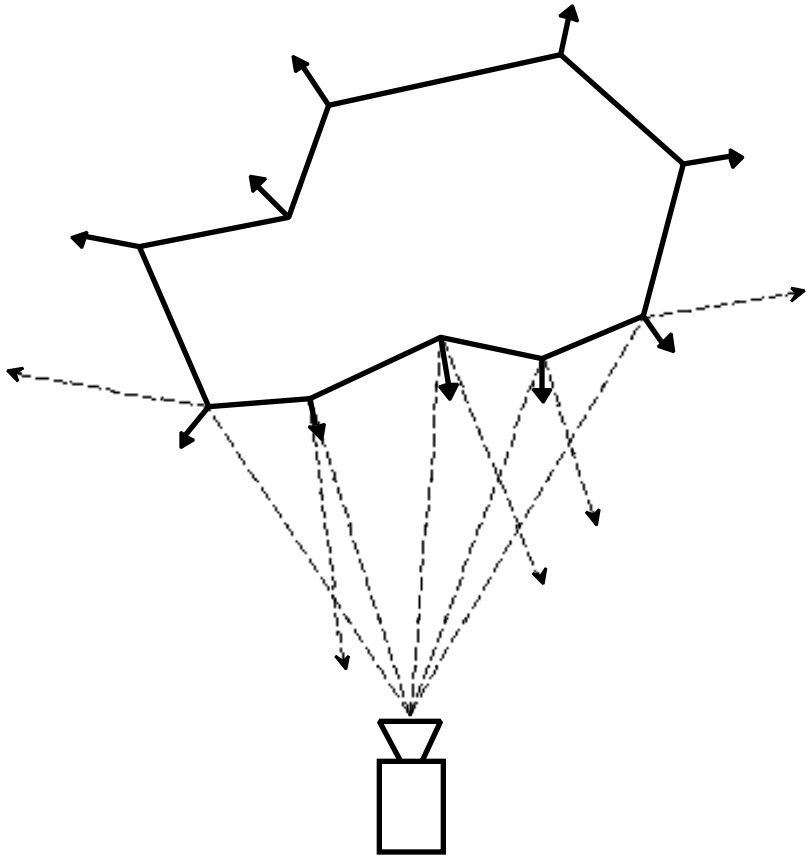
# REFLECTION CODE

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:

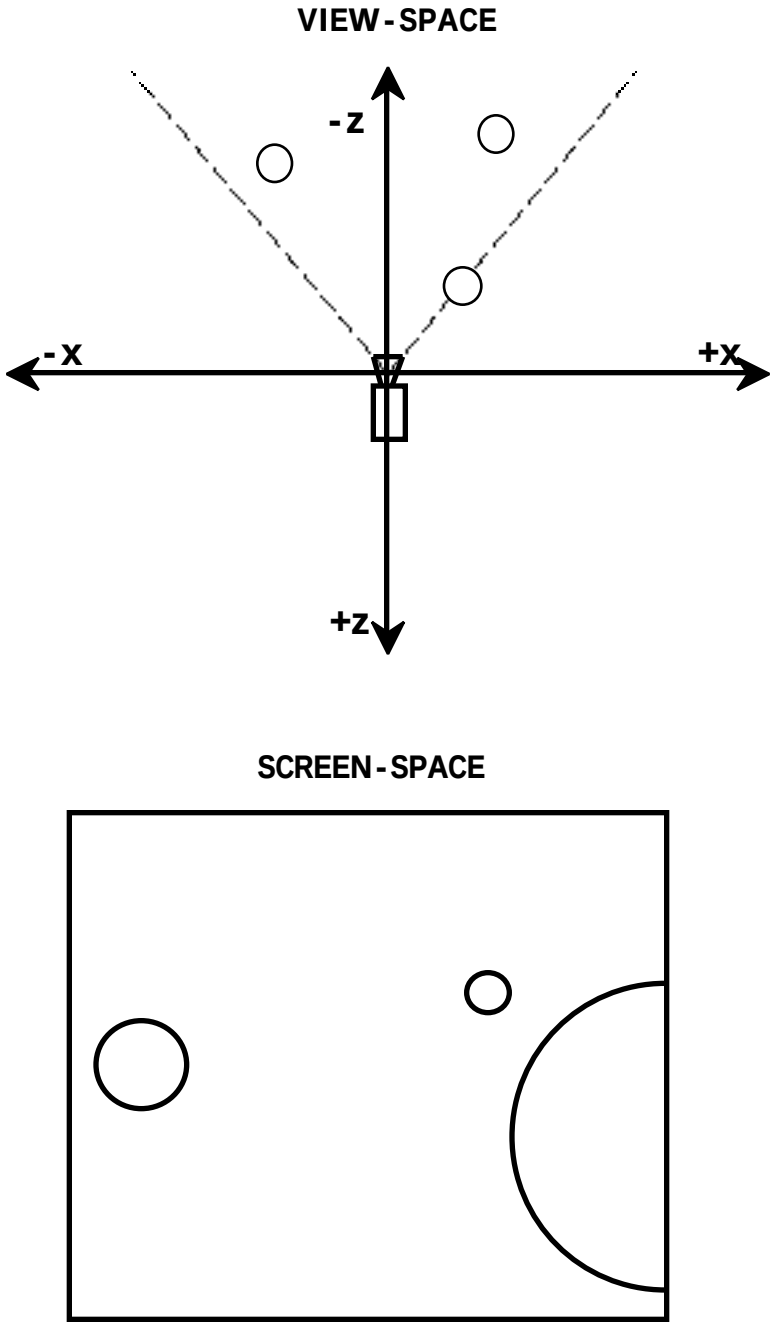
1. Calculate the inverse of the camera's `world->view` matrix which gives you a `view->world` matrix.

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

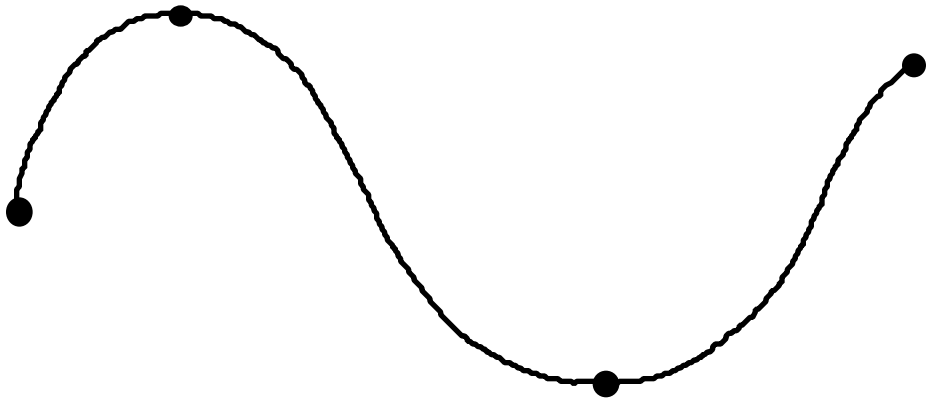
The number one technical question I've been asked by QuickDraw 3D developers is "how did you do the track in Gerbils?" The answer is splines. Splines are a wonderful thing and are very easy to generate. They're good for generating rollercoaster track as in Gerbils, but they're also good for creating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

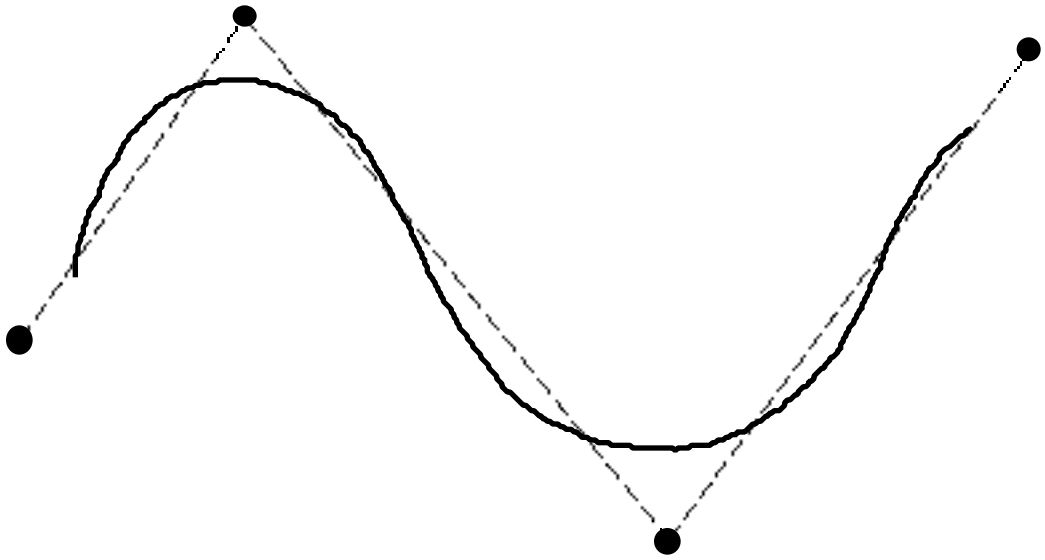
For our purposes, there are two types of splines: b-splines and cubic splines. B-splines are faster to generate than cubic splines, but b-splines do not pass through their control points whereas cubic splines do.

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
```

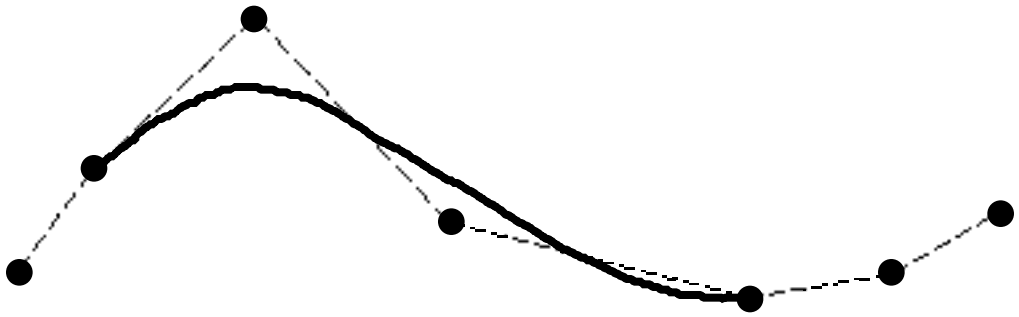
```
//                                the spline points
//

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++;
```

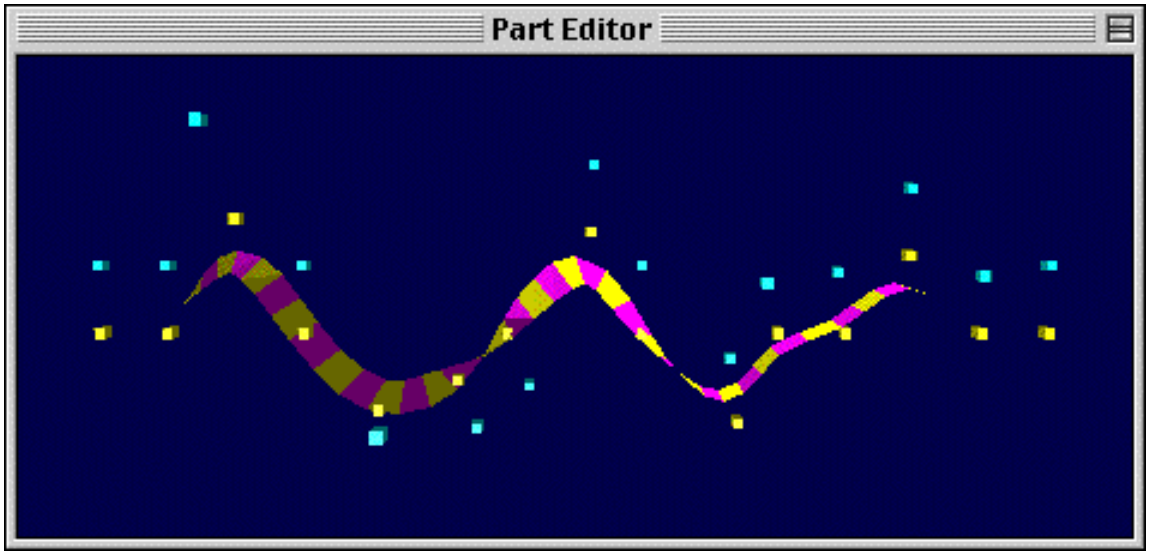
```
    /*************/
    /* ALL DONE */
    /*************/

  *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.

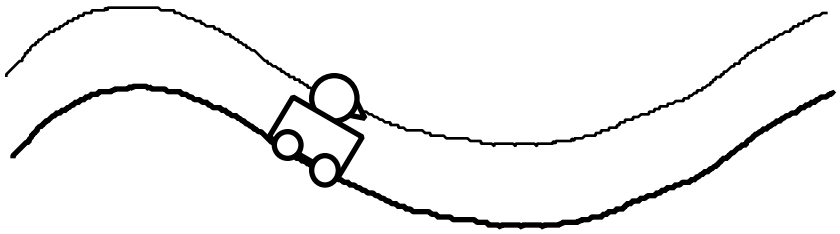
Figure 3.10




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.

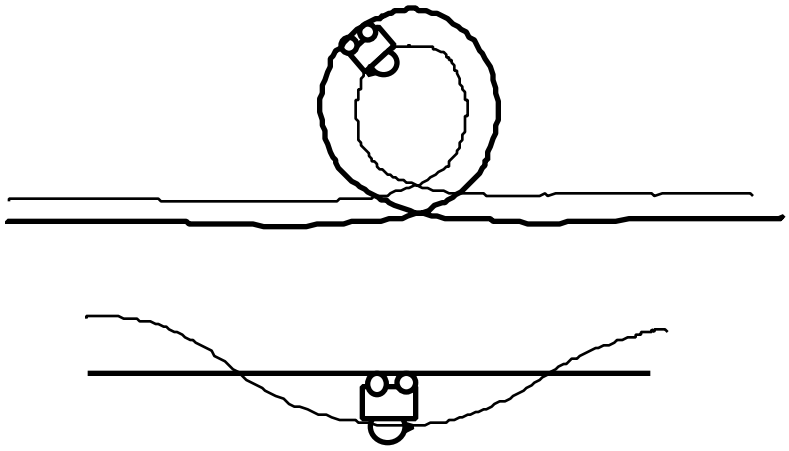
Figure 3.11



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.

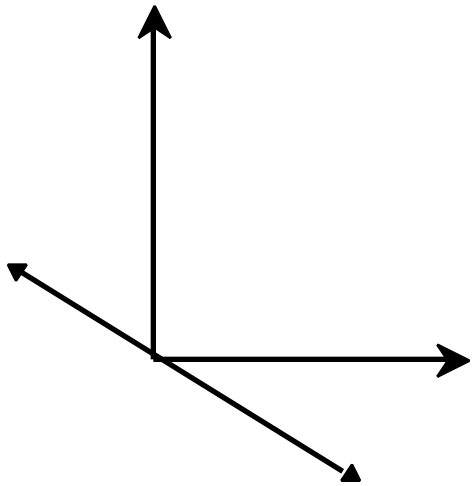
Figure 3.12



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.