# 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.

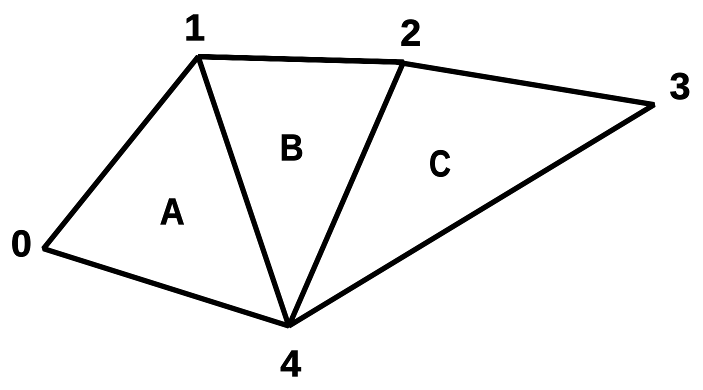
## *Triangles*

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



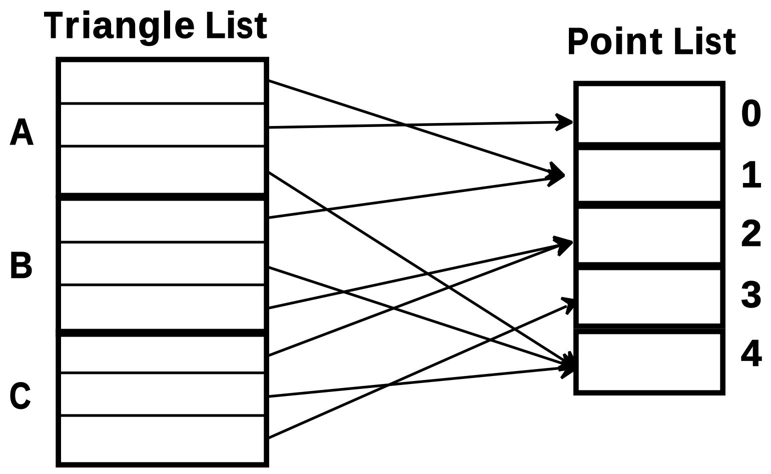
A geometry made of 3 triangles (A, B, anc C) and 5 points (0..4)

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

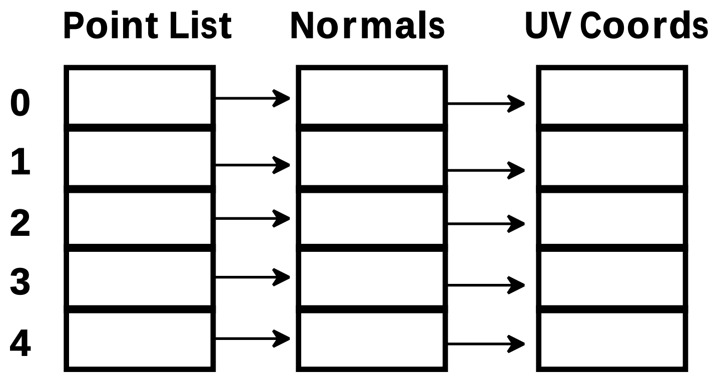


Diagram showing the parallel correlations among the point list, normal list, and UV list.

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.

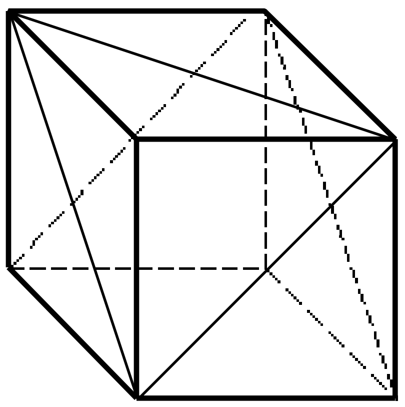
You may be wondering “How do I build my airplane model which has six different texture maps if I’m being told to only apply one material per TriMesh?” The answer is that you must build your airplane model from six different TriMeshes - one for each texture map. The reason for doing this is because QuickDraw 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

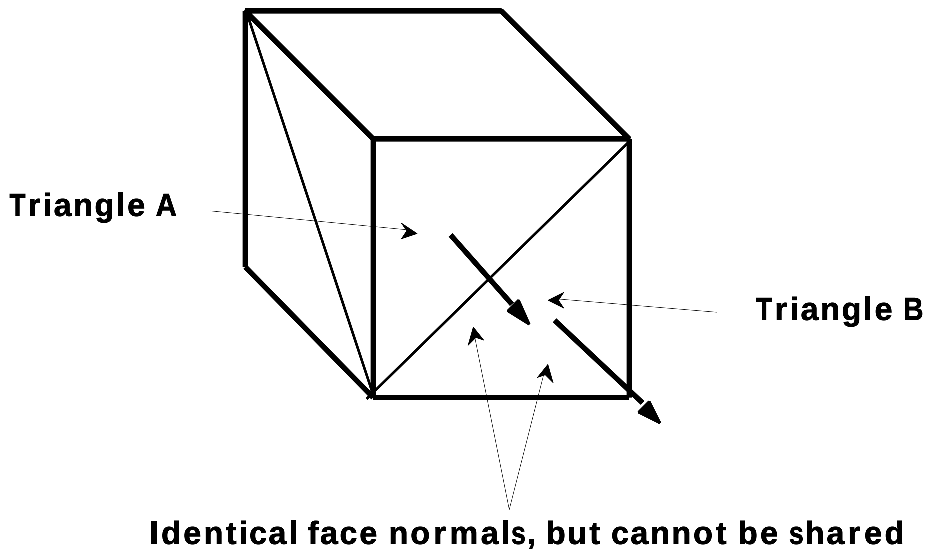
This cube is made up of 12 triangles, 8 points, and 6 normals, right? Wrong! There is no way to represent this cube in such a way using the TriMesh, and here's why:

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

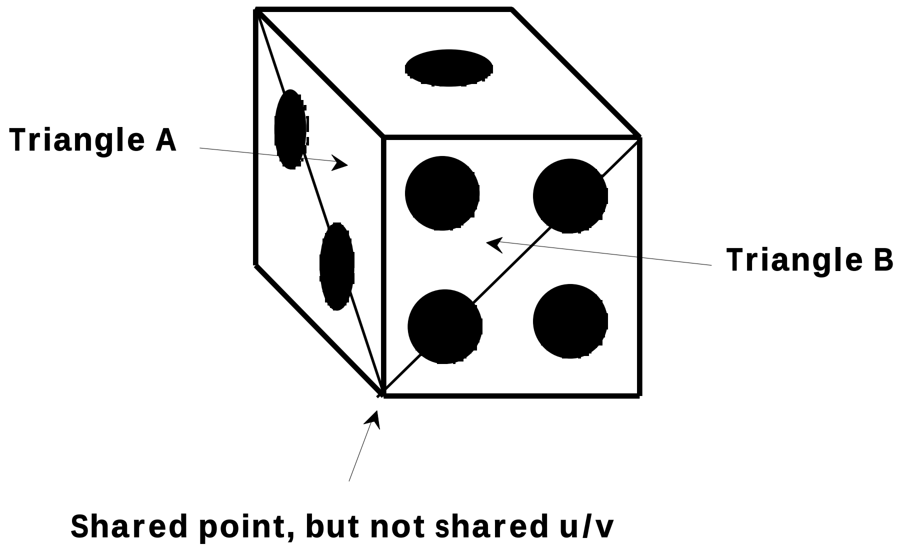


Triangle A and B have identical face normals which cannot be shared, so two copies are needed.

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?

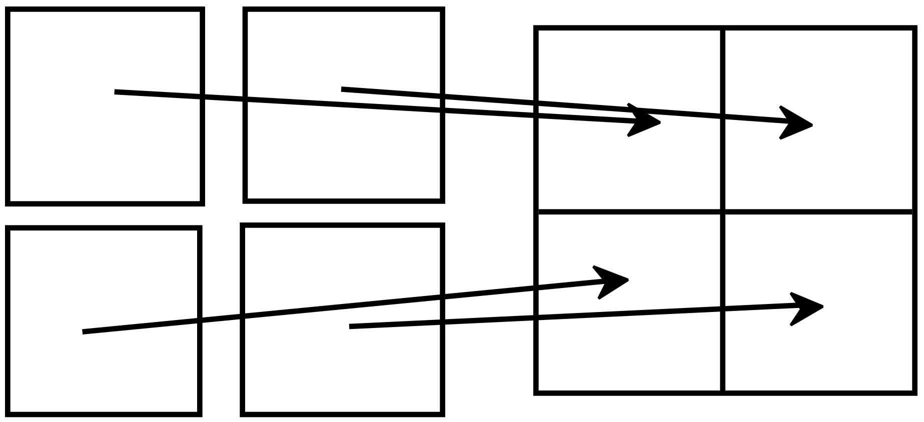
## Working Around The Restrictions

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.

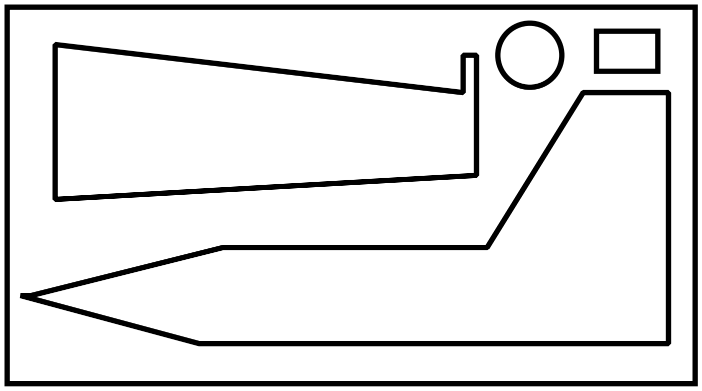
Figure 1.6



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



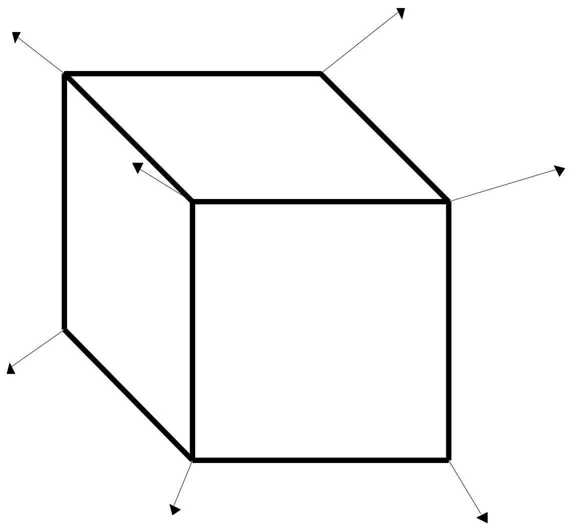
We put the wing, wheel, and windshield texture in
the “black-space” of the body texture.

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.

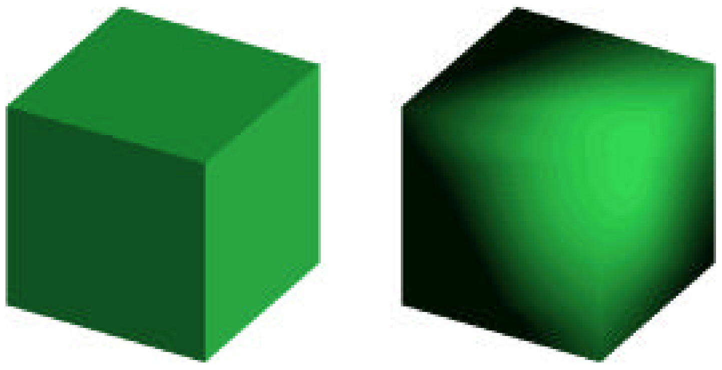
Figure 1.8



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

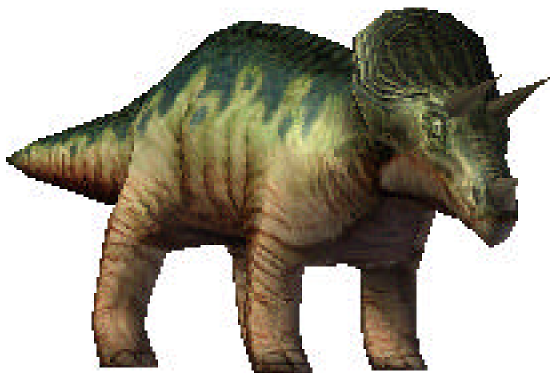


The cube on the left shows the inefficient TriMesh with lots of duplicate data.  The cube on the right doesn't have the hard edges, but it's much more efficient.

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



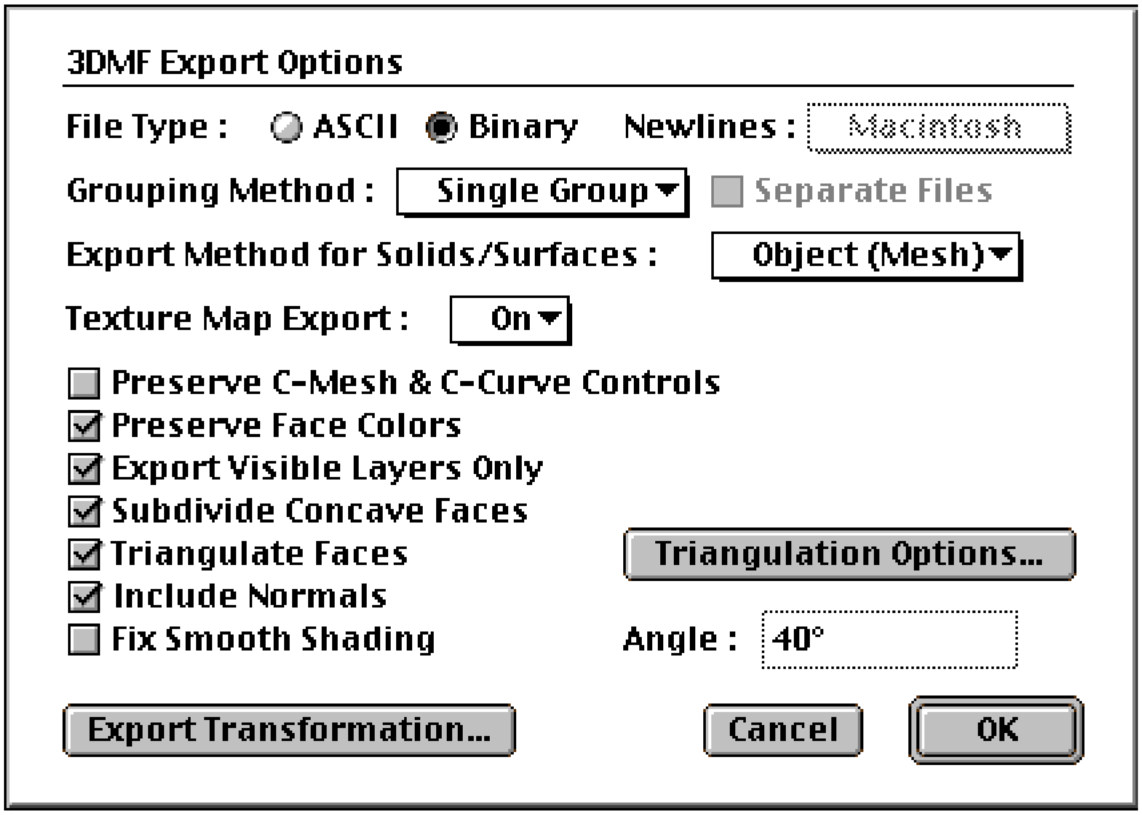
This model is entirely smooth shaded and is made from a single, highly optmized TriMesh.

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:



3DMF Export Options

File Type : ASCII [✓ Binary] Newlines : Macintosh

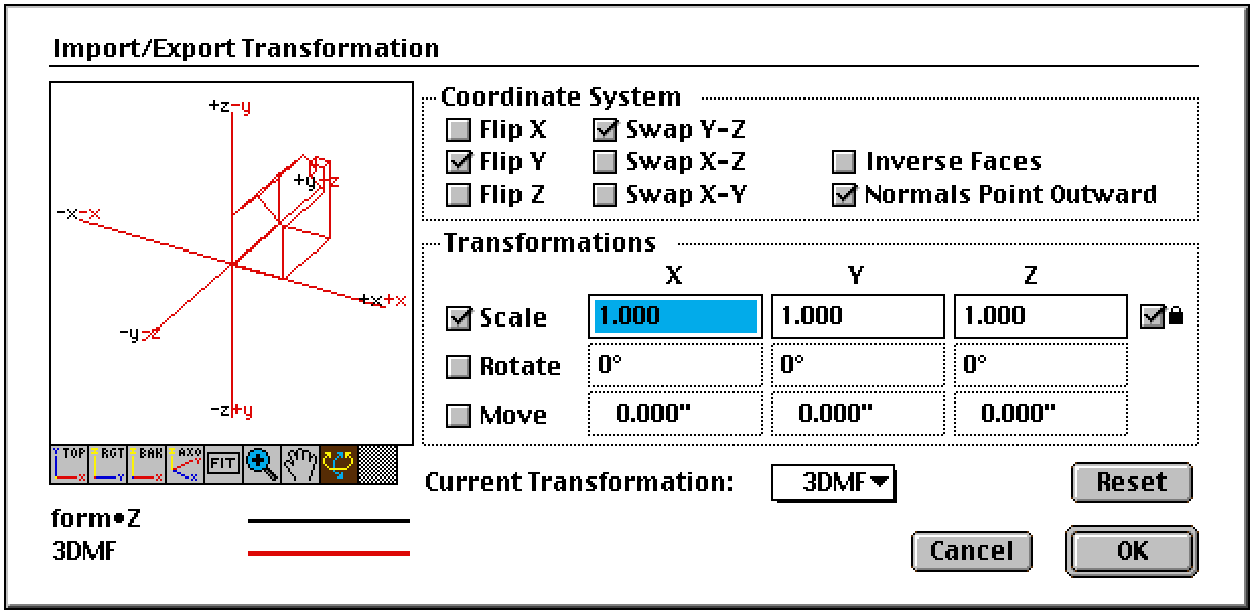
Grouping Method : Single Group ☐ Separate Files

Export Method for Solids/Surfaces : Object (Mesh)

Texture Map Export : On

☐ Preserve C-Mesh & C-Curve Controls
☑ Preserve Face Colors
☑ Export Visible Layers Only
☑ Subdivide Concave Faces
☑ Triangulate Faces
☑ Include Normals
☐ Fix Smooth Shading

Angle : 40°



Import/Export Transformation

Coordinate System

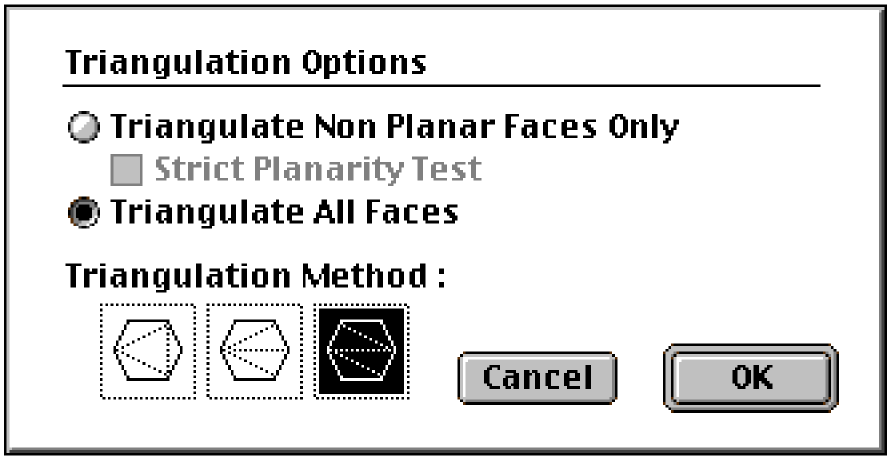
- [ ] Flip X
- [x] Flip Y
- [ ] Flip Z
- [x] Swap Y-Z
- [ ] Swap X-Z
- [ ] Swap X-Y
- [ ] Inverse Faces
- [x] Normals Point Outward

Transformations

| | X | Y | Z |
|---|---|---|---|
| ☑ Scale | 1.000 | 1.000 | 1.000 |
| ☐ Rotate | 0° | 0° | 0° |
| ☐ Move | 0.000" | 0.000" | 0.000" |

Current Transformation: 3DMF

| | |
|---|---|
| form•Z | ▬ |
| 3DMF | 🟥 |



Triangulation Options

( ) Triangulate Non Planar Faces Only

☐ Strict Planarity Test

(●) Triangulate All Faces

Triangulation Method :

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.

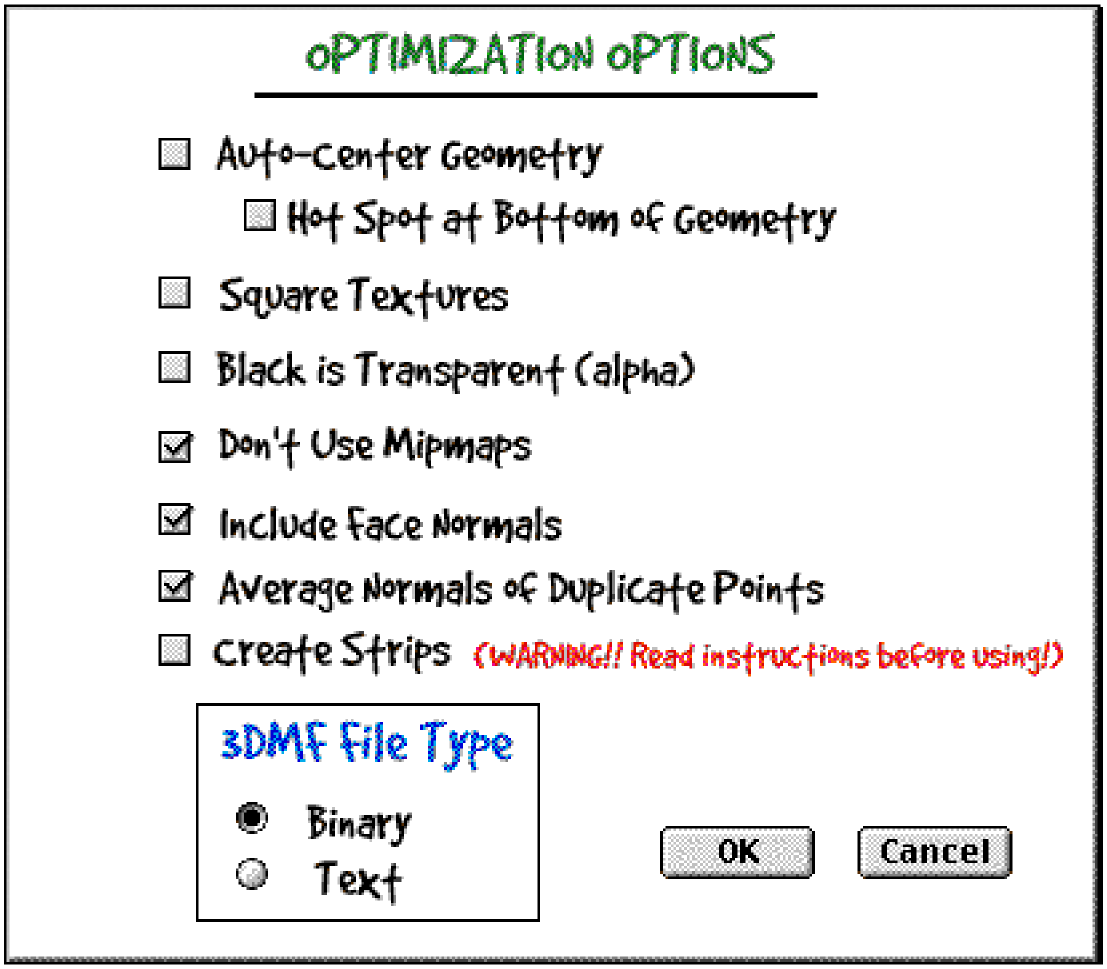
Form*Z does export transparency attributes on transparent geometry, but it does not use the transparency value in the material you have assigned to it. Rather, it uses the Transparency on/off value which you can apply to a model on an object by object basis. The transparency value applied is 50% and 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:



OPTIMIZATION OPTIONS

☐ Auto-Center Geometry

☐ Hot Spot at Bottom of Geometry

☐ Square Textures

☐ Black is Transparent (alpha)

☑ Don't Use Mipmaps

☑ Include Face Normals

☑ Average Normals of Duplicate Points

☐ Create Strips (WARNING!! Read instructions before using!)

3DMF File Type

● Binary

○ Text

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

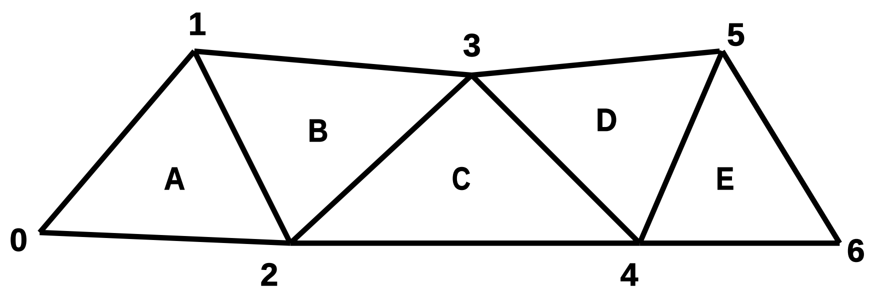
There is one more TriMesh optimization which may speed up your application: Strips and Fans. Be warned that as of this writing, this optimization actually has no effect. RAVE directly supports Strips and Fans, but QuickDraw 3D does not. QuickDraw 3D does, however, support the TriMesh (obviously), and 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”

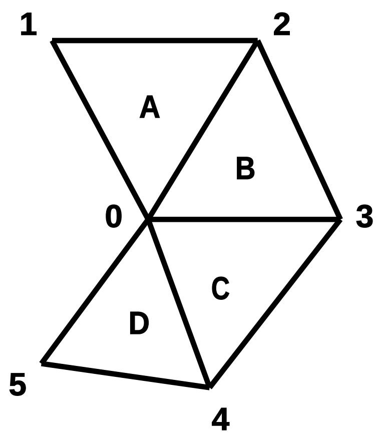
As this TriMesh is processed, we work from triangle A through triangle E. When it passes triangle A to the 3D hardware, it needs to pass vertices 0, 1, and 2.

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.