Chapter Eight

Meshes

Introduction

In Graphics Programming Module I we created our own mesh class (CMesh) to manage model geometry. While this is certainly an acceptable approach and can be useful for many tasks, the D3DX library provides a collection of mesh types that provide some key advantages over our simple vertex/index buffer wrapper class. This chapter will examine some of the core D3DX mesh types and we will learn how to work with them in our game development projects.

Some key features of D3DX provided meshes are:

Optimization: D3DX mesh objects are more than simple wrappers around a vertex and index buffer. They include important optimization features that allow for more efficient batching and rendering. These features alone make using D3DX mesh objects an attractive choice. Although D3DX mesh objects fully encapsulate batch rendering of their triangles, you are not required to use these features. If you wanted to store and render your meshes using proprietary mesh classes, you could use a D3DX mesh object to temporarily store and optimize your data for faster rendering with minimum state changes and better vertex cache coherency. Once done, you could lock the buffers and copy the optimized data back into your own mesh class and then destroy the D3DX mesh.

Asset Support: DirectX Graphics uses the X file as its native geometry file format for loading and storing 3D objects. Although we can store geometry in any file format we please, we are responsible for writing code to parse the data in that file and store it in our meshes. The X file format is flexible enough to store a wide variety of information, including complete scene hierarchies. While DirectX Graphics provides interfaces to help manually load and parse X file data structures, this is really not a task that will be relished by the uninitiated. Fortunately, we do not have to worry too much about that since the D3DX library provides functions that automate X file loading and data storage. With a single function call, we can load an X file and wind up with a 'ready to render' D3DX mesh, saving us a good deal of work. The X file format is now supported by most popular commercial 3D modeling applications, so we will be able to import high quality 3D artwork into our projects with ease. DirectX Graphics also ships (provided you also download the additional DirectX 9.0 Extras package) with command line tools that provide easy conversion of 3D Studio™ files (3ds) into the X file format. The *conv3ds.exe* is a command line tool that converts 3ds files to X files. Unfortunately these command line tools do not always work as well as one might hope. Instead of using the command line tools, DirectX Graphics now ships (provided once again that you download the Extras package) with various exporter plug-ins for popular graphics packages such as 3D Studio MAX™ and Maya™.

The Maya plug-in is called Xexport.mll. It is an mll (a Maya™ dynamic link library) that can be dragged and dropped into the plug-ins directory of the Maya™ application. This dll also ships with source code. The 3D Studio™ plug-in dll is called XskinExp.dle but using it is not quite as straightforward to integrate. First, only the source code is provided, so you will need to compile the dll yourself. Be sure that you have downloaded the 3D Studio MAX™ Software Development Kit and the Character Studio™ Software Development Kit for the source code to compile correctly (some versions of MAX may include the SDK on the CD). Unfortunately, one of the header files needed for the compile is missing from the DX9 extras package. To save confusion, we have supplied this missing file with the source code that accompanies this chapter.

> **Note**: GILES™ has the ability to import and export X files, so be sure to check out this feature if you choose to use this format in your applications. Since GILES™ imports .3ds files as well, it is worth considering as an alternative to the command line tools that ship with the DirectX SDK.

So even if you intend to use your own mesh classes for rendering, you can use a temporary D3DX mesh object to load in X file data. Again, it is little trouble to lock the mesh vertex and index buffers and copy out the data into your own mesh type. The result: fast and free X file loading for your application.

Utility: There are other useful features that become available when using D3DX meshes. For example there are intersection tests, bounding volume generation, vertex normal calculations, geometry cleaning functions to remove stray vertices, welding functions, level of detail algorithms, mesh cloning, and much more, all built right into the interfaces.

Although we are free to use the D3DX mesh solely as a utility object, often we will use all of its features: loading, optimization, and rendering. This will be the focus of our lab projects that accompany this chapter.

8.1 D3DX Mesh Types Overview

There are five mesh interfaces in D3DX, each providing a specific set of functionality. In this chapter we will discuss only four of these interfaces in detail: ID3DXBaseMesh, ID3DXMesh, ID3DXSPMesh and ID3DXPMesh. The remaining interface ID3DXPatchMesh, which manages curved surface rendering and related tasks, will be covered a bit later in this programming series.

Before getting under the hood with the four mesh types we are going to cover in this chapter, we will first briefly review the five D3DX mesh types and discuss the high level functionality they provide. After this brief overview, we will examine the mesh types in more detail and learn how to create them, optimize them, and render them.

8.1.1 ID3DXBaseMesh

ID3DXBaseMesh provides the inheritable interface for core mesh functionality. Features include vertex and index buffer management, mesh cloning, face adjacency information processing, rendering, and other housekeeping functions. Base meshes cannot be instantiated, so we will always create one of the derived mesh types -- the derived types support all of the base mesh interface methods.

> **Note:** While we will not use this interface directly, it serves a useful purpose in a game engine since we can store pointers of this type for all of our mesh objects. This allows us to query the interface to determine which of the other mesh types is actually being used. While we might use one or more derived class instances like ID3DXMesh and ID3DXPMesh in our scene, we can store pointers to each in an ID3DXBaseMesh array and cast between types as needed.

Because we will never explicitly instantiate an ID3DXBaseMesh, in this chapter we will cover its interface methods in the context of the more commonly used derived classes. We will indicate which functions belong to which interface as we progress, so that inherited functionality is not obscured by the examples. This will allow us to use code snippets that more closely reflect what we will see in our lab projects.

8.1.2 ID3DXMesh

This is the primary mesh container in DirectX and is the one we are likely to use most often. This mesh can be created and initialized manually (using functions like D3DXCreateMesh or D3DXCreateMeshFVF) or automatically as the result of file loading functions (like D3DXCreateMeshFromX). ID3DXMesh inherits all of the functionality of the ID3DXBaseMesh and provides additional functions for data optimization. Optimization involves algorithms for sorting vertex and index buffer data to provide maximum rendering performance.

8.1.3 ID3DXPMesh

ID3DXPMesh provides support for progressive meshes for run-time geometric level of detail (LOD) changes. Through this interface, the number of triangles used to render a model can be increased or decreased on the fly. The algorithm used is based on the VIPM (View Independent Progressive Mesh) technique. Models can have their triangles merged together to reduce the overall polygon count or alternatively, have their surface detail increased by using more of the original triangles in the rendered mesh. This allows us to adjust the detail level of the mesh to suit the needs of our application (ex. meshes further away from the camera might have polygonal detail reduced to speed up rendering).

8.1.4 ID3DXSPMesh

Simplification meshes provide the ability to reduce the vertex and/or triangle count in a model. Values can be provided by the application to specify which aspects of the model are more important than others during the reduction process. This provides a degree of control over which polygons are removed and which wind up being preserved in the final reduced model. Unlike progressive meshes, mesh simplification involves only face reduction and is a one-time-only operation. Since the results of the operation cannot be reversed, simplification is generally reserved for use in either a tool such as a model editor or as a one-time process that takes place during application initialization. This could be useful if you wanted to tailor your mesh triangle counts to suit the runtime environment. For example, you might wish to reduce the level of detail of certain meshes if you find that a software vertex processing device is all that is available on the current machine.

8.1.5 ID3DXPatchMesh

Patch meshes encapsulate the import, management, and manipulation of meshes which make use of higher-order curved surface primitives (called patches). A patch is defined by a series of control points describing the curvature of a surface. Because patch meshes store their data so differently from the other mesh representations, this interface does not inherit from ID3DXBaseMesh; it is derived directly from IUnknown. Most of the ID3DXBaseMesh interface methods would make little sense in terms of a patch mesh. Curved surfaces and higher-order primitives will be covered later in this series.

Let us now examine each mesh type in more detail.

8.2 ID3DXMesh

The ID3DXMesh interface is the basic mesh container in DirectX Graphics. As such, we begin our discussion by looking first at the internals of its data storage and move on to discuss its methods. There are four primary data storage buffers when working with meshes in DirectX: vertex buffers, index buffers, attribute buffers, and adjacency buffers. Let us look at each in turn.

8.2.1 The Vertex Buffer

D3DX meshes contain a single vertex buffer for storage of model vertex data. This is a standard IDirect3DVertexBuffer9 vertex buffer and is identical to the vertex buffers we have been using since Chapter Three. It can be created using any supported FVF, locked and unlocked, and read from and written to just like any other vertex buffer. All ID3DXBaseMesh derived interfaces inherit the LockVertexBuffer and UnlockVertexBuffer methods for obtaining direct access to the mesh's underlying vertex data.

```
HRESULT ID3DXMesh::LockVertexBuffer(DWORD Flags, VOID **ppData)
```

DWORD Flags
These are the standard locking flags that we have used when locking vertex buffers in previous lessons. The flags include `D3DLOCK_DISCARD`, `D3DLOCK_NOOVERWRITE`, `D3DLOCK_NOSYSLOCK`, `D3DLOCK_READONLY` and `D3DLOCK_NO_DIRTY_UPDATE` and can be used in combination to lock the buffer in an efficient manner (see Chapter Three).

VOID **ppData
This is the address of a pointer that will point to the vertex data if the lock is successful.

We release the lock on the mesh vertex buffer using the ID3DXMesh::UnlockVertexBuffer function.

```
HRESULT ID3DXMesh::UnlockVertexBuffer(VOID)
```

The following code snippet demonstrates mesh vertex buffer locking and unlocking. It is assumed that pd3dxMesh is a pointer to an already initialized ID3DXMesh.

```
ID3DXMesh *pd3dxMesh;

// Create the mesh here…

// Lock the vertex buffer and get a pointer to the vertex data
void *pVertices;
pd3dxMesh->LockVertexBuffer( 0, &pVertices );

// Fill the vertex buffer using the pointer

// When we are done, we must remember to unlock
pd3dxMesh->UnlockVertexBuffer();
```

8.2.2 The Index Buffer

D3DX meshes use a single index buffer to store model faces. In the context of the D3DX mesh functions, a face is always a triangle and the index buffer always contains indices that describe an indexed triangle list. Thus, if we called the ID3DXMesh::GetNumFaces method, we will be returned the total number of triangles in the mesh. This will always equal the total number of indices in the index buffer divided by three. D3DX meshes have no concept of quads or N-sided polygons. This is an important point to remember, especially when constructing your own mesh data and manually placing it into the vertex and index buffers of a D3DXMesh object.

Like the vertex buffer, the mesh index buffer is a standard IDirect3DIndexBuffer9 index buffer, as seen in previous lessons. Thus it can be accessed and manipulated in a similar fashion. All ID3DXBaseMesh derived interfaces inherit the LockIndexBuffer and UnlockIndexBuffer methods from the base class. This provides direct access to the mesh's underlying index data.

HRESULT ID3DXMesh::LockIndexBuffer(DWORD *Flags*, VOID **ppData)

DWORD Flags
These are the standard `D3DLOCK` flags that we have used when locking index buffers in previous lessons. The flags include `D3DLOCK_DISCARD`, `D3DLOCK_NOOVERWRITE`, `D3DLOCK_NOSYSLOCK`, `D3DLOCK_READONLY` and `NO_DIRTY_UPDATE` and can be used in combination to lock the buffer in an efficient manner.

VOID ** ppData
This is the address of a pointer that will point to the index data if the lock is successful.

We release the lock on a mesh index buffer with a single call to ID3DXMesh::UnlockIndexBuffer.

HRESULT ID3DXMesh::UnlockIndexBuffer(VOID)

The following code demonstrates how to lock and unlock a mesh index buffer. It is assumed in that pd3dxMesh is a pointer to an already initialized ID3DXMesh.

```
ID3DXMesh *pd3dxMesh;

// Pretend that we create the mesh here

WORD  *pIndices16;
DWORD *pIndices32;

if ( pd3dxMesh->GetOptions() & D3DXMESH_32BIT)
{
        // Lock the index buffer and get a pointer to the vertex data
        pd3dxMesh->LockIndexBuffer( 0, &pIndices32 );

        // This is where you could fill the index buffer using the pointer

        // When we are done we must remember to unlock
        pd3dxMesh->UnlockIndexBuffer();
}
else
{
        // Lock the index buffer and get a pointer to the vertex data
        pd3dxMesh-> LockIndexBuffer( 0, &pIndices16);

        // This is where you could fill the index buffer using the pointer

        // When we are done we must remember to unlock
        pd3dxMesh-> UnlockIndexBuffer();
}
```

In the above code, a call to ID3DXMesh::GetOptions is used to determine whether the mesh index buffer uses 32-bit or 16-bit indices. We will see in a moment that we will specify a number of option flags that inform the mesh creation and loading functions about the properties we would like our mesh buffers to exhibit. These properties include which memory pools we would like to use for our vertex/index buffers or whether the index buffer should contain 16-bit or 32-bit indices. The ID3DXMesh::GetOptions method allows us to retrieve the flags that were used to create the buffer. The return value is a bit-set stored in a DWORD that can be tested against a number of flags.

8.2.3 The Adjacency Buffer

To perform optimization and/or LOD on a mesh, it is necessary to know some information about the connectivity of the faces. That is, we wish to know which faces neighbor other faces. For example, LOD is performed by ‘collapsing’ two or more triangles into a single triangle. In order for this to be possible, the information must be at hand that tells the mesh how faces are connected to other faces. As a face in the context of any of the D3DX mesh types is always a triangle, and since a triangle always has three edges, we know that a single face can at most be adjacent to three other triangles. Therefore, when we need to send adjacency information to the mesh to perform one function or another, we will pass in an array with three DWORDs for each face. These values describe the face index numbers for neighboring

faces. If we have a mesh with 10 faces, then the adjacency information required would be an array of 30 DWORDs.

Fig 8.1 depicts a mesh with eight faces and the associated adjacency array. Keep in mind that while a triangle may be connected along its three edges to at most three other triangles, this does not mean that every triangle will be connected to three other faces. A mesh that contains a single triangle for example would obviously have no adjacent neighbors. Nevertheless, whether each face in the mesh is connected to three other faces or not, we still store three DWORDs per face. If a triangle edge is not connected to any other triangle in the mesh, then we will use a value of 0xFFFFFFFF to indicate this.

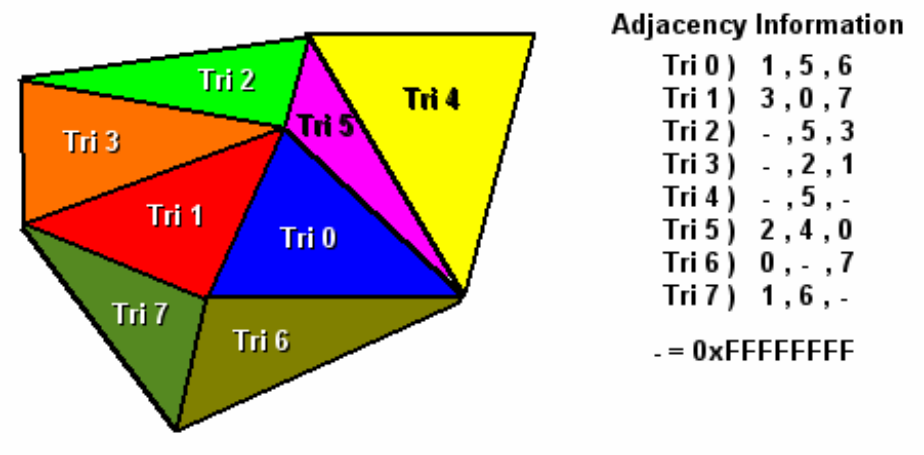


Figure 8.1

Looking at the list in Fig 8.1 we can see that triangle 0 is connected to faces 1, 5, and 6. Triangle 3 is only connected to two other triangles: 1 and 2. Using this adjacency information, a simplification process might decide to collapse triangles 0, 1, 6, and 7 by removing the shared middle vertex and collapsing the four triangles into two triangles.

While calculating face adjacency is not very difficult to do, we are spared even that minor hassle by D3DX. All mesh types derived from the ID3DXBaseMesh interface inherit a function called ID3DXBaseMesh::GenerateAdjacency to generate face adjacency data. To minimize per-mesh memory requirements, D3DX meshes do not generate this information automatically when meshes are created, so we must explicitly call this procedure if we require this data. Since we often require the use of adjacency data only once when optimizing a mesh, there is little point in keeping it in memory after the fact. For progressive meshes however, we may decide to keep this information handy as we will see later.

```
HRESULT ID3DXMesh::GenerateAdjacency(FLOAT fEpsilon,
                                     DWORD*pAdjacency);
```

FLOAT fEpsilon

Sometimes modeling packages create meshes such that faces that are supposed to be adjacent are not precisely so. This can happen for a number of reasons. For example, the level designer may have not used a correct alignment, leaving miniscule but significant gaps between faces. Sometimes this can be due to floating point rounding errors caused by cumulative vertex processing done on the mesh (such as

a recursive CSG process). Further, perhaps the two faces are not actually supposed to be touching, but you would like LOD and optimization functions to treat them as neighbors anyway. Fig 8.2 shows two triangles which should probably be connected, but in fact have a small gap between them. We can see immediately that the triangles would share no adjacent edges.

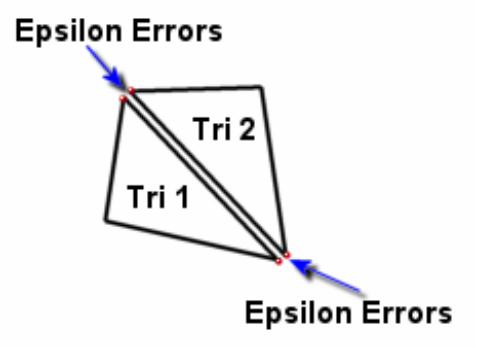


Figure 8.2

It can be frustrating when these tiny gaps prevent proper optimization or simplification. The fEpsilon value allows us to control how sensitive the GenerateAdjacency function is when these gaps are encountered. It is used when comparing vertices between neighboring triangles to test whether they are considered to be on the same edge, and therefore exist in a neighboring face. This is no different than testing floating point numbers or 3D vectors using an epsilon value. It simply allows us to configure the function to be tolerant about floating point errors. The larger the epsilon values, the more leeway will be given when comparing vertices and the more likely they are to be considered the same.

DWORD *pAdjacency
The second parameter is a pointer to a pre-allocated DWORD array large enough to hold three DWORDS for every face in the mesh.

The following code shows how we might generate the adjacency information for an already created mesh.

```
// Allocate adjacency buffer
DWORD *pAdjacency = new DWORD[pd3dxMesh->GetNumFaces() * 3];

// Generate adjacency with a 0.001 tolerance
pd3dxMesh->GenerateAdjacency( 0.001 , pAdjacency );
```

8.2.4 The Attribute Buffer

In Graphics Programming Module I we looked at how to assign textures and materials to our model faces. Recall that for each face we stored indices into global arrays that contained scene materials and textures. This is how we mapped faces in our mesh to textures and materials stored elsewhere in the application. During rendering, we would loop through each texture used by the mesh and render only the

faces that used that texture in a single draw primitive call. We would repeat this process for each texture used by the mesh until all faces had been rendered. D3DX mesh objects also provide this batch rendering technique which we know is so essential for good rendering performance.

Since D3DX meshes store data in vertex and index buffers and include no explicit face type per se, it becomes necessary to organize those buffers such that it is known in advance which groups of indices map to particular textures or materials. As we can no longer store texture and material indices in our mesh faces (because a mesh face is now just a collection of three indices in the mesh's index buffer) another buffer, called an *attribute buffer,* provides the means for doing so.

Whenever a D3DXMesh is created (either manually or via a call to the D3DXLoadMeshFromX function), an attribute buffer is created alongside the standard vertex and index buffers. There is one DWORD entry in this attribute buffer for every triangle in the index buffer. Each attribute buffer entry describes the Attribute ID for that face. All faces that share the same Attribute ID are said to belong to the same *subset* and are understood to require the same device states to be rendered. Therefore, all triangles in a subset can be rendered with a single draw primitive function call to minimize device state changes. All faces that belong to the same subset are not necessarily arranged in the index buffer in any particular order (although they can be, as we will soon discuss), but they will still be rendered together. Note that the mesh object itself does not contain any texture or material information; Attribute IDs provide the only potential link to such external concepts. In short, Attribute IDs provide a means to inform D3DX that faces with like properties can be rendered in a single draw call.

When building a mesh ourselves, the Attribute ID can describe anything we would like. It might be the index of a material or texture in a global array, or even an index into an array of structures that describe a combination of material, texture, and possibly even the lights used by the subset. Ultimately this ID is just a way for the application to inform the ID3DXMesh rendering function which faces belong to the same group and should be rendered together. It is important to understand that the application is still responsible for managing the assets that the face attributes map to (texture, materials, etc.) and for setting up the appropriate device states before rendering the subset.

Fig 8.3 depicts a nine triangle mesh, where each triangle uses one of five different textures. When the ID3DXMesh is created, there are empty vertex, index, and attribute buffers. The mesh itself contains no texture information. The textures used by the mesh are stored in a global texture array managed by the application. This is similar to the approach we took in our lab projects that used IWF files in earlier lessons -- we extracted the texture names from the file, loaded the textures into a global array, and stored the texture index in the face structure. We can no longer store the texture index in the face structure because the ID3DXMesh has no such concept. However we can store the texture index for each face in the attribute buffer instead.

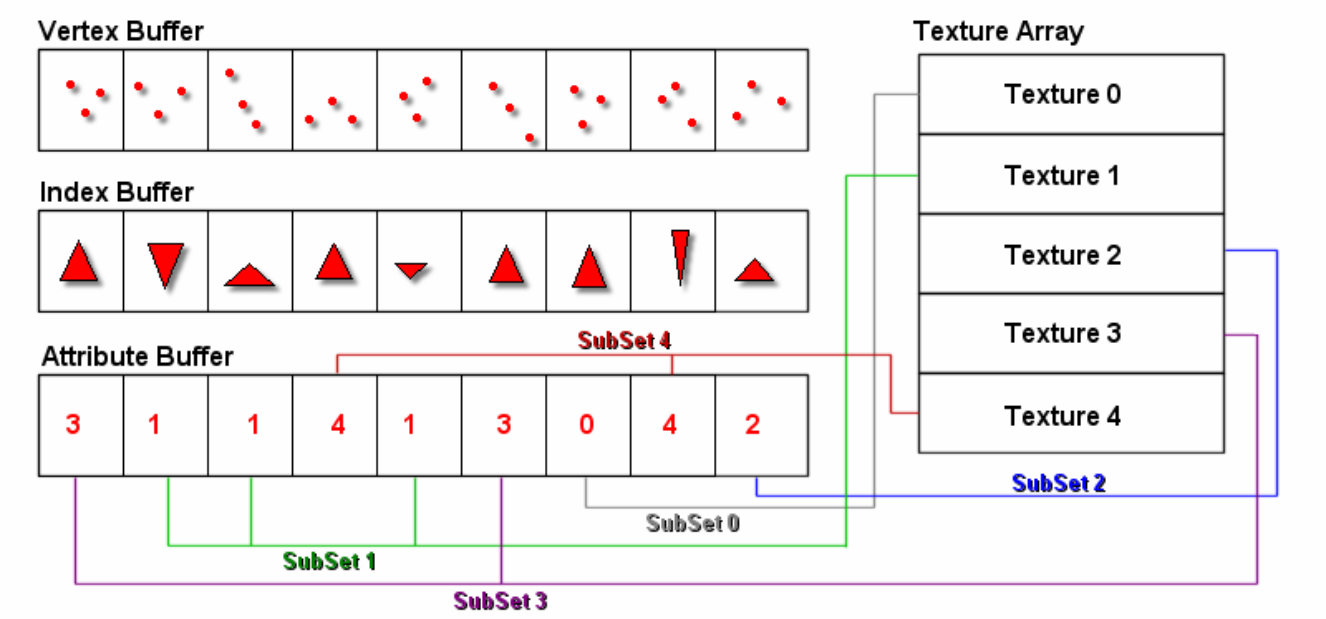


Figure 8.3

In this example, the ID3DXMesh itself has no knowledge that the Attribute IDs are in fact texture indices, but it does know that all faces with a matching Attribute ID are considered part of the same subset and can all be rendered together.

Fig 8.3 depicts the relationship between the triangle data stored in the index buffer and the Attribute ID stored in the attribute buffer. The 7th triangle uses texture 0, the 8th triangle uses texture 2, the 1st and 6th triangles use texture 3, the 2nd, 3rd, and 5th triangles use texture 1, and so on. Even in this simple example the mesh would benefit from attribute batching, as the faces belonging to a particular subset are fairly spread out inside the index buffer.

In a short while we will examine the D3DXLoadMeshFromX function. When the mesh is created using this function, its vertex, index, and attribute buffers are automatically filled with the correct data. To determine which Attribute IDs map to which asset types, the function returns an array of D3DXMATERIAL structures which contain a texture and material used by a given subset as specified by the X file that was loaded. So when we render subset 0 of the mesh, we set the texture and the material stored in the first element of the D3DXMATERIAL array. When rendering the second subset we set the texture and material stored in the second element of the D3DXMATERIAL array, and so on for each subset. Therefore, while the mesh itself does not maintain texture and material data, the D3DXLoadMesh… family of functions does return this information. The loading function correctly builds the attribute buffer such that each Attribute ID indexes into this D3DXMATERIAL array.

When loading an X file, the mesh and its internal buffers are created and filled automatically. While we rarely need to lock any of the mesh's internal buffers under these circumstances, when creating an ID3DXMesh object manually, we definitely need to be able to lock and unlock the attribute buffer.

The ID3DXMesh::LockAttributeBuffer and the ID3DXMesh::UnlockAttributeBuffer are not inherited from the ID3DXBaseMesh interface. Instead these are two of the new functions added to the ID3DXMesh interface beyond its inherited function set.

HRESULT LockAttributeBuffer(DWORD *Flags*, DWORD *ppData*);**

When locking the attribute buffer, we pass in one or more of the standard buffer locking flags to optimize the locking procedure. The second parameter is the address of a DWORD pointer which will point to the beginning of the attribute data on successful function return.

Unlocking the attribute buffer is a simple matter of calling the following function:

HRESULT UnlockAttributeBuffer(VOID);

8.2.5 Subset Rendering

To render a given subset in a mesh, we call ID3DXMesh::DrawSubset and pass in an Attribute ID. This function will iterate over the faces in the index buffer and render only those that match the ID passed in. Therefore, rendering a mesh in its entirety adds up to nothing more than looping through each subset defined in the mesh, setting the correct device states for that subset (texture and material, etc.) and then rendering the subset with a call to the ID3DXMesh::DrawSubset method. More efficient rendering is made possible when the subset data is grouped together in the mesh vertex and index buffers. We will discuss mesh optimization in detail a little later in the chapter.

HRESULT DrawSubset(DWORD *AttribId*);

The mesh in Fig 8.3 could be rendered one subset at a time using the following code.

```
for(int I = 0; I < m_nNumberOfTextures; I++)
{
     pDevice->SetTexture(0, pTextureArray[I]);     // Set Subset Attribute
     pd3dxMesh->DrawSubset(I);                     // Render Subset
}
```

Again it should be noted that while this example is using the texture index as the Attribute ID for each subset, this ID could instead be the index of an arbitrary structure in an array which might hold much more per-face information: materials, transparency, multiple textures, lights, etc. used by that subset. All that matters to D3DX is that faces with the same Attribute ID share like states and can be batch rendered. What these Attribute IDs mean to the application is of no concern to the mesh object. It is the application that is responsible for setting the correct device states before rendering the subset that corresponds to the particular Attribute ID.

8.3 ID3DXBaseMesh Derived Functions

The purpose of most of the base mesh derived functions is pretty self-explanatory, but we will quickly review them just to get a feel for the information that a mesh stores and how to access it. Then in the next section, we will look at how to actually create and use a mesh.

```
HRESULT GetDevice( LPDIRECT3DDEVICE9 *ppDevice );
```

When we create a mesh using the D3DXCreateMesh function or the D3DXLoadMesh… family of functions, we must specify a pointer to the device interface through which the mesh will be created. This is required because the device owns the vertex buffer and index buffer memory that is allocated, and thus the mesh object is essentially owned by the device as well. The ID3DXBaseMesh::GetDevice function allows us to retrieve the device that owns the mesh.

```
DWORD GetFVF(VOID);
```

When we create a mesh using the D3DXCreateMeshFVF function, we directly specify the flexible vertex format flags for the mesh vertex buffer. The ID3DXBaseMesh::GetFVF function allows us to retrieve this information so that we know the layout of the vertices stored in this internal vertex buffer.

When using the D3DXLoadMesh… family of functions, being able to retrieve the FVF flags for the mesh vertex structure is important because we often have no direct control over the original vertex format the mesh was created with. For example, when using the D3DXLoadMeshFromX function, the function will choose a vertex format that most closely matches the vertex information stored in the X file. If the vertices in the X file contain vertex normals and three sets of texture coordinates, the mesh vertex buffer will automatically be created to accommodate this information. Therefore, when the mesh is created from an X file, we need this function to determine the format with which the vertex buffer was initialized. You will see later when we discuss mesh cloning that we are not restricted to using vertices in the format specified in the X file. We can clone the mesh and specify a different vertex format to better suit the needs of our application.

```
HRESULT GetIndexBuffer( LPDIRECT3DINDEXBUFFER9 *ppIB );
```

The ID3DXBaseMesh::GetIndexBuffer function returns a pointer to the IDirect3DIndexBuffer9 interface for the internal mesh index buffer. Retrieving the index buffer interface (or the vertex buffer interface) can be useful if we wish to render the mesh data manually and not use the DrawSubset functionality. If you had your own custom rendering code that you needed to use and merely wanted to use the ID3DXMesh to optimize your dataset, this would be your means for accessing the indices. After the data was optimized, you would simply retrieve the vertex and index buffer interfaces and use them as you would under normal circumstances.

```
HRESULT GetVertexBuffer( LPDIRECT3DVERTEXBUFFER9 *ppVB );
```

Like the previous function, this function returns an interface pointer to the mesh vertex buffer so that you can use it for custom rendering or other application required manipulation.

```
DWORD GetNumBytesPerVertex(VOID);
```

This function returns the size (in bytes) of the vertex structure used by this mesh. This is useful when you need to render the mesh data manually since you need to specify the stride of the vertex when binding the vertex buffer with the IDirect3DDevice9::SetStreamSource function.

```
DWORD GetNumFaces(VOID);
```

This function returns the number of faces (triangles) stored in the mesh index buffer. Since the mesh always stores its geometry as an indexed triangle list, the result of this function will be the total number of indices in the mesh index buffer divided by 3.

```
DWORD GetNumVertices(VOID);
```

This function returns the number of vertices in the mesh vertex buffer.

```
DWORD GetOptions(VOID);
```

This function returns a DWORD which stores a series of flags that were used during creation of the mesh. The flags include information about the memory pools used for the vertex and index buffers, the size of the indices (32-bit or 16-bit) in the index buffer, whether the vertex or index buffer have been created as dynamic buffers, and whether or not they are write-only buffers.

8.4 Mesh Optimization

There is one more topic to discuss before we finally look at how to create/load a mesh: optimization of vertex and index data. While this may seem like a strange way round to do things, it should help us better understand exactly what information the D3DXLoadMeshFromX function is returning to us as well as how to efficiently organize our data when we create meshes manually.

Whether or not you intend to use the ID3DXMesh (or any of its sibling interfaces) for your own mesh storage or rendering, it would be a mistake to overlook the geometry optimization features offered by the ID3DXMesh::Optimize and the ID3DXMesh::OptimizeInPlace functions we are about to examine. Even if you have your own mesh containers and rendering API all worked out, your application may well benefit from temporarily loading the mesh data into an ID3DXMesh and using its optimization functions before copying the optimized data back into your proprietary structures. This can save you some time and energy developing such optimization routines yourself.

It is a sad but true fact that if you are starting off as a hobbyist game developer, you will likely not have access to artists to develop 3D models specifically for your applications. Often you will be forced to use models that you find on the Internet or other places where they may or may not be stored in the file in an optimal rendering arrangement. The Optimize and OptimizeInPlace functions are both introduced in the ID3DXMesh interface and will provide some relief. Both functions perform identical optimizations to the mesh data and, for the most part, take identical parameter lists. The difference between them is that the Optimize function generates a new mesh containing the optimized data and the original mesh

remains intact. The OptimizeInPlace function does not create or return a new mesh, it directly optimizes the data stored in the current mesh object. Let us take a look at the OptimizeInPlace function first.

```
HRESULT OptimizeInPlace( DWORD Flags,
                         CONST DWORD *pAdjacencyIn,
                         DWORD *pAdjacencyOut,
                         DWORD *pFaceRemap,
                         LPD3DXBUFFER *ppVertexRemap
                        );
```

DWORD Flags

The Flags parameter describes the type of optimization we would like to perform. We typically choose only one of the standard optimization flags, but we can combine the standard optimization flags with additional modifier flags. The standard optimization flags are ordered and cumulative. That is, each one listed also contains the flags for the optimizations that precede it.

D3DXMESHOPT_COMPACT – When specifying this flag, the optimizer removes vertices and/or indices that are not required. A classic example of this would be stray vertices in the vertex buffer that are never referenced by any indices in the index buffer. It is also possible that the geometry from which the mesh was created contains degenerate triangles. Because meshes are always stored as triangle lists and degenerate triangles are of little use in these situations, they will also be removed. This is the base optimization method that is performed by all other optimization flags. It is also the quickest and cheapest optimization that we can perform. Note that it may involve index and vertex data reshuffling or reordering in order to fulfill the compaction requirements.

D3DXMESHOPT_ATTRSORT - The attribute sort optimization is performed in addition to the compaction optimization listed above. That is, you do not have to specify both `D3DXMESHOPT_COMPACT` and `D3DXMESHOPT_ATTRSORT` since the `D3DXMESHOPT_ATTRSORT` flag by itself will cause a compaction and an attribute sort optimization to be performed. When a mesh is first created, either in code or by the D3DXLoadMeshFromX function, faces are typically added to the index buffer as they are encountered. This means that we may have faces that share the same Attribute ID (and are therefore part of the same subset) randomly dispersed within the index buffer. The optimizer sorts the data stored within the mesh so that all the faces that share the same Attribute ID are consecutive within the index buffer. This allows them to be efficiently batch rendered.

Fig 8.4 shows how a D3DX mesh might be created. In this example we shall assume that we have loaded geometry from an IWF file and stored it inside a D3DXMesh. This would be a simple case of calling D3DXCreateMesh to create the empty mesh object and then locking the vertex and index buffers and copying in the IWF data (we discussed how easy it was to load IWF files into memory with the IWF SDK in Chapter Five). Once we have the data file in memory, we loop through each face that was loaded (temporarily stored in the IWFSurface vector) and copy the vertices of the face into the vertex buffer and the indices of the face into the index buffer (arranged as indexed triangle lists).

To keep this example simple, we have used only the render material referenced by the surface as the Attribute ID for the face. Thus all faces that use the same material in the IWF file are said to belong to the same subset. The point here is not how the data gets into the mesh, but that when we add data in an arbitrary order to the internal buffers, triangles belonging to the same subset may not be consecutive in the buffers. In this example, we added nine triangles to the mesh in an arbitrary order; the order that they were encountered in the external file. When we add each triangle, we add its three vertices to the vertex buffer, its indices to the index buffer, and the triangle Attribute ID (the material index) to the attribute buffer.

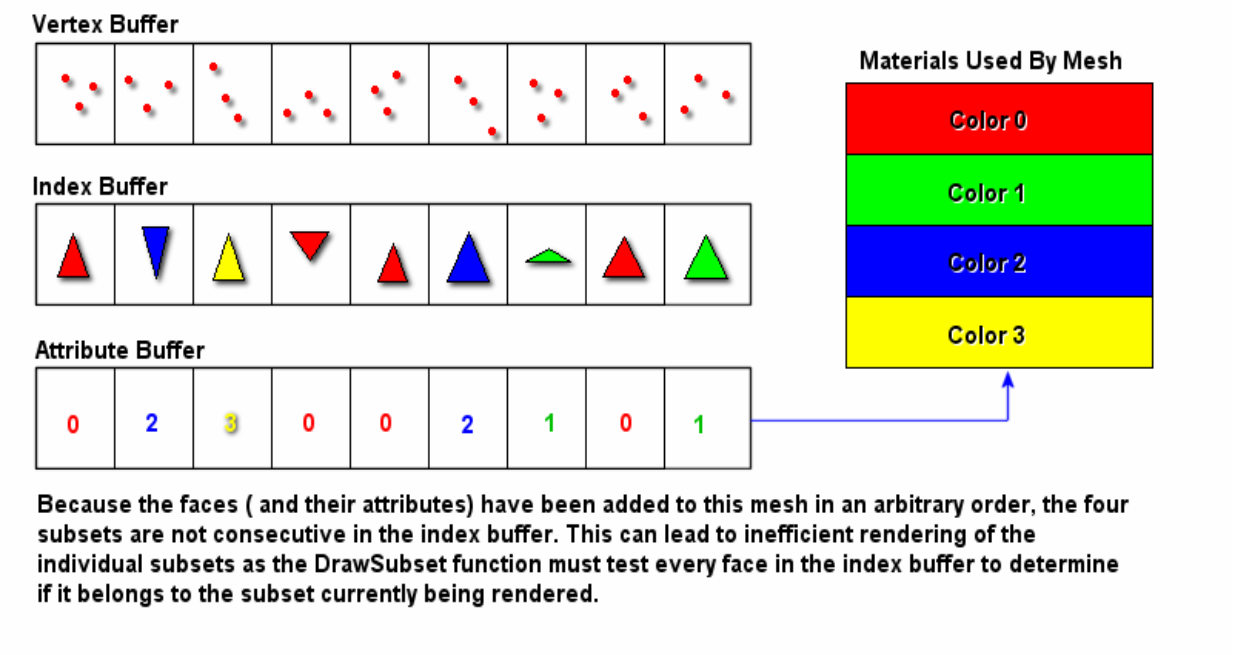


Figure 8.4

Given the disorganized nature of the buffers in Fig 8.4, if we were to call the ID3DXMesh::DrawSubset method and pass in an ID of 0, the function would need to render the 1st, 4th, 5th, and 8th triangles. Because the mesh cannot assume that all triangles belonging to the same subset are batched together by default, when we issue the render call for a subset, the function must loop through the entire attribute buffer to find and render all triangles that match the selected Attribute ID one at a time.

When we use the `D3DXOPTMESH_ATTRSORT` flag, the vertex and index buffers will be reordered such that vertices and indices belonging to the same subsets are batched together in their respective buffers. The function internally builds an *attribute table* that describes where a subset's faces begin and end in the index buffer and where the subset's vertices begin and end in the vertex buffer. This sets the stage for efficient DrawIndexedPrimitive calls.

Fig 8.5 shows how the mesh might look after an attribute optimization has been performed. Study the following image and see if you notice anything strange about the attribute buffer.

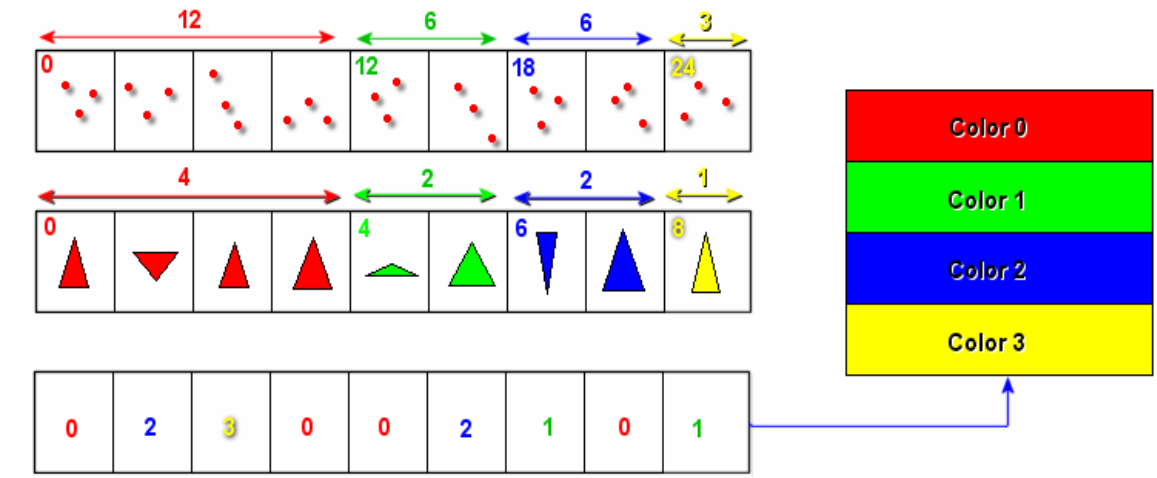


Figure 8.5

Notice how the attribute buffer no longer correctly maps to the faces. When we perform an attribute sort on a mesh, the mesh is flagged internally as having been attribute sorted and an attribute table is built. From that point on, the attribute buffer is no longer used in any calls to the DrawSubset function. Instead, batch rendering information is pulled from the from the attribute table. The DrawSubset function no longer needs to loop through the attribute buffer for each subset and find the matching triangles, because the attribute table describes where a subset begins and ends in the index and vertex buffers.

We can see in Fig 8.5 that the attribute sort has correctly batched all faces belonging to the same subset in the index and vertex buffers. It also shows how subset 0 starts at face 0 in the index buffer and consists of four triangles. The vertices for subset 0 begin at vertex 0 in the vertex buffer and the range consists of 12 vertices. Now the DrawSubset function can quickly jump to the beginning of the subset in the vertex and index buffers and pump them into the pipeline with a single DrawIndexedPrimitive call.

Once a mesh has been attribute sorted, we can gain access to the internal attribute table using the ID3DXBaseMesh::GetAttributeTable method. This returns an array of `D3DXATTRIBUTERANGE` structures (one per subset) describing where a given subset begins and ends in the vertex and index buffers. If you intend to perform geometry or attribute manipulation on an already optimized mesh, you may want to update the attribute table with the new information rather than calling the Optimize function again.

The `D3DXATTRIBUTERANGE` range structure is shown below.

```
typedef struct _D3DXATTRIBUTERANGE {
    DWORD AttribId;
    DWORD FaceStart;
    DWORD FaceCount;
    DWORD VertexStart;
    DWORD VertexCount;
} D3DXATTRIBUTERANGE;
```

The first member of the structure contains the zero-based Attribute ID for the subset. If this value was set to 5 for example, then this structure would describe where the vertices and indices for subset 5 begin and end in the vertex and index buffers. The FaceStart member holds the triangle number (offset from zero) where the subset begins in the index buffer. The FaceCount member describes how many triangles beginning at FaceStart belong to the subset. The VertexStart member describes the zero-based index of the vertex in the vertex buffer where the subset vertices begin. The VertexCount member describes the number of vertices that belong to the subset. Essentially, these are the members you would use if you decided to call DrawIndexedPrimitive yourself.

To retrieve the attribute table, we use the ID3DXBaseMesh inherited function GetAttributeTable.

```
HRESULT GetAttributeTable
(
    D3DXATTRIBUTERANGE *pAttribTable,
    DWORD *pAttribTableSize
);
```

The first parameter is a pointer to a pre-allocated array of `D3DXATTRIBUTERANGE` structures that the function will fill with subset information. The second parameter is the number of subsets (starting at zero) we would like to retrieve information for. We must make sure that we have allocated at least as many `D3DXATTRIBUTERANGE` structures as the number specified in the pAttribTableSize parameter.

If we would like all of the information for every subset the mesh contains, we must first know how many `D3DXATTRIBUTERANGE` structures to allocate memory for. If we call this function and set the first parameter to NULL and pass the address of a DWORD as the second parameter, the function will fill the passed DWORD with the number of subsets in the mesh. This allows us to allocate the memory for the `D3DXATTRIBUTERANGE` structures before issuing a second call to the function to retrieve the actual attribute table. It should be noted that an attribute table does not exist until the mesh has undergone an attribute sort. If you call this function without first attribute sort optimizing the mesh, 0 will be returned in the pAttributeTableSize parameter.

The following code snippet demonstrates how we might optimize a mesh such that it is compacted and attribute sorted. We then retrieve the mesh attribute table. The code assumes that pd3dxMesh has already been created and filled with data but has not yet been optimized.

```
// Allocate adjacency buffer needed for optimize
DWORD *pAdjacency = new DWORD[ pd3dxMesh->GetNumFaces() * 3];

// Generate adjacency with a 0.001 tolerance
pd3dxMesh->GenerateAdjacency( 0.001 , pAdjacency );

// Optimize the mesh ( D3DXOPTMESH_ATTRSORT = Attribute Sort + Compact )
pd3dxMesh->OptimizeInPlace(D3DXOPTMESH_ATTRSORT,pAdjacency,NULL,NULL,NULL);

// No longer need the adjacency information
delete []pAdjacency;
```

```
// We need to find out how many subsets are in this optimized mesh
DWORD NumberOfAttributes;

// Get the number of subsets
pd3dxMesh->GetAttributeTable( NULL , &NumberOfAttributes );

// Allocate enough D3DXATTRIBUTERANGE structures, one for each subset
D3DXATTRIBUTERANGE *pAttrRange = new D3DXATTRIBUTERANGE[NumberOfAttributes];

// Call the function again to populate array with the subset information.
pd3dxMesh->GetAttributeTable( pAttrRange , &NumberOfAttributes );

// Examine the actual attribute range data here and do what you want with it
```

Notice that in order to perform an optimization of any kind, we must supply the function with face adjacency information.

The ID3DXMesh interface includes an additional function (beyond the ID3DXBaseMesh inherited functions) that can be used to specify an attribute table manually.

HRESULT SetAttributeTable(CONST D3DXATTRIBUTERANGE *_pAttrTable_,
DWORD _cAttrTableSize_);

The first parameter is the new array of `D3DXATTRIBUTERANGE` structures and the second parameter is the number of attributes in the array. It is very rare that you will need to modify the attribute table of an optimized mesh, but if necessary, then this is the function you should use to set the new data. Note that it does not alter the index and vertex buffer information in any way; it simply sets the attribute table internally. When a mesh has been attribute sorted, it is marked as having been so.

> **Note:** When you set the attribute table manually you must make sure that you correctly identify the subset ranges, because the DrawSubset function will blindly use this information to render each subset. When we call DrawSubset for example and pass in an attribute ID of 5, the subset's vertex and index start and count values will be taken from this table regardless of what is stored in the attribute buffer.

D3DXMESHOPT_VERTEXCACHE – This flag is used to inform the optimizer that we would like the vertex and index buffers of the mesh to be optimized to better utilize the vertex cache available on most modern hardware accelerated 3D graphics cards. This will often involve reordering the vertex and index data of the mesh such that vertices that are transformed and stored in the vertex cache on the GPU are reused as much as possible. If a vertex is shared by 10 faces for example, this optimization will try to order the index buffer and vertex buffer such that this vertex is only transformed and lit once. It can then render all faces that use the vertex before it is evicted from the vertex cache. This is an extremely effective optimization which aims to increase the cache hit rate of hardware vertex caches.

Again, specifying this single flag also includes the optimizations performed by the two previous flags just discussed. Therefore, specifying `D3DXMESHOPT_VERTEXCACHE` also performs the `D3DMESHOPT_COMPACT` and `D3DXMESHOPT_ATTRSORT` optimization processes. The `D3DXMESHOPT_VERTEXCACHE` flag and the `D3DMESHOPT_STRIPREORDER` flags (discussed next) are mutually exclusive since they have very different goals.

D3DXMESHOPT_STRIPREORDER – The name of this optimization flag can sometimes cause some confusion. It seems to imply that the vertex and index buffer data would be modified to be rendered as indexed triangle strips, but this is not the case. A D3DXMesh will always be stored using an indexed triangle list format. So what does this optimization actually do?

Some 3D graphics cards can only render triangle strips at the hardware level. This means that when we render triangle lists or fans for example, the driver will convert these primitive types to strips before rendering. This is not something we are ever aware of or have to make provisions for, but this strip creation process can carry overhead on such cards. While the overhead is generally minimal, this optimization flag makes it clear that we would like the indexed triangle lists of the mesh to be re-arranged such that they can be more quickly be converted into strips by the driver. The optimizer reorders the faces so that, as much as possible, physically adjacent triangles are consecutively referenced. This increases the length of adjacent triangle runs in the buffer, which helps speed things up in cases where strip generation is required.

This flag and `D3DXMESHOPT_VERTEXCACHE` are mutually exclusive due to the fact that they both try to attain a best-fit ordering given a different set of rules. The `D3DOPTMESH_STRIPREORDER` flag, like the `D3DMESHOPT_VERTEXCACHE` flag, contains all the optimization processes previously described.

The optimization flags can thus be summed up as follows:

```
D3DXMESHOPT_COMPACT      =    Compact process only.
D3DXMESHOPT_ATTRSORT     =    Compact + Attribute Sort.
D3DXMESHOPT_VERTEXCACHE  =    Compact + Attribute Sort + Cache Reorder.
D3DXMESHOPT_STRIPREORDER =    Compact + Attribute Sort + Strip Reorder.
```

Modifier Flags

There are a number of modifier flags that can be combined with any of the previously discussed standalone optimization flags. They are listed below along with a brief description of their purpose.

D3DXMESHOPT_IGNOREVERTS - This flag instructs the optimizer not to touch the vertices and to work only with the face/index data. Thus, if we were to perform a compact optimization with this modifier flag, no vertices would be removed -- even if they were never referenced. In that case, only unused face indices would be removed. This is a useful modifier flag if you wish to optimize the mesh but want the vertex buffer to remain unchanged.

D3DXMESHOPT_DONOTSPLIT - This is a modifier flag that is used primarily for attribute sorting. When an attribute sort optimization is performed, vertices that are used by multiple attribute groups may be split (i.e. duplicated) so that a best-order scenario is obtained when batch rendering, while keeping vertices in a local neighborhood to the subset. This way, software vertex processing performance is not compromised.

For example, let us say that we had 300 attributes (subsets) and 5000 vertices. Assume that attribute 0 and attribute 299 both reference vertex 4999. In this case, the vertex attribute range for attribute 0 would span the entire contents of the vertex buffer. This would seriously hurt

software vertex processing performance since we know that when calling DrawIndexedPrimitive on a software vertex processing device, the entire range of specified vertices is transformed in one pass. In this example then, attribute 0 would have a vertex start index of 0 and a vertex count of 5000. So even if this was only a single triangle subset, all 5000 vertices would need to be transformed and possibly lit when rendering subset 0. This is not a problem when using a hardware vertex processing device because the vertex cache is used instead. So in order for mesh performance to downgrade gracefully to a software vertex processing device, under these conditions the vertex may be duplicated and moved to the beginning of the buffer. Attribute 0 would then have a localized set of vertices to minimize block vertex transformations on a software vertex processing device.

This duplication process is performed automatically by the attribute sort optimization process and normally that is a good thing. However, if for any reason you consider this a problem and wish to prevent the automatic duplication of vertices, specifying this flag will prevent this duplication optimization from being performed.

D3DXMESHOPT_DEVICEINDEPENDENT - During vertex cache optimization, a specific vertex cache size is used which coincides with the cache provided on the hardware. This approach works well for modern hardware. This flag specifies that the optimizing routine should alternatively assume a default, device-independent vertex cache size (a fixed cache size) because this usually works better on older hardware.

Example Combinations:

```
D3DXMESHOPT_VERTEXCACHE|D3DXMESHOPT_DEVICEINDEPENDANT|D3DXMESHOPT_DONOTSPLIT
```

With this combination, the data will be compacted to remove un-referenced vertices and degenerate triangles and will be attribute sorted. When the attribute sort is being performed, the optimization routine will not duplicate vertices to minimize subset vertex ranges. This would make the mesh transform and render much slower on a software vertex processing device. The mesh will also have its triangles and vertices reordered to better utilize the vertex cache. In this case we want a fixed, device-independent vertex cache size to be assumed.

```
D3DXMESHOPT_ATTRSORT|D3DXMESHOPT_IGNOREVERTS
```

This combination instructs the optimization routine to compact and attribute sort only the index data and not alter the vertex buffer in any way. If there are any un-referenced vertices in the vertex buffer, they will not be removed. Furthermore, since we are stating that we do not want the vertex data touched, vertices will not be duplicated to produce better localized vertices for a subset. This means that such a mesh will suffer the same poor software vertex processing performance if an attribute/subset references vertices over a large span of the vertex buffer.

CONST DWORD *pAdjacencyIn

This is a pointer to an adjacency array calculated using ID3DXBaseMesh::GenerateAdjacency. As discussed, this is an array of DWORDS (3 per face) describing how faces are connected. Although it might be argued that it would be nice if the Optimize and OptimizeInPlace functions called

GenerateAdjacency automatically, this would require that the adjacency information be recalculated with every call to the optimize functions. Since you may already have the adjacency information or need to call optimize several times for several copies of the mesh, you can just calculate the adjacency information once and pass it in each time it is needed.

DWORD *pAdjacencyOut

Most mesh optimizations involve the rearranging of vertices and indices in the internal buffers. Since it is possible that faces may be removed by the compaction algorithm or rearranged by the attribute sort/ vertex cache optimizations, the adjacency index information has likely changed as well. If your application needs to maintain the adjacency data, then this pointer can be filled with the new adjacency information when the function returns. If we pass NULL, the adjacency information will not be returned. Thus if you need the adjacency information in the future, you will need to call the GenerateAdjacency function again.

Since the optimization process never introduces new faces (at most it only removes triangles), as long as this buffer is at least as large as the original adjacency buffer (pAdjacencyIn), there will be enough room to house the adjacency information of the optimized mesh.

DWORD *pFaceRemap

This parameter allows us to pass in a pointer to a pre-allocated array that on function return will contain information describing how faces have been moved around inside the index buffer after the optimization step. This array is useful when you have external attributes/properties associated with the mesh triangles that need to be updated after the mesh is optimized. On function return, the following relationship will be true:

pFaceRemap[***OldFaceIndex***] = ***NewFaceIndex***

For example, imagine that you were writing a level editing system and that you have several decals in your levels that are attached to faces by an index. When the mesh is optimized, the faces may have been rearranged such that each face now has a completely different index. In such a case, you would want to update the face index stored in your decal structure so that it tracked the faces it was attached to after the optimization process. If the decal was attached to face 5 for example, after the optimization face 5 may have been moved to face slot 10 in the index buffer. The decal would need to be updated so that it did not blindly attach itself to face index 5, which would now be a completely different face.

Before calling the optimization function, you can inquire how many faces the mesh has and allocate an array of DWORDS (1 per face) and pass this buffer into the function. When the function returns, each element in the array will describe the new post-optimization index for each face. In our example we said that the decal was attached originally to face 5. When the optimization was performed, this face had been moved to become face 10 in the index buffer. In this case, we could simply check the value stored in pFaceRemap[5] and it would hold a value of 10 -- the new location of the original face 5. We could then update the face index stored in the decal.

If you do not require this information about the post optimized mesh (which is often the case), you can set this parameter to NULL and no face remap information will be returned.

LPD3DXBUFFER *pVertexRemap

The final parameter is the address of a pointer to an ID3DXBuffer interface. In principle, it serves the same purpose for vertices as the previous parameter did for faces. The ID3DXBuffer interface provides access to generic memory buffers that are used by various D3DX functions to return or accept different data types (vertex or adjacency information for example). Unlike the face remap parameter where we pre-allocated the array, the vertex remap parameter should be the address of a pointer only. This will be used to allocate the buffer inside the function and return the information to the caller. If your application does not need the returned vertex buffer re-map information, you can just pass NULL. But if you do use it, make sure to release the ID3DXBuffer interface to free the underlying memory.

If we do not specify the `D3DMESHOPT_IGNOREVERTS` modifier flag, it is likely that the vertex data was reorganized during the call. Therefore, we will get back a buffer that contains enough space to hold one DWORD for every vertex in the original pre-optimized mesh. For each element in the returned buffer the following relationship is true.

pVertexRemap [***OldVertexIndex***] = ***NewVertexIndex***

Optimization Example I

We have now covered all of the parameters to the ID3DXMesh::OptimizeInPlace function. We have seen that the optimization features are quite impressive and extremely easy to use. Most of the time we will not need the face re-map and vertex re-map information or the newly compiled adjacency information and we will pass NULL as the last three parameters. However, the following code shows how we might optimize the mesh and store all returned information just in case. This code assumes that pd3dxMesh is a pointer to a valid ID3DXMesh interface.

```
// Allocate adjacency buffer needed for optimize
DWORD *pOldAdjacency = new DWORD[ pd3dxMesh->GetNumFaces() * 3];
DWORD *pNewAdjacency = new DWORD[ pd3dxMesh->GetNumFaces() * 3];
DWORD *pFaceRemap    = new DWORD[ pd3dxMesh->GetNumFaces()    ];
ID3DXBuffer* pVertexRemap;

// Generate adjacency with a 0.001 tolerance
pd3dxMesh->GenerateAdjacency( 0.001 , &pOldAdjacency );

// Optimize the mesh ( D3DXOPTMESH_ATTRSORT = Attribute Sort + Compact )
pd3dxMesh->OptimizeInPlace ( D3DXMESHOPT_ATTRSORT, pOldAdjacency,
                             pNewAdjacency, pFaceRemap, &pVertexRemap );
```

Optimization Example II

In the next example we do not need the face re-map, vertex re-map, or new adjacency information. This simplifies the code to only a few lines. Here we perform a compaction, attribute sort, and a vertex cache optimization.

```
// Allocate adjacency buffer needed for optimize
DWORD *pOldAdjacency = new DWORD [ pd3dxMesh->GetNumFaces() * 3];

// Generate adjacency with a 0.001 tolerance
pd3dxMesh->GenerateAdjacency( 0.001 , &pOldAdjacency );

// Optimize the mesh ( D3DXMESHOPT_VERTEXCACHE = Cache + Attribute Sort + Compact )
pd3dxMesh->OptimizeInPlace(D3DXMESHOPT_VERTEXCACHE, pOldAdjacency,
                           NULL, NULL, NULL );
```

The ID3DXMesh::Optimize function works in almost exactly the same way as the OptimizeInPlace call we just studied. The exception is that the current mesh data is not altered in any way. Instead, a new mesh is created and the optimization takes place there. Once the function has returned, you will have two copies of the mesh: the original un-optimized version whose interface you used to issue the ID3DXMesh::Optimize call, and a new optimized D3DXMesh object. These meshes have no interdependencies, so the original un-optimized mesh could be released if no longer needed.

```
HRESULT Optimize
(
    DWORD Flags,
    CONST DWORD *pAdjacencyIn,
    DWORD *pAdjacencyOut,
    DWORD *pFaceRemap,
    LPD3DXBUFFER *ppVertexRemap,
    LPD3DXMESH   *ppOptimizedMesh
);
```

Unless otherwise specified, when we use the ID3DXMesh::Optimize function, the newly generated mesh will inherit all of the creation parameters of the original mesh.

> **Note:** When we create (or clone) a mesh, we can specify one or more creation flags using members of the D3DXMESH enumerated type. This allows us to control things such as the resource pools the index and vertex buffers are created in. Because the Optimize function is creating a new mesh for the optimized data, we can also combine members of the D3DXMESH enumerated type with the usual D3DXMESHOPT flags discussed previously. This allows us to specify not only the optimization flags we require, but also the creation flags for the newly created optimized mesh.
>
> Although the D3DXMESH enumerated type will be covered shortly, you should know for now that we cannot use the D3DXMESH32_BIT, D3DXMESH_IB_WRITEONLY and D3DXMESH_WRITEONLY mesh creation flags during this call.

8.5 ID3DXBuffer

Before we cover the mesh loading routines, let us briefly discuss the ID3DXBuffer interface in a little more detail. The ID3DXBuffer is used to return arbitrary length data from D3DX functions. It is used in a number of function calls to return things like error message strings, face adjacency, and vertex and material information from optimization and loading functions.

Once a buffer of information has been returned from a D3DX function, you can save it for later use, retrieve a pointer to the buffer data area, or discard the buffer information by releasing the interface. Typically we will copy the data out to more useful structures used by our application. Just remember to release the buffer interface when it is no longer needed.

This interface is derived from IUnknown and exposes only two methods. One returns the size of the buffer and the other returns a void pointer to the raw buffer data. The two methods of the ID3DXBuffer interface are shown and described next.

```
DWORD ID3DXBuffer::GetBufferSize(VOID);
```

This function returns a DWORD describing the size of the buffer in bytes.

```
LPVOID ID3DXBuffer::GetBufferPointer(VOID);
```

This function returns a void pointer to the raw buffer data. This is similar to a pointer that gets returned when we lock a vertex buffer or a texture. Notice that there is no concept of locking or unlocking an ID3DXBuffer object like we saw with other resources.

To use the ID3DXBuffer interface for storing your own data collections, you can create an ID3DXBuffer object by calling the global D3DX function D3DXCreateBuffer.

```
HRESULT D3DXCreateBuffer(DWORD NumBytes, LPD3DXBUFFER *ppBuffer);
```

To allocate an ID3DXBuffer using this function, we pass in the size of the buffer to allocate and the address of an ID3DXBuffer interface pointer. If the function is successful, this second parameter will point to a valid ID3DXBuffer interface.

8.6 Mesh Loading Functions

8.6.1 D3DXLoadMeshFromX

The global D3DX function D3DXLoadMeshFromX creates a new D3DXMesh and loads X file data into its vertex, index, and attribute buffers. It basically creates a ready-to-render mesh for our application with a single function call. However, no data optimization will have been performed on the returned mesh, so if you want to optimize the resulting mesh, you will need to do this as a separate step with a call to ID3DXMesh::OptimizeInPlace after the mesh has been loaded. More often than not, you will be loading your meshes from X files. This function does all the hard work of parsing the X file data and creating a new mesh for you. It is shown below along with a description of its parameter list.

```
HRESULT D3DXLoadMeshFromX
(
    LPCTSTR pFilename,
    DWORD Options,
    LPDIRECT3DDEVICE9 pDevice,
    LPD3DXBUFFER* ppAdjacency,
    LPD3DXBUFFER* ppMaterials,
    LPD3DXBUFFER* ppEffectInstances,
    DWORD* pNumMaterials,
    LPD3DXMESH* ppMesh
);
```

LPCTSTR pFilename

This is a string containing the file name of the X file we wish to load. If the compiler settings require Unicode, the data type LPCTSTR resolves to LPCWSTR. Otherwise, the string data type resolves to LPCSTR.

DWORD Options

Here we specify one or more flags from the `D3DXMESH` enumerated type describing how we would like the mesh to be created. This enumerated type contains members that can be used for all mesh types (including ID3DXPatchMesh, which is not covered in this chapter), so not all members will be valid for ID3DXMesh creation. These flags control items like the vertex and index buffer resource pool and whether the buffers are static or dynamic. These are basically wrappers around the standard vertex buffer and index buffer creation flags we discussed in Chapter Three.

D3DXMESH_32BIT

If this flag is specified then the index buffer will be created with 32-bit indices instead of the default 16-bit indices. This is analogous to specifying the `D3DFMT_INDEX32` usage flag when creating an index buffer using the IDirect3DDevice9::CreateIndexBuffer function.

D3DXMESH_DONOTCLIP

This informs the mesh to create its underlying vertex and index buffers such that they will never require clipping by the pipeline. This is analogous to the `D3DUSAGE_DONOTCLIP` flag used when manually creating a vertex buffer with the IDirect3DDevice9::CreateVertexBuffer function. The default state (the absence of this flag) is to create buffers that pass through the clipping process of the pipeline. If you know that your mesh will always be rendered such that it is within the bounds of the render target, this flag can speed up pipeline processing.

D3DXMESH_POINTS

This is a simple wrapper around creating a vertex buffer with the `D3DUSAGE_POINTS` flag. This is used for rendering a special primitive called a point sprite. Point sprites will be discussed in Module III.

D3DXMESH_RTPATCHES
D3DXMESH_NPATCHES

These flags can be used to create the mesh for storage of higher-order primitives and N-patches. These types are beyond the scope of this chapter and will be covered later in this course series. They are analogous to creating vertex buffers and index buffers manually using the `D3DUSAGE_RTPATCHES` and `D3DUSAGE_NPATCHES` flags.

D3DXMESH_VB_SYSTEMMEM
D3DXMESH_VB_MANAGED

These two flags are mutually exclusive. They allow us to specify the resource pool where we would like the mesh vertex buffer stored. By default, if we do not specify one of these flags, the vertex buffer will be created in the default memory pool. The same rules about resource persistence apply here when a device is lost and reset (see Chapter Three). Default pool buffers become invalid when a device is lost. This requires having to release and recreate the mesh from scratch using either the D3DXLoadMesh… or D3DXCreateMesh functions. To avoid this step when using the default pool, a common practice is to store two meshes: one in system memory and one in video memory. When the video memory copy becomes lost, it is released and

recreated by cloning from the system memory mesh. As discussed in Chapter Three, all of this can be avoided through the use of the managed memory pool (D3DXMESH_VB_MANAGED). When the device is lost and reset, the DirectX memory manager will automatically handle the release and recreation of the internal buffers.

D3DXMESH_VB_WRITEONLY
This is analogous to specifying the D3DUSAGE_WRITEONLY flag during normal vertex buffer creation. In Chapter Three we discussed using this flag for optimal rendering performance. However, we also learned that we should not do this if we intend to read from the vertex buffer since the lock call could potentially fail or return an aliased pointer directly into video memory, resulting in terrible performance.

When using this flag to create a mesh, we have to be very careful because many of the mesh functions (GenerateAdjacency, OptimizeInPlace, and even D3DXLoadMeshFromX) require read access to the vertex and index buffers. Specifying this flag will cause these functions to fail. Note that even though this flag can be specified in the D3DXLoadMeshFromX function, the function will fail if this is the case.

So it would seem as if there is no way to create a mesh with the D3DXMESH_VB_WRITEONLY flag and then optimize it, because to optimize a mesh we first need to generate the adjacency information (which will fail if the vertex buffer is write-only). Furthermore, the vertex buffer itself will usually be read from and reordered during an attribute sort (which will also fail if the mesh has a write-only vertex buffer).

To get the benefit of optimized write-only vertex buffers in our meshes, we will load the mesh into a temporary mesh (usually in system memory) created without the D3DXMESH_VB_WRITEONLY flag. At this point, adjacency information can be generated and optimization performed. Finally, we will clone the mesh and specify that the clone have the write-only flag set. We will discuss mesh cloning later in this chapter.

D3DXMESH_VB_DYNAMIC
This flag is analogous to specifying D3DUSAGE_DYNAMIC when creating a normal vertex buffer. Because dynamic vertex buffers carry some overhead due to a buffer swapping mechanism that allows the buffer to be locked without stalling the pipeline, you should only create a mesh with a dynamic vertex buffer if you intend to lock the and alter the vertex buffer contents in time critical situations. Dynamic vertex buffers were discussed in Chapter Three.

D3DXMESH_VB_SOFTWAREPROCESSING
This flag is analogous to the D3DUSAGE_SOFTWAREPROCESSING flag that we can specify when creating a vertex buffer manually. When using a mixed-mode device, we must indicate when a vertex buffer (and therefore a mesh) is created whether we intend to render it when the device is in software or hardware vertex processing mode. By default, the mesh vertex buffer will be created for hardware vertex processing. If we intend to use the mesh in software vertex processing mode on a mixed-mode device, we must specify this flag.

D3DXMESH_IB_SYSTEMMEM
D3DXMESH_IB_MANAGED
D3DXMESH_IB_WRITEONLY
D3DXMESH_IB_DYNAMIC
D3DXMESH_IB_SOFTWAREPROCESSING
The five flags listed above control how the index buffer of the mesh is generated. They are analogous to their vertex buffer counterparts previously discussed. Vertex buffers and index buffers are stored on the same type of memory surface, so the semantics of both are the same.

D3DXMESH_VB_SHARE
Later in the chapter we will discuss mesh cloning using the ID3DXBaseMesh::Clone and ID3DXBaseMesh::CloneFVF functions. By default these functions create a new mesh object and copy the data from the original mesh into its vertex, index, and attribute buffers. When we specify this flag as an option to the ID3DXBaseMesh::Clone function, the result will be a new mesh object that shares the vertex buffer with the original mesh. This is often referred to as *instancing* (see Chapter One). Any modifications made to the master vertex buffer will affect all cloned meshes. It should be noted that this is not a valid mesh creation flag for the D3DXCreateMesh or D3DXLoadMeshFromX functions. It should only be used in a mesh cloning operation.

D3DXMESH_USEHWONLY
This flag is only valid for the ID3DXSkinInfo::ConvertToBlendedMesh function. It creates a mesh that has per-vertex blend weights and a bone combination table from a standard ID3DXMesh. Skinned meshes will be covered later in this course, so we can ignore this flag for now.

The next five members are combinations of the previous flags discussed. They emerge from the concept that we will often create both the index buffer and the vertex buffer in the same resource pools using the same functionality.

D3DXMESH_SYSTEMMEM
Informs the mesh creation functions that we wish both the vertex buffer and the index buffer of the mesh to be created in system memory. Equivalent to specifying both `D3DXMESH_VB_SYSTEMMEM` and `D3DXMESH_IB_SYSTEMMEM`.

D3DXMESH_MANAGED
Informs the mesh creation functions that we wish both the vertex buffer and the index buffer of the mesh to be created in the managed memory pool. Equivalent to specifying both `D3DXMESH_VB_MANAGED` and `D3DXMESH_IB_MANAGED`.

D3DXMESH_WRITEONLY
Specifying this flag will create the mesh index buffer and vertex buffer with the write-only flag. Equivalent to specifying both `D3DXMESH_VB_WRITEONLY` and `D3DXMESH_IB_WRITEONLY`.

D3DXMESH_DYNAMIC
Equivalent to specifying both `D3DXMESH_VB_DYNAMIC` and `D3DXMESH_IB_DYNAMIC`. This will create a mesh that can have both its vertex buffer and index buffer efficiently locked without stalling the pipeline.

D3DXMESH_SOFTWAREPROCESSING
Equivalent to specifying both `D3DXMESH_VB_SOFTWAREPROCESSING` and `D3DXMESH_IB_SOFTWAREPROCESSING`. Used to specify when using a mixed mode device that the mesh will be rendered in software vertex processing mode.

We have not covered all of the flags that can be specified as the second parameter to the D3DXLoadMeshFromX function. However you will usually find yourself using only one or two of the flags we have looked at here. Please consult the SDK documentation if you would like more information on other flag types.

LPDIRECT3DDEVICE9 pDevice
This is a pointer to the device interface. Mesh objects are always owned by the device object because the vertex and index buffers themselves are owned by the device object.

LPD3DXBUFFER * ppAdjacency
This is the address of an ID3DXBuffer pointer. It should *not* point to an already valid buffer interface since the function creates the buffer object when called. If the function is successful and the mesh is successfully created, this pointer will point to a valid ID3DXBuffer interface that contains the face adjacency information for the mesh. There will be three DWORDs in this buffer for every triangle in the mesh. If your application does not require the face adjacency information, you can simply release this buffer.

LPD3DXBUFFER *ppMaterials
This buffer represents an array of D3DXMATERIAL structures. The D3DXMATERIAL structure is a superset of the D3DMATERIAL9 structure and contains both a D3DMATERIAL9 and a texture filename.

```
typedef struct D3DXMATERIAL
{
      D3DMATERIAL9 MatD3D;
      LPSTR pTextureFilename;
} D3DXMATERIAL;
```

There is one structure in the array for each mesh subset. Thus, if the numbers in the attribute buffer range from 0 to 5 (indicating six subsets) then this buffer will store six D3DXMATERIAL structures. The subset ID itself is the index of the D3DXMATERIAL in the array.

The D3DXMATERIAL structure describes a texture/material pair. This information is stored in the X file and describes the texture and material information for each subset. For example, before calling DrawSubset and passing a value of 0 for the first subset of the mesh, we first set the material and texture described in the first element of the D3DXMATERIAL array.

Note that this information will not be maintained by the mesh object after the call. The application must store the material and texture information in its own arrays for correct subset rendering. Also note that while the material information is stored in the X file, only the texture filename is returned. It is the application's responsibility to load the texture.

While the term 'material' is actually quite commonly used in 3D graphics programming circles to describe all of the rendering attributes for a surface, it can be somewhat confusing given the limited definition of a material that we have been using since studying lighting in Chapter Five. The D3DXMATERIAL structure might have been more appropriately named something like D3DXATTRIBUTE or D3DXSUBSETATTRIBUTE to more accurately describe what each element in the material buffer holds, and to minimize confusion. As it stands, just try to keep this in mind moving forward. We can see that because each structure holds the texture and material used by a given subset, two separate subsets that have unique material information but share the same texture information will be represented by two unique D3DXMATERIAL structures in the buffer. Therefore, as multiple D3DXMATERIAL structures may reference the same texture, we must make sure that we do not load the texture multiple times.

Something else which is worth noting is that the X file format supports vertices with multiple sets of texture coordinates but supports only a single texture per face. That is, there is no explicit multi-texture support in the default X file format (although you can create custom chunks to support this feature).

Another small note about the X file format is that it does not store a material ambient property. Although the D3DXMATERIAL structure encapsulates a D3DMATERIAL9 structure (which does indeed support an ambient color), it will always be set to 0 for all materials extracted from the X file. This can create some degree of confusion if you are not aware of this limitation, so it might be wise to set the ambient color of the material to (1,1,1,1) when copying the material data out of the buffer. This will allow for controlling the ambient setting with the `D3DRS_AMBIENT` render state.

DWORD *pNumMaterials
This is the address of a DWORD which will contain the number of materials in the D3DXMATERIAL buffer. Consequently, this value also describes the number of subsets in the mesh and informs us that the attribute buffer will contain Attribute IDs in the range [0, NumMaterials-1].

LPD3DXBUFFER* ppEffectInstances
In Module III we will study Effects, which are essentially a high level abstraction that encapsulates rendering concepts like texture states, transforms states, render states, and more. This buffer is an array of D3DXEFFECTINSTANCE structures (one for each subset). Each structure contains the filename of the effect file that should be used for rendering the subset and an array of default values that can be passed into the effect file code. This allows an artist or modeler to store the default parameters for an effect in the X file. It should be noted that the X file does not store the actual effect file. Effect files will need to be loaded by the application as was the case with textures stored in the material buffer.

LPD3DXMESH* ppMesh
The final parameter is the address of a pointer to an ID3DXMesh interface. If the function is successful this will point to a valid mesh interface when the function returns.

The following code snippet demonstrates how we might use this function to load an X file into an ID3DXMesh. In this example it is assumed that the CScene class has an array allocated to store D3DMATERIAL9 structures and an array of TEXTURE_ITEM structures to store the textures and their filenames. We use a structure rather than just storing the texture interface pointers so that we can store the texture filename to make sure we do not load it more than once.

```
typedef struct _TEXTURE_ITEM
{
    LPSTR               FileName;   // File used to create the texture
    LPDIRECT3DTEXTURE9  Texture;    // The texture pointer itself.
} TEXTURE_ITEM;
```

The following code loads the X file and immediately releases the returned adjacency and effect instance buffers because they are not needed in this example. After that, we get a pointer to the returned D3DXMATERIAL buffer and start copying the material information to our CScene class D3DMATERIAL9 array and load the textures and into the CScene::m_pTextureList array.

```
ID3DXBuffer        *pAdjacency;
ID3DXBuffer        *pSubSetAttributes;
ID3DXBuffer        *pEffectInstances;
ID3DXMesh          *pMesh;
DWORD               NumAttributes;

D3DXLoadMeshFromX("Cube.x", D3DXMESH_MANAGED, pDevice , &pAdjacency,
                  &pSubSetAttributes, &pEffectInstances, &NumAttributes, &pMesh);

// We don't need this info in this example so release it
pAdjacency->Release();
pEffectInstances->Release();

// Lets get a pointer to the data in the material buffer
D3DXMATERIAL *pXfileMats = (D3DXMATERIAL*) pSubSetAttributes->GetBufferPointer();
```

At this point we can use the pointer to the D3DXMATERIAL data to step through the material buffer and process the information. The first step copies the D3DMATERIAL9 information into the CScene material array. Since an X file does not contain ambient information, we set the ambient property to opaque white after the copy.

```
// Loop through and store all the materials and load and store all the textures
for ( DWORD  i = 0 ; i < NumAttributes ; i++ )
{
    // first copy the material data into the scenes material array
    pScene->m_Materials[i] = pXFileMats[i].MatD3D;
    pScene->m_Materials[i].Ambient = D3DXCOLOR ( 1.0f , 1.0f , 1.0f , 1.0f );
```

Now we will process the texture for the current attribute. We will check to see whether any of the previous textures we have loaded share the same filename and if so, store a copy of the pointer in the current TEXTURE_ITEM element and increase the texture's reference count.

```
    // Used to track the whether a texture has already been loaded.
    BOOL bTextureFound = FALSE;

   // Next we need to check if the texture filename already exists in our
   // scene. If so, we don't want to load it again. We will just store its
   // pointer in the new array slot.

   for ( ULONG q = 0; q < pScene->m_TextureCount; ++q)
   {
```

```
        if( _tcsicmp( pScene->m_pTextureList[q]->FileName,
                  pXFileMats[i].pTextureFilename ) == 0 )
        {
                pScene->m_pTextureList[i]->Texture=
                                        pScene->m_pTextureList[q]->Texture;

                pScene->m_pTextureList[i]->Filename=
                           strdup(pScene->m_pTextureList[q]->Filename);

                pScene->m_pTextureList[i]->AddRef();

                bTextureFound = TRUE:
        }

    } // Next Texture
```

If the loop ends and bTextureFound is still false, then this is a new texture and must be loaded and the filename stored for future comparisons.

```
     // If the texture did not already exist load it into our array
     if( !bTextureFound )
     {
        D3DXCreateTextureFromFile(pScene->m_pD3DDevice,
                                  pXFileMats[i].pTextureFilename,
                                  &pScene->m_pTextureList[i]->Texture);

        pScene->m_pFilename = strdup(  pXFileMats[i].pTextureFilename );
     }


     // a new texture has been added to our CScene TEXTURE_ITEM array.
     pScene->m_TextureCount++;

} // next material in buffer
```

The code above requires some tighter error checking (for example, a material may not have a valid texture filename) but it gives us a good idea of the basic steps.

At this point, each face in our mesh has an Attribute ID assigned and stored in the attribute buffer. This ID describes the index of the material and texture it should be rendered with. With this in mind, we can render our mesh like so:

```
pDevice->SetFVF( pMesh->GetFVF() );

// render our mesh
for ( DWORD i = 0; i < NumAttributes; i++ )
{
      pDevice->SetMaterial ( &pScene->m_Materials[i] );
      pDevice->SetTexture  ( 0 , pScene->m_pTextureList[i].Texture );
      pMesh->DrawSubset    ( i );
}
```

8.6.2 D3DXLoadMeshFromXInMemory

We can also load the X file data in its entirety into a block of memory and instantiate a mesh object using this stored data. The block of memory should be an exact representation of the X file (headers, etc.). This can be done by inquiring about the size of the file, allocating a block of memory to accommodate it, and then reading every byte of the file into the memory buffer. Once the file is in memory, you can use the following function to create an ID3DXMesh:

```
HRESULT D3DXLoadMeshFromXInMemory
(
    LPCVOID Memory,
    DWORD SizeOfMemory,
    DWORD Options,
    LPDIRECT3DDEVICE9 pDevice,
    LPD3DXBUFFER *ppAdjacency,
    LPD3DXBUFFER* ppMaterials,
    LPD3DXBUFFER* ppEffectInstances,
    DWORD *pNumMaterials,
    LPD3DXMESH *ppMesh
);
```

This function is analogous to the texture loading function D3DXCreateTextureFromFileInMemory (see Chapter Six) in that rather than loading the X file from a file on the hard disk, it loads it from data that is stored in memory. With the exception of the first two parameters, all other parameters are identical to the D3DXLoadMeshFromX function.

LPCVOID Memory
This is a pointer to the first byte in the block of memory containing the X file data that has been loaded.

DWORD SizeOfMemory
This parameter describes the size of the X file data memory block in bytes.

8.6.3 D3DXLoadMeshFromXResource

Sometimes storing models as resources -- which are compiled either inside the application module or an external compiled module -- is preferred since it prevents users from gaining access to your models outside of the application. For smaller applications this is fine, but we will look at better alternatives for keeping our data safe in Module III.

The last seven parameters are identical to those described in the D3DXLoadMeshFromX function. The first three parameters are used to pass information about the module the X file resource is stored in, the name of the resource within the module, and the type of resource that it is.

```
HRESULT D3DXLoadMeshFromXResource
(
    HMODULE Module,
    LPCTSTR Name,
    LPCTSTR Type,
    DWORD Options,
    LPDIRECT3DDEVICE9 pDevice,
    LPD3DXBUFFER *ppAdjacency,
    LPD3DXBUFFER *ppMaterials,
    LPD3DXBUFFER* ppEffectInstances,
    DWORD *pNumMaterials,
    LPD3DXMESH *ppMesh
);
```

HMODULE Module
This is the handle of an already loaded module that contains the resource.

LPCTSTR Name
This is the name that has been given to the X file resource in the module. It is used to extract the correct resource from the specified module.

LPCTSTR Type
This is a string describing the resource type.

8.6.4 D3DXLoadMeshFromXof

```
HRESULT D3DXLoadMeshFromXof
(
    LPDIRECTXFILEDATA pXofObjMesh,
    DWORD Options,
    LPDIRECT3DDEVICE9 pD3DDevice,
    LPD3DXBUFFER *ppAdjacency,
    LPD3DXBUFFER *ppMaterials,
    LPD3DXBUFFER* ppEffectInstances,
    WORD* pNumMaterials,
    LPD3DXMESH *ppMesh
);
```

This function is used to work with lower level X file parsing routines. While DirectX Graphics now provides most of the high level X file loading and creation functions that we should need, many of these functions were not available in earlier versions of DirectX. Even DirectX 8 had no mechanism to automatically load mesh hierarchies (Chapter Nine), so X files had to be loaded at a lower level. Even now, it is possible that you may still need to resort to lower level X file loading -- if you intend to place customized template types in your X files, then this is certainly the case.

From the very early days of DirectX, a set of interfaces has been available to parse X files. Although a complete discussion of these low level interfaces is beyond the scope of this chapter, there is some

coverage included in the next chapter and in the DX9 SDK documentation itself (check the section called 'The X File Reference'). For now, we will briefly explain the basics of how these interfaces work.

The first thing we need to do to load data manually from an X file is call the function DirectXFileCreate. This will create a DirectXFile object and return a pointer to an IDirectXFile interface.

```
IDirectXFile *pXfile;
DirectXFileCreate ( &pXfile );
```

The IDirectXFile interface exposes three methods:

- CreateEnumObject - used for enumerating the memory chunks in the X file (meshes, materials matrices, etc.)

- CreateSaveObject - creates an interface that allows you to save data out to an X file (with custom templates if desired) \

- RegisterTemplates - used to register your own custom templates so that they are recognized by both the enumerator and save objects.

Normally the first thing we would do after creating the interface is register the standard X file templates. The following function call makes sure that the X file interfaces understand all of the common templates found in an X file.

```
pXfile->RegisterTemplates((VOID*) D3DRM_XTEMPLATES, D3DRM_XTEMPLATE_BYTES);
```

We now call the CreateEnumObject function to create an enumeration object for a given file. We will get back an IDirectXFileEnumObject interface, which we can use to step through the file contents.

```
IDirectXFileEnumObject *pEnumObject;
pXfile->CreateEnumObject( "MyXfile.x" , DXFILELOAD_FROMFILE ,&pEnumObject );
```

The first parameter is actually dependant on the second parameter. In this case, because we are specifying `DXFILELOAD_FROMFILE`, the first parameter should be the X file name. The third parameter is the address of an IDirectXFileEnumObject interface that will point to a valid enumeration object on function return.

Once we have the enumeration object, we can use it to step through the *top-level* data chunks in the X file. A top-level object in an X file is an object which has no parent, but may or may not have multiple child objects. (The concept of parent/child hierarchies will be discussed in detail in the next chapter).

We use the IDirectXFileEnumObject::GetNextDataObject function to fetch the next top-level data object from the X file. It will return a pointer to an IDirectXFileData interface representing the retrieved data object. This interface can then be used to inquire about the object's properties. The following code shows how we can retrieve top-level objects in the mesh file using a while loop.

```
while (SUCCEEDED(pEnumObject->GetNextDataObject(&pFileData)))
{
    pFileData->GetType(&pGuid);

    if (*pGuid==TID_D3DRMMesh) ParseMesh ( pFileData );

    pFileData->Release();
}
```

Once we have the data object, we can call the IDirectXFileData::GetType function to return the GUID for the object type. The DirectX SDK provides a list of all object types and their associated GUIDs so that we can use them to make decisions about how and what to parse. In the simple example above, we are ignoring all top-level objects except meshes. Once we locate a mesh, we can extract the data from it using the D3DXLoadMeshFromXof as shown below.

> **Note**: If we load a mesh hierarchy manually using this approach, then we will also want to also process the TIF_D3DTMFrame top-level object and any child objects. Frame objects can contain additional child meshes and matrices describing parent-child spatial relationships. Processing a frame object is a recursive concept where we test for children and create child meshes as needed. These are stored along with their respective matrices. In our example above, child meshes and frames would be ignored and only top-level meshes would be created. In the next chapter, we will discuss scene hierarchies.

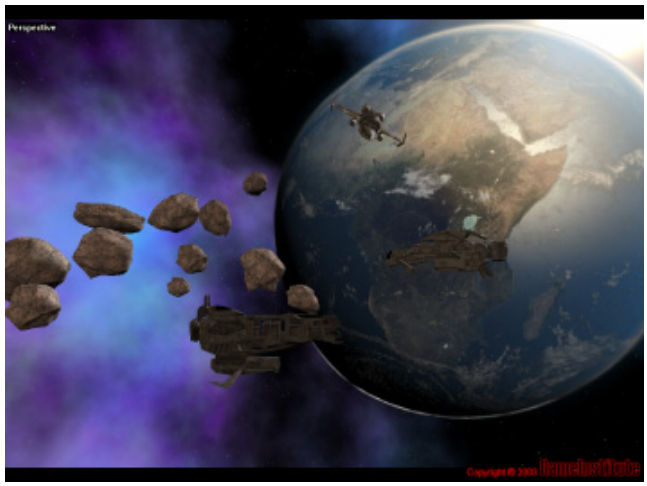
```
void ParseMesh ( IDirectXFileData * pFileData )
{
    ID3DXBuffer * pAdjacency;
    ID3DXBuffer * pMaterials;
    ID3DXBuffer * pEffects;

    D3DXLoadMeshFromXof ( pFileData , D3DXMESH_MANAGED , pDevice ,
                          pAdjacency, pglbMaterials[glbMeshCount],pEffects,
                         &pglbNumMaterials[gblMeshCount],
                         &pMeshArray[glbMeshCount] );

    glbMeshCount++;
    pAdjacency->Release();
    pEffects->Release();

}
```

The above function assumes that pglbMaterials is a global array of ID3DXBuffer interface pointers. For each mesh extracted, we can store its buffer in an array for later processing. The number of materials is stored in a global and this tells us how many materials are in the corresponding materials buffer. The returned ID3DXMesh interface is also stored in a global array. The adjacency and effect buffers are simply released as we are not using them in this example.

> **Note:** D3DXLoadMeshFromX, D3DXLoadMeshFromXInMemory, D3DXLoadMeshFromXof and D3DXLoadMeshFromResource do not support the loading of multiple meshes and arranging them in a hierarchical fashion. It is often the case (as we will see in the next chapter) that an X file may contain a number of mesh objects which may or may not be arranged hierarchically. When an X file is loaded using one of these functions, a single mesh is returned. If the file contained multiple meshes, they will be collapsed into a single mesh. We still get the full geometry set, but we lose the ability to move each sub-mesh independently. If the file contains a mesh hierarchy, such that a mesh can have child meshes and transformation matrices, this function will transform the child meshes using the matrices in the X file

before collapsing it into the returned mesh. This ensures that meshes that are defined relative to parent meshes in the X file correctly collapse into a single mesh with one coordinate space. In Chapter Nine we will examine how to load mesh hierarchies.

8.7 Mesh Creation



Often we will need to create mesh objects manually for the purpose of filling them with data from another file format (IWF for example). In Lab Project 8.1, we will create a simple mesh viewing application that supports X and IWF file loading. Some of our scene meshes will be loaded from X files using the D3DXLoadMeshFromX function and others will be created manually and populated with mesh data from an IWF file.

Creating an empty mesh is done with a call to either the D3DXCreateMeshFVF or D3DXCreateMesh functions.

D3DXCreateMesh allows us to specify a mesh vertex format using something called a *declarator*. Vertex declarators and the D3DXCreateMesh function will be discussed in Module III of this series when we examine vertex shaders. For the rest of this course, we will use the D3DXCreateMeshFVF version of the function. D3DXCreateMeshFVF allows us to describe the desired vertex format using the more familiar flexible vertex format flags.

```
HRESULT D3DXCreateMeshFVF
(
    DWORD NumFaces,
    DWORD NumVertices,
    DWORD Options,
    DWORD FVF,
    LPDIRECT3DDEVICE9 pDevice,
    LPD3DXMESH *ppMesh
);
```

DWORD NumFaces
This parameter tells the function how many triangles the mesh will have. This value will be used to allocate the mesh index buffer and its attribute buffer to ensure enough space to accommodate this many triangles.

DWORD NumVertices
This parameter tells the function how many vertices will be stored in the vertex buffer. The function uses this value to allocate a vertex buffer large enough to hold only this number of vertices.

DWORD Options
The options flag has the same meaning we saw in the D3DXLoadMesh… functions. It informs the function about memory resource pools, static vs. dynamic properties and/or write-only status. Unlike D3DXLoadMeshFromX and its sister functions, you can use the `D3DXMESH_VB_WRITEONLY`, `D3DXMESH_IB_WRITENONLY` and `D3DXMESH_WRITEONLY` flags and the function will not fail. As mentioned previously however, if you do specify any of these flags, then any subsequent calls to optimize the mesh or generate face adjacency information will fail. Thus, only use the write-only flags if you do not plan on touching the buffers that have been created.

DWORD FVF
Since we are creating an empty mesh, we can choose the vertex format we wish the mesh to use. This flag will be used in conjunction with the NumVertices parameter to allocate the appropriate amount of vertex buffer memory. These FVF flags are no different than the FVF flags we have been using throughout this series when creating a vertex buffer.

LPDIRECT3DDEVICE9 pDevice
Because the device object will own the vertex and index buffer memory, we must pass in a pointer to the device interface for mesh creation.

LPD3DXMESH *ppMesh
The final parameter is the address of an ID3DXMesh interface pointer. On successful return, it will point to the newly created mesh object interface.

The following code demonstrates creating an empty mesh object with enough space in its vertex buffer for 24 untransformed and unlit vertices. The index buffer is large enough to contain 12 triangles (12*3 = 36 indices) and an attribute buffer large enough to hold 12 DWORDs -- one for each face.

```
ID3DXMesh *pMyMesh;
DWORD fvfFlags = D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1
D3DXCreateMeshFVF (24, 12, D3DXMESH_MANAGED, fvfFlags, pDevice, &pMesh);
```

At this point we have an empty mesh and we can begin filling it with our required information. This can be done by locking the buffers and copying the vertex, index, and attribute information into them. Check the source code and workbook for Lab Project 8.1 to see how this was done using IWF data exported from GILES™.

8.8 Mesh Cloning

Cloning is basically the process of duplicating mesh data. However, since we can also specify a desired vertex format, cloning is actually a bit more sophisticated than a simple copy operation. Every mesh type that is derived from ID3DXBaseMesh inherits the Clone and CloneFVF functions. Whenever we call a clone function, a new mesh will be generated and the original mesh data will be copied into the new mesh (although vertex buffer sharing is supported as well). The old mesh remains intact and can continue to be used normally.

Before we discuss some important uses for mesh cloning, let us first take a look at the ID3DXBaseMesh::CloneFVF function to see how we can perform a cloning operation. It should be noted that the ID3DXBaseMesh::Clone function (without the FVF) works with declarators rather than FVF flags. We will discuss vertex declarators in Module III of this series.

```
HRESULT ID3DXBaseMesh::CloneMeshFVF
(
    DWORD Options,
    DWORD FVF,
    LPDIRECT3DDEVICE9 pDevice,
    LPD3DXMESH *ppCloneMesh
);
```

DWORD Options
These flags will be a combination or zero or more members from the D3DXMESH enumerated type that will affect the way the cloned mesh is created. We discussed the various members of this enumerated type when we examined D3DXLoadMeshFromX and their meaning is the same. This means that the cloned mesh need not have its vertex buffers allocated in the same resource pools as the original mesh. It may also have dynamic buffers where the original did not. For example, we could create a mesh in the system memory pool, optimize it, and then clone it out to a mesh in the default pool with the `D3DXMESH_WRITEONLY` flag. This would create a video memory based optimized mesh for fast rendering. This is in fact a very common use of cloning.

One member of the D3DXMESH enumerated type is only used when cloning: `D3DXMESH_VB_SHARE`. With this flag, the cloned mesh will share the vertex buffer with the original mesh and the result is a memory savings. This can be very useful for proprietary LOD algorithms or even just simple mesh instancing. Modifying the original vertex buffer will affect all meshes cloned in this way. The default behavior is to create a cloned mesh with its own vertex, index, and attribute buffers.

DWORD FVF
This value allows us to pass FVF flags for the cloned mesh vertices. The result is that we can clone a mesh into a different vertex format. This can be very useful when we load X file meshes and have no direct control over the FVF format stored in the file. By cloning the mesh into a copy that uses a new vertex format, we can create additional room for texture coordinates or vertex normals that may not have been stored in the original X file. Once we have a cloned copy of the mesh, we could release the original mesh and make the clone the final mesh that we use for rendering.

LPDIRECT3DDEVICE9 pDevice
A pointer to the device that will own the mesh resources.

LPD3DXMESH *ppCloneMesh
Provided the function is successful, this will point to a valid interface for the newly cloned mesh.

Cloning Advantages

1. *Removal of unnecessary or extraneous vertex components.* It is easy to strip off vertex components that we do not intend to support in our engine. After loading a mesh (from a file for example) we simply specify the desired FVF and clone it. The new mesh will have only the required data and we can safely release the original mesh. As an example, imagine we load a mesh from an X file which contains vertex normals, but our engine has no use for them (perhaps it is not using the lighting pipeline). We could clone a new mesh which does not have vertex normals and then release the original. Our new mesh is now a streamlined version of the original X file representation containing only the data our application requires.

In the following code, pMesh is a pointer to a mesh that was created using the D3DXLoadMeshFromX function and it contains all of the components loaded from the X file. Assume that in this particular case, the mesh includes three sets of texture coordinates, two of which we do not need. We will create a clone that contains only the vertex components we are interested in.

```
LPD3DXMESH NewMesh;
pMesh->CloneMeshFVF(D3DXMESH_MANAGED, D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1,
                    m_pD3DDevice, &NewMesh);
pMesh->Release();
```

2. *Adding missing vertex components.* We can also add vertex components that may not have been present in the original mesh. For example, if the mesh did not include vertex normals, we can add them by cloning with `D3DFVF_NORMAL` included in the flag. Note that while the clone will now have space for the required vertex components, the data itself will still need to be initialized. The cloning functions do not automatically generate the actual values for these added components. They can only make space for them in the vertices so that they can be later initialized.

In the next example, if the original mesh does not contain vertex normals, then we clone a version that does. Once we have a clone that includes space for vertex normals, we call D3DXComputeNormals (another very useful global D3DX function) to compute the vertex normals for the clone. The code assumes that m_pMesh is a pointer to a mesh that was loaded using D3DXLoadMeshFromX.

```
if(!m_pMesh->GetFVF() & D3DFVF_NORMAL)
{
    LPD3DXMESH NewMesh;

    // Introduce a normal component into our VB
    m_pMesh->CloneMeshFVF(m_pMesh->GetOptions(),
                          m_pMesh->GetFVF() | D3DFVF_NORMAL, m_pD3DDevice, &NewMesh);

    // Generate some new vertex normals
    D3DXComputeNormals( NewMesh, NULL );
```

```
    // Release old mesh and store
    m_pMesh->Release();
    m_pMesh = NewMesh;

} // End if no normals
```

3. *Building a render-ready mesh.* In certain situations, mesh cloning becomes a requirement. In a few cases, such as when working with the utility classes (ID3DXSPMesh/ID3DXPatchMesh), the clone functions exposed by these interfaces are the only way to get the data back out to a render-ready format -- an ID3DXMesh.

4. *Maximum benefit from optimization functions*. Write-only buffers make rendering more efficient, but in order for mesh optimization functions to succeed, they need read access to the buffer data. Write-only vertex buffers will cause these routines to fail. In order to achieve both ends, a common practice is to load the mesh into system memory and perform the geometry optimizations. Since the optimization process is performed by the host CPU and not the graphics card, they will be performed more rapidly on a system memory mesh anyway. Once the mesh has been optimized, we will clone the system memory mesh to a default pool or managed pool mesh. This is where we can specify the write-only flag for the clone's vertex buffer. At this point we can either release the system memory copy or store it for later use (perhaps to re-clone on device loss if using the default pool for the clone).

The following code loads a mesh into system memory and performs the optimization step. The optimized mesh is then cloned out to the default pool with the write-only flag set for both buffers.

```
ID3DXBuffer *pAdj ;
ID3DXBuffer *pMat;
ID3DXBuffer *pEff;
ID3DXMesh *pSysCopyMesh;
ID3DXMesh *pRenderCopyMesh;

// Load the X file into a system memory mesh
D3DXLoadMeshFromX("Cube.x" , D3DXMESH_SYSTEMMEM , pDevice , &pAdj , &pMat,
                  &pEff , &NumMaterials , &pSysCopyMesh );

// Release the effects buffer, we will not use it in this example
pEff->Release();
pEff = NULL;

DWORD *pAdjacency = pAdj->GetBufferPointer();

// Optimize the mesh and pass in the adjacency array
pSysCopyMesh->OptimizeInPlace(D3DXMESHOPT_VERTEXCACHE, pAdjacency, NULL, NULL, NULL);

// Optimization has been done se we no longer need the adjacency info
pAdj->Release();
pAdj = NULL;
pAdjacency = NULL;

// Clone the mesh out to a default pool mesh with write only vertex and index buffers
pSysCopyMesh->CloneFVF(D3DXMESH_WRITEONLY, pSysCopyMesh->GetFVF(),
                       pDevice, &pRenderCopyMesh);

// Release the system memory copy
pSysCopyMesh->Release();
```

To conclude our discussion of the ID3DXMesh interface, there is one more function inherited by all mesh types from the ID3DXBaseMesh interface which we should mention. It is called UpdateSemantics and it is loosely related to the Clone function in that it allows you to alter the format of a mesh vertex buffer without cloning it. However there is a limitation imposed that does not allow us to change to a format that would cause the vertex buffer to change in size. So you might use this function to change the vertex format such that a mesh uses a vertex with a specular color instead of a diffuse color. In that case we are simply changing the meaning of the DWORD stored in each vertex (a semantic difference).

> **Note:** ID3DXBaseMesh::UpdateSemantics function only works with vertex declarators (not FVF flags) so we will examine it in Module III of this series.

8.9 Progressive Meshes

In any given scene, it is probably fair to say that a number of objects will be situated at a distance from the viewer such that their full detail is no longer appreciable. Consider a model of a car that is constructed using 3000 polygons. If the car is far off in the distance, it might only take up a handful of screen space pixels when rendered. In that case the user will no longer be able to clearly see that the car has four doors, or that there is a driver sitting inside the car. Logos or other forms of writing on the car are certainly no longer able to be clearly viewed. In such a case, rendering only a handful of polygons for the car would not noticeably degrade the image presented to the viewer, given the already reduced level of detail present in the few pixels being drawn. Transforming, lighting, and rasterizing 3000 polygons under these circumstances would be inefficient to say the least. Reducing the polygon count of the object provides a significant savings with respect to both GPU processing and bandwidth.

Ideally what we would like to do is perform this process on the fly. As objects move closer to the camera, the polygon count would increase and more model detail would be visible. As objects move further away, the polygon count would be reduced. We would also prefer this detail reduction to take place in such a way that the model does not become noticeably corrupt. That is, the selection of which polygons to remove should be based on a heuristic that understands which ones contribute more than others. Certainly we want to be careful about removing polygons that would clearly distort the overall shape of the object.

What we require is a way to progressively refine the *level of detail* (LOD) of a mesh at runtime so that detail is removed when no longer needed and re-introduced when that detail is once again required.

> **Note:** While LOD is often discussed in the context of geometric manipulation, it applies to other concepts as well where we might need less detail at one time and more at another. For example, we could consider mip-mapping to be an LOD technique for textures. Animations might also be modified to be more or less complex based on factors such as view distance.

Much research in the field of mesh LOD has taken place and there are a variety of different algorithms that have become essential tools in game development. Algorithms that are based on camera distance and sometimes even orientation are called **view dependant**. View Dependant Progressive Meshes (VDPM) is the embodiment of that technique. Other algorithms are not tied directly to the camera, but instead tessellate or simplify geometry based on other criteria. These are called **view independent** algorithms and are implemented as View Independent Progressive Meshes (VIPM).

While VIPM methods do not downgrade/upgrade a mesh based on distance to the viewer per se, that is not to suggest that they cannot be used in that way. When using VIPM methods, we typically pass in a target vertex or face count for the model. This target can certainly be calculated by taking into account camera distance. Thus VIPM is more flexible in this respect and we can progressively alter mesh LOD based on whatever heuristics we choose (model distance from the camera, the capabilities of the end user's machine, a geometry detail setting that the user may choose from a menu, etc.).

VIPM techniques can be especially useful for console development where there is typically a much smaller amount of memory available to store polygons and textures. It can also be useful to achieve a desired frame rate. If the frame rate of the application drops below a certain threshold, meshes can be

simplified to lower resolutions to help achieve a more fluid gaming experience. If the frame rate is very high, more detail can be added back. In the next section we will take a brief look at the VIPM algorithm used by the D3DXPMesh and D3DXSPMesh objects to perform mesh simplification. Please note that you do not have to be aware of how VIPM works in order to use progressive mesh support in D3DX, but many students may find what is going on under the hood interesting. This will also allow you to more intelligently use the mesh interfaces in an efficient way.

8.9.1 View Independent Progressive Meshes (VIPM)

Progressive mesh support was introduced in DirectX 8.0 as part of the D3DX library. The progressive mesh support in D3DX uses a VIPM algorithm to increase/decrease the face count or vertex count to a specified vertex or face count target through requests from the calling application.

When we first create a progressive mesh, a number of calculations are done behind the scenes to build lookup tables. These tables describe how the original mesh data we pass in should be downgraded over a series of different stages. Each stage will typically store the collapse information for two triangles (although this can vary), reducing the face count of the mesh by two. This triangle removal is set in motion by a vertex being removed (i.e. *collapsed* onto a neighboring vertex). This removes an edge from the model and in turn, the triangles that share that edge. As you can imagine, a complex mesh will have many collapse stages pre-calculated for it -- each stage performing one edge collapse, typically removing a single vertex and two triangles from the mesh.

The mesh data is not actually simplified at progressive mesh creation time, but what is stored is a series of edge collapse structures. These structures can be traversed at runtime to simplify the mesh via the requests made by the application (through the ID3DXPMesh interface). With this approach, we can downgrade the mesh one stage at a time to a very low level of detail. Because the simplification stages are pre-computed at mesh creation time and simply have to be traversed at runtime in response to a simplification request, this makes the runtime part of the simplification process more efficient.

> **Note:** This is a unidirectional simplification algorithm. It is not possible to tessellate the mesh beyond the original polygon count using VIPM. Polygon splitting is not a component of the algorithm, so the original model we start with is the highest LOD we can achieve.

When specifying a target face or vertex count, the LOD routine must be able to quickly determine which vertices to remove and which triangles are affected. As mentioned, the removal of a vertex will usually result in the removal of an edge, which in turn usually removes two triangles from the index buffer. Of course, other triangles that share that vertex may also be affected and they will need to have their indices remapped to point to a neighboring vertex instead. This is the *vertex collapsing* process.

In order to go from the highest level of detail down the desired face count, the progressive mesh will step through an ordered series of pre-calculated edge collapse structures (built at mesh creation time). Each structure stores information about which vertex must be removed next, which triangles need to be removed as a result of the collapsed edge, and which remaining triangles index the removed vertex and need to have their indices remapped to a neighboring vertex instead. This very same information also describes how the simplification can be undone. We can reverse an edge collapse (a process known as a

vertex split) to re-introduce vertices and edges that have been removed. Eventually this will bring the mesh back up to its highest LOD. Again, the highest level of detail will be the dataset that the progressive mesh was first created with. While we can remove detail from this dataset and re-introduce the detail we removed, we can never introduce more detail than was contained in the original model.

> **Note:** Progressive meshes cannot add detail to the original dataset used to create the progressive mesh. No surface subdivision is ever done to the input data set. Progressive meshes can only vary the detail of a model from its original polygon count down to a lower polygon count and back up again.

Calculating which vertex to remove each time we need to collapse an edge in the mesh can be a very time-intensive task. It is usually performed by calculating an error metric that determines how much a collapse alters the surface topology of the mesh. Since this can involve testing every vertex in the vertex buffer and finding the vertex that has the lowest error (and removing that vertex), we can rest assured that the least important vertices are removed from the mesh first.

Given the time consuming nature of this task, each collapse step will be pre-calculated when the progressive mesh is first created. In this manner, the progressive mesh knows in advance the exact order that vertices will need to be removed to achieve a certain level of detail. We will see later that when we create a progressive mesh, we can influence the error metric calculation and assign priorities to make some vertices more or less important than others. Lower priority vertices will be candidates for removal early on in the simplification chain. Because the simplification routine knows the exact order that vertices will need to be removed during runtime simplification, the mesh vertex buffer can be built using the exact order of removal. Vertices that will be removed first can be stored at the end of the vertex buffer and vertices to be removed later (or not at all) can be stored at the front of the buffer. This makes LOD changes and subsequent rendering considerably more efficient.

When a simplification step is executed at runtime by our application and a vertex is removed, the D3DX object need only decrement the number of vertices in the vertex buffer for the render call. This means that the vertices currently being used to render the mesh at its current level of detail will always be in a continuous block in the vertex buffer. This is especially important if the progressive mesh is going to perform well on a software vertex processing device. As we know from our discussions in Chapter Three, a linear block of vertices will be transformed (and possibly lit) by the software pipeline.

The same logic extends to the triangles. Since D3DX will pre-calculate the order in which triangles will be removed during edge collapses, the index buffer can also be ordered accordingly. Triangles that will be removed first will be placed at the end of the index buffer and triangles that will be removed later (or not at all) will be placed near the front of the index buffer. A single edge collapse structure will tell the D3DXMesh at render time how many triangles will be taken away. If multiplied by 3, D3DX will know how many indices to 'remove' from the index buffer. Again, this is really just a decrement of the total count.

Fig 8.6 depicts a single pre-calculated simplification step. We see the removal of an edge by collapsing vertex 8 onto vertex 4, which causes the blue and orange triangles to be removed from the mesh. In this example, vertex 8 was determined during mesh initialization to be the first candidate for removal because it would have the least impact on the mesh topology. Note its position at the end of the vertex buffer. The current index count is also decreased by six to account for the two triangles lost during the edge collapse. This effectively pops six indices off the end of the currently used portion of the index

buffer. Triangles that were not removed but that referenced vertex 8 are then remapped to point to vertex 4.

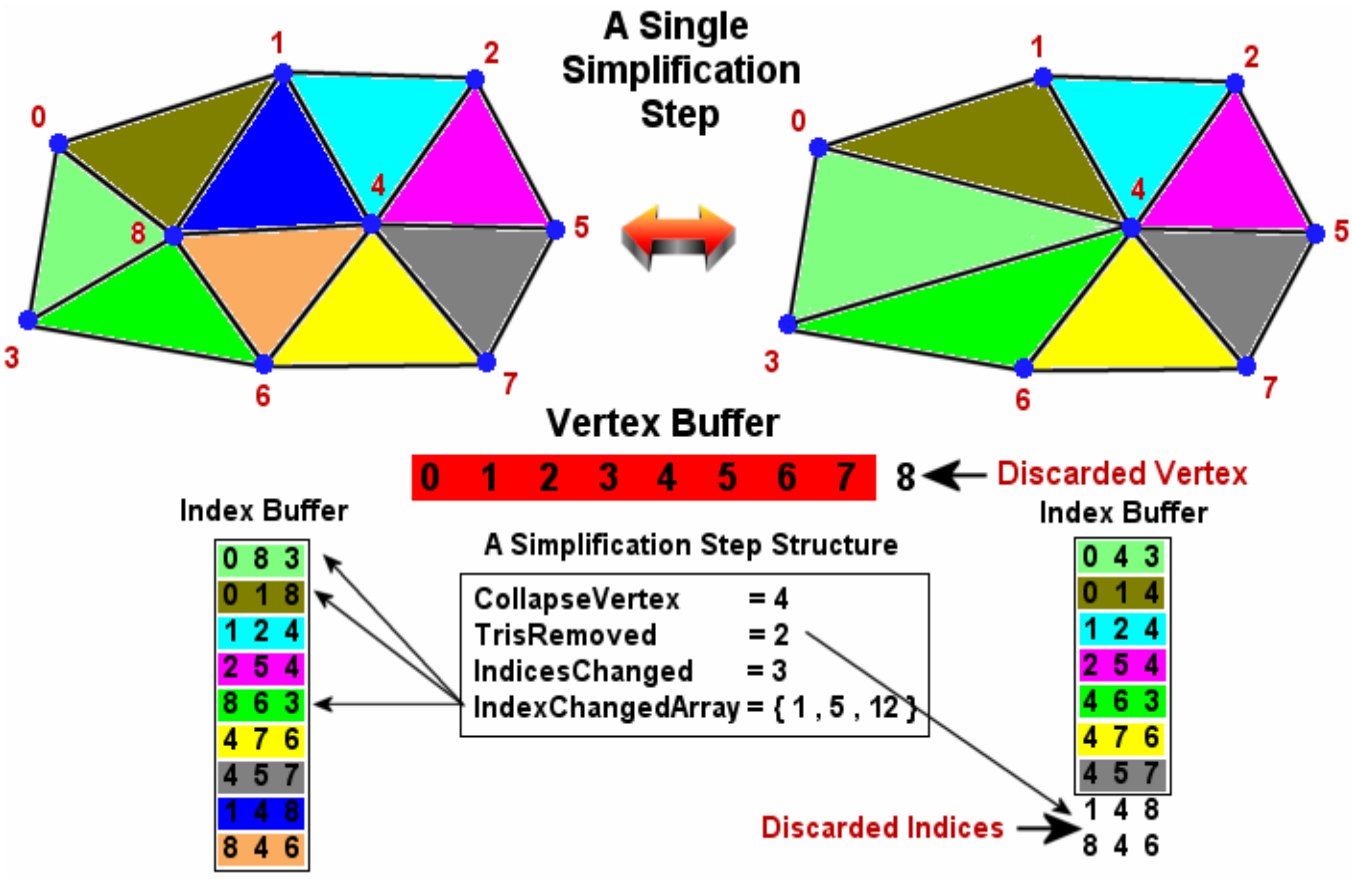


Figure 8.6

While the index remapping for the triangles that are not removed (yet still affected by the edge collapse) is done at runtime by the mesh object, each pre-computed simplification structure contains an array of indices that will need re-mapping when the simplification step is performed. In Fig 8.6 we can see that the IndexChangedArray contains the numbers of all indices in the index buffer that will need to be assigned the new vertex index because they reference the vertex which has just been removed (vertex 8). The six indices that are snipped off the end of the index buffer do not need to be remapped even if they reference the removed vertex because they are not going to be used until we undo the collapse (re-adding vertex 8 to the vertex buffer).

The following pseudo-code shows the use of the pre-compiled simplification step array (generated at mesh creation time) to simplify the mesh down to as close to the requested face count as possible. This is not necessarily the exact process carried out by the D3DX progressive mesh object mind you, but it does give us a theoretical understanding of VIPM.

```
if (CurrentFaceCount > RequestedFaceCount) // we need to process the next simplification step
{
        SimplificationStep *SimpStep = &SimplificationStepArray [CurrentStepIndex++];

        for ( I = 0 ; I < SimpStep->IndicesChanged;  I++)
        {
                pIndices[ SimpStep->IndexChangedArray[I] ] = SimpStep->CollapseVertex;
        }

        CurrentlyUsedVertices --;
        CurrentlyUsedIndices -= SimpStep->TrisRemoved * 3;
}
```

In this example the application has requested a target face count. If it is lower than the current face count then we need to process more of the pre-compiled simplification steps to further simplify the current mesh representation. If the mesh is currently set to the maximum level of detail then the first simplification step we process will be the first one in the pre-compiled array. This tells us the first vertex/edge to remove. Each simplification step tells us the indices that need to be changed to execute the current vertex removal in its IndexChangedArray. It also tells us the new values that these indices have to be set to in the structure's CollapseVertex member. Therefore, all we need to do is loop through each index in this array and change its value from the vertex about to be removed to the collapse vertex. That takes care of the triangles that will survive the collapse that reference the vertex about to be removed. Once done, we simply decrement the number of vertices we are using, effectively dropping the last vertex in the mesh, and decrease the current index count by multiplying the number of triangles dropped by three. Remember, we do not need to manually perform any of these steps, this is all hidden away inside the D3DXPMesh object. All we need to do is call a single function to set a new desired vertex or face count. The D3DXPMesh object will perform the simplification for us using a method similar to that described above.

A key point to understand is that we do not simply jump from one simplification step to another using some arbitrary position in the array. Instead we must process all of the simplification steps in between. If we currently have a face count of A and we want to simplify the mesh to face count D, we must process simplification steps A, B, C, and D to get the desired LOD. This is because a simplification step is computed relative to the one that came before it. This minimizes memory footprint for the simplification data structures.

Fig 8.7 shows that if the maximum face count of our mesh was 100, and we wanted to reduce the face count down to 90, the progressive mesh would need to carry out the first five simplification steps stored in the array. In this example we are assuming that each simplification step removes two triangles, but that may not always be the case with complex mesh topologies.

Figure 8.7

It should also be noted that this process will only be performed when the desired face count or vertex count has been modified. The resulting mesh would be used for rendering until such a time as a new target face or vertex count was specified. Thus, if we had already simplified the mesh down to 96 faces in our example, the internal simplification index would be set to 3 in Fig 8.7 because simplification steps 1 and 2 would have already been executed. If we then decided to simplify the mesh even further by another six faces, and passed in a face count of 90 (from the current 96) the simplification process would start at the current simplification level (3) and execute only simplification steps 3, 4, and 5.

Although it may not be obvious at first, the same information stored in the simplification structures we have been using to collapse edges also tell us everything we need to know to re-add that edge to the mesh and undo the simplification steps. Even at the lowest level of detail, nothing has actually been removed from the vertex or index buffers. The vertices and indices at the end of these buffers are just currently being ignored. Therefore, if we specify a desired face count that is higher than the current level of detail (but not higher than the maximum level of detail the progressive mesh was first created with), the mesh simplification routine can step backwards through the simplification array starting at the current position. Along the way it can restore the indices that were changed in the previous simplification step so that they once again point at the vertex that was removed. It can re-introduce the vertices and indices at the end of the currently unused portions of the vertex and index buffers by incrementing the counters, thereby adding the triangles that were removed in that step.

The following pseudo-code demonstrates how to reverse a simplification step and re-introduce previously deleted triangles and vertices until the desired face count is reached. It is essentially the exact opposite of the simplification code we just looked at.

```
if (CurrentFaceCount < Requested FaceCount && CurrentFaceCount < MaximumFaceCount)
{
        SimplificationStep *SimpStep = &SimplificationStepArray [ --CurrentStepIndex ];

        for ( I = 0 ; I < SimpStep->IndicesChanged; I++)
        {
                pIndices [ SimpStep->IndexChangedArray[I] ] = CurrentlyUsedVertices;
        }

        CurrentlyUsedVertices ++;
        CurrentlyUsedIndices += SimpStep->TrisRemoved * 3;
}
```

This code shows that if we need to re-introduce face detail, then we decrement the CurrentStepIndex and get a pointer to the last simplification step that was performed. The mesh has an array of indices that were changed and stored in each simplification step's IndexChangedArray. At first it might seem that we have no way of knowing which vertex those indices were re-mapped from when the collapse was performed. However, remember that for each simplification step we perform, the vertex removed is the one at the end of the currently used section of the vertex buffer. So when we are about to undo a simplification step, the next vertex just outside the currently used portion of the buffer *is* the vertex those indices were mapped to before that simplification step occurred. Therefore, we loop through each of the indices changed by that simplification step and reset their values to the CurrentlyUsedVertices number. If we currently have 10 vertices in use, then the 11^{th} vertex (i.e. Vertex[10]) will be the vertex we need to reset these indices to reference. Once we have remapped these indices to the original vertex, we simply increment the current vertex count by 1. This extends the size of the usable section of the vertex buffer to re-introduce the deleted vertex that these updated indices have now been reset to reference. Finally, we know that the triangles removed by the simplification step are still in the index buffer, but just outside the section currently being used. Therefore we increase the count of currently used indices to re-introduce the triangles that were removed.

While we do not need to know the details of VIPM and its various implementations to use the D3DX provided mesh class, it is certainly interesting material to study and may come in handy if you ever have to create your own simplification or LOD routines. This basic understanding of VIPM will also help us to properly use the D3DX supplied interface and give us some insight for decision making when it comes to issues of performance and efficiency.

> **Note:** Progressive mesh rendering, despite its potential for reduced polygon count, does not always outperform brute-force mesh rendering. This is because the algorithm requires that the vertex and index buffers be arranged in a manner such that the technique executes quickly. The buffer rearrangement will often be to the detriment of some of the optimization techniques discussed earlier (attribute sorting, vertex cache performance, etc.). Index buffer updates also hurt performance, so keep these ideas in mind as you experiment with the technique.

8.9.2 ID3DXPMesh

The ID3DXPMesh interface (derived from ID3DXBaseMesh) contains functionality that encapsulates dynamic mesh LOD. Based on our discussion in the last section, we now have a fairly good theoretical understanding of what this interface will do for us behind the scenes when we wish a mesh to be dynamically simplified.

Since its introduction in DirectX 8.0, the ID3DXPMesh has been based on the VIPM method introduced by Hughes Hoppe in a 1996 ACM SIGGRAPH paper. While the VIPM method used by the D3DX progressive mesh is not necessarily the most optimal reduction scheme, it is simple to implement, easy to use and is relatively hardware friendly. Unfortunately, it suffers from poor utilization of the vertex cache on hardware vertex processing devices given the vertex buffer ordering requirements we discussed in the last section.

It is also worth noting that the VIPM technique discussed requires the use of a dynamic index buffer to handle LOD changes. Because simplification is performed by the host CPU and involves a good deal of index buffer touching, the progressive mesh index buffer will typically be allocated in system memory. This allows the CPU to write to the buffer as quickly as possible. Although this means that we are rendering from a system memory index buffer, it is generally not as expensive as passing 2-byte indices over the bus (or 4 byte indices if 32-bit indices are used) given large sections of index data on a hardware vertex processing device. Fortunately the VIPM method used by the D3DX progressive mesh does not need to modify the vertex buffer contents once data has been properly ordered. Therefore, the vertex buffer will typically be created in video memory (local/non-local) by default on a hardware vertex processing device.

While ID3DXPMesh does introduce many additional functions beyond those inherited from ID3DXBaseMesh, most are simple ‘Get’ functions used for inquiring about the current settings or state of the mesh. Fortunately, there are surprisingly few methods that are required to use the LOD features.

8.9.3 Progressive Mesh Generation

D3DX provides no functions to load a progressive mesh from a file or even to create an empty progressive mesh. A progressive mesh is created via a standard ID3DXMesh which determines how the progressive mesh looks at its highest level of detail. D3DX exposes the D3DXGeneratePMesh function to convert a standard mesh to a progressive mesh. This provides the indirect means for creating a progressive mesh using data stored in an X file: we call D3DXLoadMeshFromX to create a normal ID3DXMesh and then feed it into the D3DXGeneratePMesh function to create a progressive mesh. Once the progressive mesh has been initialized, the ID3DXMesh interface is no longer needed and can be released. It is at this stage (when the progressive mesh is first created), that D3DX will analyze the topology of the mesh and will determine the order in which vertices and edges should be removed from the mesh. The progressive mesh will have its vertices and indices arranged in its index and vertex buffers so that triangles that will be removed first will have their vertices and indices placed at the rear of their respective buffers. After this step has been performed, the function will then pre-compute and

store all the edge collapse structures and arrange them in a list that can be traversed at runtime to perform efficient simplification in response to application requests. At this point, the function returns to the calling application a newly created progressive mesh interface ready to be simplified using a few simple ID3DXPMesh interface function calls.

```
HRESULT D3DXGeneratePMesh
(
    LPD3DXMESH pMesh,
    CONST DWORD *pAdjacency,
    CONST LPD3DXATTRIBUTEWEIGHTS pVertexAttributeWeights,
    CONST FLOAT *pVertexWeights,
    DWORD MinValue,
    DWORD Options,
    LPD3DXPMESH *ppPMesh
);
```

LPD3DXMESH pMesh

This is the input mesh from which the progressive mesh will be built. It defines the progressive mesh dataset at its highest possible level of detail. Note that this mesh is a standard ID3DXMesh which can be created manually using D3DXCreateMesh or loaded from an X file using D3DXLoadMeshFromX.

CONST DWORD *pAdjacency

Since pre-computing edge collapses relies on information describing how triangles and vertices are connected, this function requires us to pass in the adjacency information for the input mesh. This array will contain three DWORDs per-triangle and is used to compile a table of edge collapse information. The mesh will also reorganize its vertex and index buffers according to collapse order and will use this adjacency information to build a new adjacency buffer for the reorganized data. Unlike the D3DXMesh object which does not maintain adjacency information, progressive meshes will maintain their own internal face adjacency information. So once the mesh has been created, we can discard the input adjacency information of the input mesh along with the input mesh itself.

CONST LPD3DXATTRIBUTEWEIGHTS pVertexAttributeWeights

When the progressive mesh is first created, a pre-compiled simplification step lookup table will be built. It is used to order the data in the vertex and index buffers. To ensure that the mesh transitions as smoothly as possible between detail levels, each vertex receives an error value which describes how dramatic the change to the mesh topology will be if the vertex is removed. When calculating each simplification step, the vertex with the lowest error will be marked as the next to be collapsed. By default (or if we pass NULL as this parameter) the vertex position, normal, and blend weight contribute equally to the error produced for each vertex. For each vertex, a comparison is made between it and neighboring vertices to see which of the neighboring vertices is most similar to the current vertex being processed. This tells us which neighbor makes the best target for collapsing the vertex onto while keeping mesh distortion to a minimum. We would then compare various components of the two vertices and record the error value which is generated by summing the difference in the two vertex components. After doing this for each vertex, we will have a per-vertex error value describing the level of distortion that will be caused if this vertex is removed. At this point, the vertex list can be ordered such that vertices with lower error values are stored at the back of the vertex buffer and the triangles they remove pushed to the back of the index buffer.

Be default, the blend weight, vertex position, and vertex normal are used to build the error value for a vertex, and they are used in equal measure. By passing in a `D3DXATTRIBUTEWEIGHTS` structure, we can force the simplification step compiler to consider additional vertex components when generating the error for each vertex. If we pass in NULL as this parameter, a `D3DXATTRIBUTEWEIGHTS` structure with the following values is used (and is usually acceptable for most situations):

```
D3DXATTRIBUTEWEIGHTS AttributeWeights;

AttributeWeights.Position =  1.0;
AttributeWeights.Boundary =  1.0;
AttributeWeights.Normal   =  1.0;
AttributeWeights.Diffuse  =  0.0;
AttributeWeights.Specular =  0.0;
AttributeWeights.Tex[8]   =  {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
```

By default, the vertex colors and texture coordinates are not taken into account. Note that this can lead to distortion that is not topological but instead color based. If the texture coordinates or colors of a vertex that was collapsed onto another are very different, the visual results may certainly be noticeable.

This structure allows the application to change this default behavior by adding weights to additional vertex components or increase/decrease the weight of a default component such that it is considered more/less important in the comparison function and generates a larger/smaller error value, respectively. For example, the default values above show that the position, the boundary, and the normal of the vertex each contribute to the vertex error calculation equally. If we were to set the vertex normal to 2.0 instead of 1.0, we would double its importance and any error produced by differing vertex normals would be doubled. This would make vertices with different normals much less likely candidates for removal. If we were to set the normal member to 0.0, then when calculating the error of collapsing vertex A onto vertex B, the normals of vertex A and B would be ignored and would not influence the error calculation in any way.

The default values generally work pretty well most of the time and you will probably be able to pass NULL as this parameter and live with the results. However, if you discover that one of your simplified meshes is suffering severe texture distortion for example, you can weight the relevant texture coordinate set into the error calculation by using a larger value. Setting the structure's Tex[0] member to 4.0 for example, would quadruple the error caused by different texture coordinates. The result would be that two vertices with different texture coordinates are much less likely to be collapsed onto one another.

CONST FLOAT *pVertexWeights

This parameter is used to pass an array of per-vertex weights for the mesh. If NULL is passed, each vertex is assumed to have an equal weight of 1.0. Once the error for a vertex is calculated, its error is multiplied by its weight in this array, allowing the user to increase/decrease the importance of a particular vertex in a mesh.

Total Vertex Error = Vertex Error x Vertex Weight

Let us consider a mesh of a human face as an example. The nose vertices are likely to be very close together and thus may be removed from the mesh very early on in the simplification chain. Of course, the nose is a pretty important feature, so using this array we could set the weight of the nose vertices to a

larger value to ensure that they will be removed very late in the process (if at all). This allows us to retain the basic outline of the face even at a distance.

Another example might be rendering a section of terrain as a progressive mesh. As the terrain shifted to a lower LOD, a hill might disappear which currently has the mesh of a building sitting on top of it. When the hill polygons were simplified away, the building would be left floating in mid-air. Therefore, it is common practice in terrain engines that use such forms of dynamic LOD to be able to specify certain portions of the terrain mesh (such as those under buildings for example) as being non-removable.

If you do not pass NULL as this parameter, the array you pass in should hold a floating point weight value for every vertex in the input mesh. The higher the vertex weight, the less likely that vertex is to be removed. Passing NULL simply means that all vertices will be weighed equally (1.0).

DWORD MinValue

When the progressive mesh is generated, a number of simplification steps are pre-computed (one for each edge collapse). As you can imagine, a large number of edge collapse structures will be required to take a highly detailed mesh down to a handful of polygons. This value specifies a minimum number of vertices or faces (depending on the Options parameter to this function that will be discussed next) and simplification information will only be generated down to this level. This allows us to save memory and speed up the mesh creation process by instructing it not to compile and store information beyond a level of detail that we have no intention of using.

Once the mesh has been created and its simplification information pre-compiled, there is no way to simplify the mesh any lower than this number of faces or vertices. If you set the minimum number of faces to 100 for example, the edge collapse information would be generated at mesh creation time to take the mesh from its original polygon count to 100 faces and back up again. If during the render loop you called the ID3DXPMesh::SetNumFaces function and specified a value of 10, the edge collapse information for this LOD would not be available in the lookup table and the mesh would be simplified to its minimum number of faces (100 in this example).

It is worth noting that this value is not always achievable, since the mesh distortion caused by simplifying down to the requested minimum might be too extreme. However, if we do specify a minimum number that cannot be reached, the collapse information will be generated down to the lowest possible level of detail. Thus, passing zero as this parameter will always guarantee that we can simplify the mesh down to the lowest resolution considered possible by the progressive mesh object.

DWORD Options

The ***MinValue*** parameter just discussed allows us to specify the minimum requested resolution for the progressive mesh as either a minimum vertex count or a minimum face count. The ***Options*** parameter allows us to specify one member of the D3DXMESHSIMP enumerated type to tell the progressive mesh object how to interpret the ***MinValue*** parameter. The enumerated type has two members:

```
typedef enum _D3DXMESHSIMP
{
        D3DXMESHSIMP_VERTEX = 1,
        D3DXMESHSIMP_FACE = 2
} D3DXMESHSIMP;
```

LPD3DXPMESH *ppPMesh
The final parameter is the address of an ID3DXPMesh interface pointer. If the function is successful and the progressive mesh is successfully created, this will point to a valid interface for that progressive mesh object.

8.9.3.1 Data Validation and Cleaning

Certain types of geometric data may fail to be converted into a progressive mesh because the dataset is considered invalid. Common problems include twisted or invalid faces or adjacency loops that make the simplification table building process impossible. While this is generally caused by the input data being corrupt or erroneous to begin with, there is a way for us to validate whether a mesh that we have just loaded or created will make an appropriate progressive mesh. The D3DX library contains a function called D3DXValidMesh that will validate an ID3DXMesh object and return information describing whether it is suitable for progressive mesh generation or has geometric problems that will cause the progressive mesh generation process to fail.

```
HRESULT D3DXValidMesh
(
    LPD3DXMESH pMeshIn,
    CONST DWORD *pAdjacency,
    LPD3DXBUFFER *ppErrorsAndWarnings
);
```

LPD3DXMESH pMeshIn
This is the ID3DXMesh interface that we wish to test for progressive mesh compliance. If the function returns D3D_OK, then this mesh data is suitable for progressive mesh generation.

CONST DWORD *pAdjacency
This is a pointer to an array of DWORDs (3 for each triangle in the input mesh) describing the adjacency information for each triangle in the mesh. This information is generated automatically by the D3DXLoadMeshFromX function or can be requested from the input mesh using the ID3DXMesh::GenerateAdjacency function.

LPD3DXBUFFER *ppErrorsAndWarnings
If the function returns D3DERR_INVALIDMESH, this buffer will contain a *string* of errors and warnings describing where and why the mesh data contains errors. This will be useful for debugging the mesh.

If the mesh is not suitable for progressive mesh generation, the D3DX library provides a function called D3DXCleanMesh which can be used to try to repair the invalid geometry. So before we generate a progressive mesh from an ID3DXMesh, we should call the D3DXValidMesh function to test for compliance, and if the standards are not met, call D3DXCleanMesh to repair the data.

D3DXCleanMesh actually creates a new mesh along with new adjacency information. The original mesh and adjacency information buffer are not altered by the function so you can keep them if you wish. However you will generally want to release them and use the new data this function produces.

```
HRESULT D3DXCleanMesh
(
    D3DXCLEANTYPE CleanType,
    LPD3DXMESH    pMeshIn,
    const DWORD  *pAdjacencyIn,
    LPD3DXMESH   *ppMeshOut,
    DWORD        *pAdjacencyOut,
    LPD3DXBUFFER *ppErrorsAndWarnings
);
```

D3DXCLEANTYPE CleanType
This first method allows you to control how aggressively the cleanup is performed and control exactly what is considered for cleaning. It can be one of the following enumeration members. Note that although there are actually five cleaning options, there are only really two cleaning techniques you can switch on and off: the merging of back faces and front faces that share the same space, and the removal of bowties.

```
typedef enum _D3DXCLEANTYPE
 {
    D3DXCLEAN_BACKFACING = 1,
    D3DXCLEAN_BOWTIES    = 2,
    D3DXCLEAN_SKINNING   = D3DXCLEAN_BACKFACING,
    D3DXCLEAN_OPTIMIZATION   = D3DXCLEAN_BACKFACING,
    D3DXCLEAN_SIMPLIFICATION = D3DXCLEAN_BACKFACING|D3DXCLEAN_BOWTIES
} D3DXCLEANTYPE;
```

D3DXCLEAN_BACKFACING
Merge triangles that share the same vertex indices but have face normals pointing in opposite directions (back-facing triangles). Unless the triangles are not split by adding a replicated vertex, mesh adjacency data from the two triangles may conflict. This is something you will usually want to do before performing optimization on a mesh or converting that mesh to a skin (skins will be discussed later in the course).

D3DXCLEAN_BOWTIES
If a vertex is the apex of two triangle fans (a bowtie) and mesh operations will affect one of the fans, then split the shared vertex into two new vertices. Bowties can cause problems for operations such as mesh simplification that remove vertices, because removing one vertex will affect two distinct sets of triangles.

D3DXCLEAN_SKINNING
Use this flag to prevent infinite loops during skinning setup mesh operations. It is the same as specifying D3DXCLEAN_BACKFACING.

D3DXCLEAN_OPTIMIZATION
Use this flag to prevent infinite loops during mesh optimization operations. It is the same as specifying D3DXCLEAN_BACKFACING

D3DXCLEAN_SIMPLIFICATION
Use this flag to prevent infinite loops during mesh simplification operations. Performs back face merging and bowtie removal.

LPD3DXMESH pMeshIn
This is the interface pointer to the ID3DXMesh that you would like to try to repair before progressive mesh generation. It will typically be a mesh that failed the validation test. This mesh is not affected by the operation and the clean data is output in a separate mesh. After the mesh has been cleaned, this input mesh can be released.

CONST DWORD *pAdjacencyIn
This is where you pass in the pointer to the input mesh's adjacency information. This array should contain 3 DWORDs per triangle in the input mesh. This adjacency buffer can be released after the function returns and a new cleaned mesh has been generated as it may no longer describe the adjacency information of the output mesh correctly. It is merely used for the cleaning process and can be safely discarded on function return.

> **Note:** For virtually all functions and methods that expect adjacency input information, we can specify NULL for this parameter. If we do, the function will internally generate the mesh adjacency information by calling the mesh's GenerateAdjacency method behind the scenes.

LPD3DXMESH *ppMeshOut
This is the address of an ID3DXMesh pointer that will point to a new mesh containing the cleaned mesh data. If the input mesh did not require cleaning, then this interface will simply be another interface to the input mesh object. The input mesh interface can still be safely released as the reference count mechanism will prevent the underlying mesh object from being destroyed.

DWORD *pAdjacencyOut
This is a pointer to a pre-allocated buffer that the function will fill with the cleaned mesh's adjacency information. It should be large enough to store three DWORDs for each triangle in the cleaned mesh. It is worth noting that the adjacency input buffer can also be used as the adjacency output buffer (i.e. you can pass in the same pointer for both pAdjacencyIn and pAdjacencyOut). The input adjacency information will be overwritten with the new adjacency information for the cleaned mesh in that case.

LPD3DXBUFFER *ErrorsAndWarnings
This is the address of an ID3DXBuffer pointer that will be used to allocate a buffer of error information if the mesh cleaning fails.

Generating a progressive mesh in our code might look as follows:

```
HRESULT      hRet;
LPD3DXMESH   pStandardMesh;
LPD3DXPMESH  pProgressiveMesh;
ID3DXBuffer *pAdj;
ID3DXBuffer *pEff;
ID3DXBuffer *pMat;
DWORD        NumMaterials;
DWORD       *pAdjacency;
```

```
// Load our standard mesh
hRet = D3DXLoadMeshFromX("Factory.x" ,D3DXMESH_MANAGED, pDevice, &pAdj, &pMat,
                         &pEff, &NumMaterials , &pStandardMesh);

if (FAILED(hRet)) return false;

// We wont use the effects buffer so release it
pEff->Release();

pAdjacency = (DWORD*) pAdj->GetBufferPointer();

// Check to see if mesh is valid
hRet = D3DXValidMesh( pStandardMesh, pAdjacency, NULL );

if (FAILED(hRet))
{
    LPD3DXMESH pCleanedMesh;
    // Attempt to repair the mesh
    hRet = D3DXCleanMesh( pStandardMesh, pAdjacency, &pCleanedMesh, pAdjacency );


            if (FAILED(hRet))
            {
                pMat->Release();
                pStandardMesh->Release();
                pAdj->Release();
                return false;
            }

    // We repaired ok, let's store pointer
    pStandardMesh->Release();
    pStandardMesh = pCleanedMesh;

} // End if invalid mesh

// Generate our progressive mesh and request simplification down to potentially
// 10 faces
hRet = D3DXGeneratePMesh(pStandardMesh, pAdjacency, NULL, NULL, 10,
                         D3DXMESHSIMP_FACE, &pProgressiveMesh );

            if (FAILED(hRet))
            {
                pMat->Release();
                pStandardMesh->Release();
                pAdj->Release();
                return false;
            }

// We're done, release our original mesh
pStandardMesh->Release();
```

In the code above we loaded X file data into a standard ID3DXMesh and then validated it. If it passed validation then we generated a progressive mesh with the requested minimum level of detail of 10 faces.

If the mesh failed the validation, then we cleaned it, and if the mesh was cleaned successfully, went on to generate the progressive mesh from the newly cleaned mesh.

> **Note:** When a progressive mesh is first generated, its current level of detail is initially set to its lowest level of detail.

8.9.4 Setting LOD

Because ID3DXPMesh is derived from ID3DXBaseMesh, all of the same rendering functionality is available. To render the progressive mesh, we loop through each of its subsets and call ID3DXPMesh::DrawSubset in exactly the same way that we rendered a standard mesh. This call will render the mesh at its currently set level of detail.

To increase/decrease the current level of detail is quite simple: we just specify a desired vertex count or a desired face count. Given what we have learned thus far, we cannot specify a face or vertex count that is lower than the minimum value specified during progressive mesh creation because precompiled information will not be available down past that level. We also cannot specify a vertex or face count that is higher than the vertex or face count of the original model since this describes the maximum level of detail for the mesh. Specifying counts outside the allowed range will result in clamping to the minimum or maximum level of detail allowed by the mesh.

ID3DXPMesh exposes two functions to allow for current LOD configuration of either the vertex or face counts: ID3DXPMesh::SetNumFaces and ID3DXPMesh::SetNumVertices.

```
HRESULT SetNumFaces( DWORD Faces );
```

DWORD Faces
The only parameter to this function is the number of faces we would like for the current LOD. Clamping will occur if the value is outside the min/max ranges discussed previously. It should also be noted that this value is a request only. While the progressive mesh will try to achieve the requested face count, the exact face count may not be able to be reached. The final LOD may have more or fewer faces than the requested value.

```
HRESULT SetNumVertices( DWORD Vertices );
```

DWORD Vertices
This parameter tells the function the desired number of vertices for the current LOD. As with the previous function, clamping will occur if the value is outside the min/max ranges and the value is a request only.

These two functions are all we need to perform dynamic LOD on our meshes (we will usually use one or the other). In the following pseudo code, we see how we might subtract detail from the object based on whether the current frame rate has dropped below a target threshold.

```
// If the frame rate has dropped unacceptably low, reduce the mesh face count in increments of
// 10 until (hopefully) it becomes acceptable
if ( CurrentFrameRate < MinimumFrameRate )
{
        pMesh->SetNumFaces( pMesh->GetNumFaces() – 10 );
}
else
// If the current frame rate is some way over an acceptable frame rate, we can introduce more
// detail if available, making sure we create a richer environment on higher powered machines.
if ( CurrentFrameRate > AcceptableFrameRate + 10 )
{
       pMesh->SetNumFaces( pMesh->GetNumFaces() +10 );
}

// Render the mesh
for ( int I = 0; I < NumOfAttributes; I++)
{
          pMesh->DrawSubset (I);
}
```

8.9.5 Retrieving LOD

As discussed in the last section, there are min/max face and vertex counts that define the extreme LOD ranges in a progressive mesh. It is useful to know these thresholds when performing runtime simplification so that you do not waste time calling ID3DXPMesh::SetNumFaces function to add/reduce the face/vertex count of a mesh that is already at its maximum or minimum level of detail. ID3DXPMesh exposes four functions to retrieve these values.

`DWORD GetMaxFaces(VOID)`
This function returns the maximum number of faces the mesh can be set to. Any calls to ID3DXPMesh::SetNumFaces that pass a value larger than the value returned by this function will be clamped to this value. The mesh can never have more faces than the value returned by this function.

`DWORD GetMaxVertices(VOID)`
This function returns the maximum number of vertices the mesh can be set to with the ID3DXPMesh::SetNumVertices function. If we specify a value to the ID3DXPMesh::SetNumVertices function that is larger than the value returned from this function, the value will be clamped to this max vertex value.

`DWORD GetMinFaces(VOID)`
This function returns the minimum number of faces the mesh can be set to. Any calls to ID3DXPMesh::SetNumFaces that pass a value smaller than the value returned by this function will be clamped to this value. The mesh can never have fewer faces than the value returned by this function.

`DWORD GetMinVertices(VOID)`
This function returns the minimum number of vertices the mesh can be set to with the ID3DXPMesh::SetNumVertices function. If we specify a value to the ID3DXPMesh::SetNumVertices

function that is smaller than the value returned from this function, the value will be clamped to this minimum vertex value.

It is also useful to know the current LOD the progressive mesh is using, so ID3DXPMesh exposes two functions that return the current number of vertices used by the mesh and the current number of triangles. As mentioned in the last section, when we set the LOD using SetNumFaces or SetNumVertices, the exact face or vertex count requested may not actually be achievable and the mesh will be simplified as near as possible to the value requested. Therefore, after we call SetNumFaces or SetNumVertices we can use GetNumFaces and GetNumVertices to retrieve the exact face or vertex count the mesh was simplified to.

`DWORD GetNumFaces(VOID)`
Returns the current number of triangles used to render the mesh at its current level of detail.

`DWORD GetNumVertices(VOID)`
Returns the current number of vertices used to render the mesh at its current level of detail.

8.9.6 LOD Trimming

When we set the current level of detail using either SetNumVertices or SetNumFaces, we know that no vertices or triangles are physically being deleted from the vertex and index buffers. Only count values are being incremented and decremented to determine which sections of our buffers are used to render the mesh. However, at some point we may want to physically and persistently reduce the dynamic simplification range of the mesh by *trimming* the mesh. Trimming allows us to lower the maximum level of detail and increase the minimum level of detail that the mesh can be set to. Trimming is much more than simply setting a new minimum and maximum value. Instead, memory is physically freed as a result of this process.

Trimming is useful if you decide that you do not need the mesh to ever use its maximum LOD -- in which case you can set a new lower maximum. The mesh can free the vertices and indices at the end of its buffers as well as the memory taken up by the pre-compiled simplification steps that process the edge collapses at these higher LODs.

While it is clear that reducing the maximum level of detail will free up memory and result in a persistent change, it might not be immediately obvious why setting a new (higher) minimum level of detail would be beneficial. The reason is that when we specify a minimum LOD at mesh creation time, a series of edge collapse information structures have to be stored (one for each edge collapse) describing the step-by-step simplification (edge-by-edge) down to the minimum LOD. If we decide that we do not wish to simplify the mesh beyond a certain point, then we can release the memory used by the simplification steps at the bottom of the edge collapse chain. So ‘trimming’ is an appropriate description of what is occurring. We are contracting the dynamic range of the mesh and making it more memory efficient.

> **Note:** Trimming is a permanent change. Once you trim the maximum level of detail for example, you can never increase that maximum value. Likewise, once you increase the minimum level of detail, the edge

collapse information at the bottom of the simplification chain is discarded and you can never decrease that minimum value. The data is gone for good.

ID3DXPMesh exposes two trimming functions to alter the dynamic simplification range of the progressive mesh:

```
HRESULT TrimByFaces
(
    DWORD NewFacesMin, DWORD NewFacesMax,
    DWORD *rgiFaceRemap, DWORD *rgiVertRemap
);

HRESULT TrimByVertices
(
    DWORD NewVerticesMin, DWORD NewVerticessMax,
    DWORD *rgiFaceRemap,  DWORD *rgiVertRemap
);
```

The first two parameters to each function are the new dynamic range of the progressive mesh. In the case of TrimByFaces, these are the new minimum and maximum face count. NewFacesMin must be greater than or equal to the current minimum number of faces and NewFacesMax must be less than or equal to the current maximum number of faces. The same logic holds true for TrimByVertices for vertex counts instead of face counts.

Both functions accept two pointers to two pre-allocated DWORD buffers which return the vertex and face re-mapping information caused by the trimming function. The trimming function may cause vertices and indices to be shuffled about inside the index and vertex buffers and this could be problematic if you have external objects or structures linked to vertices or faces by index. Although both of these parameters can be set to NULL (and probably will be most of the time), if you need to know where vertices or faces have moved, you can supply these buffers and the function will fill them.

The rgiFaceRemap parameter should be large enough to hold one DWORD for every face in the pre-trimmed mesh's maximum LOD (its maximum face count). When the function returns, each element in the array will describe how that face has been mapped to another. If rgiFaceRemap[5] equals 10 for example, then this means the face that was originally 6^{th} in the index buffer has now been moved to the 11^{th} slot in the buffer.

The rgiVertexRemap parameter should either be set to NULL or it should point to a buffer large enough to hold one DWORD for every vertex in the pre-trimmed mesh's maximum vertex count. When the function returns, this buffer will describe the new location of a vertex in the vertex buffer. If rgiVertexRemap[200] equals 10 for example, it means that the 201^{st} vertex in the vertex buffer has now been moved to the 11^{th} position in the vertex buffer.

The following code snippet shows how we might trim a progressive mesh to subtract 200 triangles from its dynamic range. To do so we decrease maximum face count by 100 triangles and increase the minimum face count by 100 triangles.

```
pMesh->TrimByFaces(pMesh->GetMinFaces()+100,pMesh->GetMaxFaces()-100, NULL, NULL);
```

8.9.7 Progressive Mesh to Standard Mesh Cloning

The ID3DXPMesh interface is derived from ID3DXBaseMesh and as such, inherits ID3DXBaseMesh::CloneMeshFVF (and the declarator version not discussed in this course). This function will allow us to create an ID3DXMesh clone of the progressive mesh at its *current* level of detail. This function is identical to the ID3DXMesh::CloneMeshFVF function covered earlier in the chapter.

This presents an interesting use of the progressive mesh as a one-off simplification utility class for meshes that will not use dynamic LOD in the run-time engine and therefore do not require the rather large simplification step chain to be computed and stored in memory. We can load a standard ID3DXMesh, use it to generate a progressive mesh using the D3DXGeneratePMesh function, set the level of detail to a desired face or vertex count, and then clone the progressive mesh using CloneMeshFVF back out into a vanilla ID3DXMesh. The clone would be a standard ID3DXMesh with a lower level of detail than the original dataset and the progressive mesh interface could be released as we no longer need its simplification abilities.

While there is nothing wrong with using ID3DXPMesh to perform one-off standard mesh simplifications in this way, this is exactly what the ID3DXSPMesh interface is designed to do. It simplifies the mesh in the same way but does not incur any of the memory overhead of the progressive mesh caused by the progressive mesh's runtime simplification system. We will discuss simplification meshes shortly.

The code snippet below demonstrates progressive mesh cloning for mesh simplification. The code assumes that the input mesh has already been validated and has more than 1000 faces to start.

```
ID3DXMesh  *pStandardMesh; // Assume this mesh already contains data
ID3DXPMesh *pPMesh;

D3DXGeneratePMesh(pStandardMesh , pAdjacency , NULL , NULL , 1 ,
                  D3DMESHSIMP_FACE , &pPMesh);

pStandardMesh->Release();
pPMesh->SetNumFaces( 1000 );

pPMesh->CloneMeshFVF( D3DXMESH_MANAGED , pPMesh->GetFVF  ,
                      pDevice , &pStandardMesh);

pPMesh->Release();
```

The result of the above example is a regular ID3DXMesh which has had its face count reduced using a temporary ID3DXPMesh to carry out the simplification task.

8.9.8 Progressive Mesh to Progressive Mesh Cloning

ID3DXPMesh also includes a cloning function (two functions if you count the declarator version) that allows us to clone a progressive mesh to a new progressive mesh. This is useful if we need to make a copy of the progressive mesh and have it remain progressive, or if we would like to change the vertex format of an already progressive mesh, perhaps to make room for a new component.

```
HRESULT ClonePMeshFVF
(
    DWORD Options,
    DWORD FVF,
    LPDIRECT3DDEVICE9 pDevice,
    LPD3DXPMESH *ppCloneMesh
);
```

The parameter list is identical to the standard cloning function except for the fact that the last parameter must be the address of a pointer to an ID3DXPMesh rather than an ID3DXMesh interface. If the function is successful, the new progressive mesh will contain all of the mesh geometry and collapse structures as the original and it will share the same current LOD state. That is, if we have a 1000 face progressive mesh, reduce its current face count to 100 and then clone it, the clone will also be set to render using 100 faces. Of course, the identical collapse structures and geometry are stored within the clone, so it can always adjust its LOD independently after it is created and can also be adjusted back up to a 1000 face mesh (the maximum detail level of the progressive mesh from which it was cloned).

8.9.9 Progressive Mesh Optimization

The progressive mesh interface exposes two optimization functions, much like the ID3DXMesh. The first, ID3DXPMesh::Optimize creates an optimized standard (non-progressive) mesh whose dataset contains the faces and vertices of the progressive mesh's *current* level of detail. This is exactly like cloning the progressive mesh using the ID3DXPMesh::CloneMeshFVF function to create a standard output mesh and then performing an OptimizeInPlace on it. The original progressive mesh is not altered by this method. This can be a useful way for generating optimized low-poly representations of complex meshes. The function is identical to the ID3DXMesh::Optimize function and as such the parameter list will not be explained again. The function is shown below for your reference.

```
HRESULT ID3DXPMesh::Optimize
(
    DWORD Flags,
    DWORD *pAdjacencyOut,
    DWORD *pFaceRemap,
    LPD3DXBUFFER *ppVertexRemap,
    LPD3DXMESH *ppOptMesh
);
```

ID3DXPMesh does not expose the OptimizeInPlace function for optimization of the current mesh data. This is due in part to the fact that the optimizations that can be performed on the progressive mesh base geometry data are limited by the fact that triangles and vertices have to be in a precise order so that edges can be correctly and speedily collapsed and split. It is still possible for triangles to be reordered in the progressive mesh index buffer in a limited way to optimize for attribute sorting or higher vertex cache hit rates, but the vertex data itself will not be touched. To make this distinction clear, the ID3DXPMesh interface exposes a function to perform an in-place optimization of the progressive mesh base geometry. The name of this function is ID3DXPMesh::OptimizeBaseLOD and it accepts a much smaller parameter list than the ID3DXMesh::OptimizeInPlace function.

```
HRESULT OptimizeBaseLOD
(
    DWORD Flags,
    DWORD *pFaceMap
);
```

DWORD Flags
This is one of the D3DXMESHOPT flags discussed earlier in the chapter (see ID3DXMesh::Optimize or the ID3DXMesh::OptimizeInPlace function). The efficiency of the optimization is limited by the requirements of the progressive mesh system where collapse order takes precedence.

DWORD *pFaceMap
This is a pointer to an array large enough to hold one DWORD for each face in the pre-optimized mesh. When the function returns, it will be filled with information describing how triangles may have moved. This allows us to correct external references if necessary.

8.9.10 Vertex History

Edge collapsing and splitting can result in a visible popping effect as the mesh topology changes suddenly between detail levels. To solve this problem, many 3D engines use a morphing technique to gradually perform the collapse in a controlled fashion over time. As long as the vertex to be collapsed and the neighbor it will collapse onto are known, this is a relatively easy thing to do.

While ID3DXPMesh does not support morphing directly, it does expose a function that can be used to assist us if we want to do it on the application side. ID3DXPMesh::GenerateVertexHistory will fill an application supplied DWORD array (one DWORD per max vertices in the mesh) describing the new index for a vertex collapse.

```
HRESULT GenerateVertexHistory(DWORD *pVertexHistory);
```

For example, if pVertexHistory[500] equals 2 and pVertexHistory[505] equals 2, this tells is that at the current level of detail, vertices 500 and 505 have both been collapsed onto vertex 2.

Morphing techniques are beyond the scope of this chapter, but there are plenty of references in books and on the web if you are interested in researching it. We will examine morphing concepts later in this course series.

8.10 ID3DXSPMesh

The last mesh interface that we will cover in this chapter is the simplification mesh: ID3DXSPMesh. Fortunately, we will be able to cover this interface very quickly since nearly all of the progressive mesh concepts apply to simplification meshes. In fact, we can think of a simplification mesh as a progressive mesh with no dynamic LOD or rendering functionality. In this sense, it is a utility object that is used to apply polygon reduction methods to a standard input mesh to produce a lower detail result. For performance reasons we would not want to use this class for real-time work, but it is quite useful for performing a one-time simplification offline in a tool or at application startup. Perhaps you would use this interface to tailor mesh LOD according to the capabilities of the end user's machine.

Unlike the ID3DXMesh interface and the ID3DXPMesh interface, the ID3DXSPMesh interface is not derived from the ID3DXBaseMesh interface; it is derived directly from IUnknown. This means that the simplification mesh object does not inherit DrawSubset and as such, is not explicitly renderable.

ID3DXSPMesh exposes the ReduceFaces and ReduceVertices functions to simplify the mesh which, once done, are irreversible. Once the mesh is reduced, it must be cloned for rendering. ID3DXSPMesh has two cloning functions (four if you include declarator versions): CloneMeshFVF which clones the simplified data out to a standard output mesh and ClonePMeshFVF which clones the simplified data out to a progressive mesh. In the latter case, the simplified data will serve as the base geometry (maximum LOD) for the progressive mesh. Once cloned, the simplification mesh can be released. It is faster and more memory efficient to use a simplification mesh for one-time reduction because the overhead of building the runtime edge collapse structures for a progressive mesh is avoided.

8.10.1 Simplification Mesh Creation

We create a simplification mesh object using the global D3DX function D3DXCreateSPMesh. Like the progressive mesh, a simplification mesh needs to be created from a standard ID3DXMesh and cannot be loaded from a file or created manually. The parameter list is almost identical to the GeneratePMesh function used to generate progressive meshes.

```
HRESULT D3DXCreateSPMesh
(
    LPD3DXMESH pMesh,
    CONST DWORD *pAdjacency,
    CONST LPD3DXATTRIBUTEWEIGHTS pVertexAttributeWeights,
    CONST FLOAT *pVertexWeights,
    LPD3DXSPMESH *ppSMesh
);
```

The function accepts a standard ID3DXMesh containing the dataset to simplify. The second parameter points to the input mesh adjacency information. The third and fourth parameters allow us to pass in priorities for vertex component types and for the individual vertices themselves.

The final parameter is the address of a pointer to an ID3DXSPMesh interface to be created on success. As with the generation of a progressive mesh, illegal geometry will cause the function to fail, so we should call D3DXValidMesh and D3DXCleanMesh to repair the mesh in such cases.

The code below shows how we might create a simplification mesh.

```
HRESULT      hRet;
LPD3DXMESH   pStandardMesh;
LPD3DXSPMESH pSimpMesh;
ID3DXBuffer *pAdj;
ID3DXBuffer *pEff;
ID3DXBuffer *pMat;
DWORD        NumMaterials;
DWORD       *pAdjacency;

// Load our standard mesh
hRet = D3DXLoadMeshFromX("Factory.x" ,D3DXMESH_MANAGED, pDevice, &pAdj, &pMat,
                         &pEff, &NumMaterials , &pStandardMesh);

if (FAILED(hRet)) return false;

// We wont use the effects buffer so release it
pEff->Release();

pAdjacency = (DWORD*) pAdj->GetBufferPointer();

// Check to see if mesh is valid
hRet = D3DXValidMesh( pStandardMesh, pAdjacency, NULL );

if (FAILED(hRet))
{
    LPD3DXMESH pCleanedMesh;

    // Attempt to repair the mesh
    hRet = D3DXCleanMesh( pStandardMesh, pAdjacency, &pCleanedMesh, pAdjacency );

    if (FAILED(hRet))
    {
        pMat->Release();
        pStandardMesh->Release();
        pAdj->Release();
        return false;
    }

    // We repaired ok, let's store pointer
    pStandardMesh->Release();
    pStandardMesh = pCleanedMesh;

} // End if invalid mesh

// Generate our simplification mesh object
hRet = D3DXCreateSPMesh( pStandardMesh, pAdjacency, NULL, NULL, &pSimpMesh );
if (FAILED(hRet))
{
    pMat->Release();
```

```
    pStandardMesh->Release();
    pAdj->Release();
    return false;
}

// We're done, release our original mesh
pStandardMesh->Release();
```

8.10.2 Simplification Mesh Usage

We can simplify the mesh data by calling ReduceFaces or ReduceVertices. Both calls accept a single DWORD that describes the target face or vertex count, respectively. This count must be less than or equal to the face or vertex count in the original input mesh.

```
HRESULT ID3DXSPMesh::ReduceFaces( DWORD Faces );
HRESULT ID3DXSPMesh::ReduceVertices( DWORD Vertices );
```

In the case of both of these functions, the exact target face count or vertex count requested may not be able to be met due to certain restrictions.

Continuing the previous code listing, we could request a simplification of the mesh data down to 10 faces using the following code:

```
pSimpMesh->ReduceFaces ( 10 );
```

ID3DXSPMesh also exposes the GetNumFaces and GetNumVertices functions so that we can inquire about the exact face or vertex count after simplification.

```
DWORD ActualNumberOfFaces = pSimpMesh->GetNumFaces();
```

Since ID3DXSPMesh supports no rendering functions, we will need to clone the simplified data out to another mesh type. ID3DXSPMesh::CloneMeshFVF clones the simplified dataset into a standard ID3DXMesh which can then be used for render operations and is identical to this same call in other mesh interfaces.

```
HRESULT ID3DXSPMesh::CloneMeshFVF
(
    DWORD Options,
    DWORD FVF,
    LPDIRECT3DDEVICE9 pDevice,
    DWORD *pAdjacencyOut,
    DWORD *pVertexRemapOut,
    LPD3DXMESH *ppCloneMesh
);
```

ID3DXSPMesh::ClonePMesh is identical to the previous function with the only difference being the final parameter changes to LPD3DXPMESH (a progressive mesh). The maximum level of detail (the base LOD) of the newly created progressive mesh will be the same as the current simplified mesh.

8.10.3 ID3DXSPMesh Misc. Methods

Before we finish up our discussion of the simplification mesh, let us briefly look at the remaining functions exposed by the interface.

HRESULT GetDevice(LPDIRECT3DDEVICE9 *ppDevice)
This function returns a pointer to the device to which the simplification mesh is bound. Notice that we never passed in a pointer to an IDirect3DDevice9 interface when we created the simplification mesh. That is because the owner device is inherited from the standard input mesh.

DWORD GetFVF(VOID)
Allows us to retrieve the FVF flags of the simplification mesh.

DWORD GetMaxFaces(VOID)
Returns the number of faces that exist in the input mesh.

DWORD GetMaxVertices(VOID)
Returns the number of vertices that exist in the input mesh.

DWORD GetOptions(VOID)
Returns a DWORD describing the creation flags for the simplification mesh. What we are really getting here are the creation options for the input mesh since we never explicitly specify creation options when generating a simplification mesh.

HRESULT GetVertexAttributeWeights(LPD3DXATTRIBUTEWEIGHTS *pAttributeWeights*)
Returns the D3DXATTRIBUTEWEIGHTS structure used to perform simplification. This structure describes how much weight each vertex component has when calculating the error metric between two vertices. It will contain either the same values that you passed into the D3DXCreateSPMesh function or a structure containing the default values if you passed NULL. The default values use only the position, weight, and normal of the vertex when calculating vertex collapse errors.

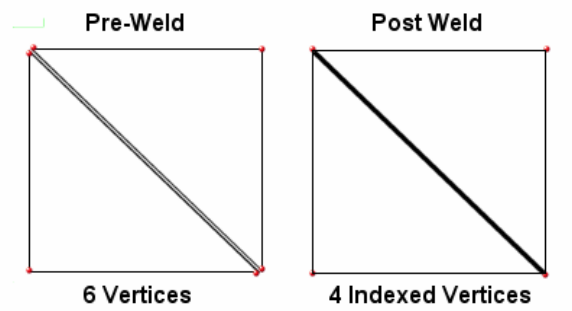
HRESULT GetVertexWeights(FLOAT *pVertexWeights)
Allows you to retrieve the weight of each vertex used to bias the error metric for collapses. If you passed in a per-vertex weights array when you called D3DXCreateSPMesh, then this buffer will be returned with the weight information of each vertex that you passed in. If you passed NULL, then the buffer will be filled with the default weights for each vertex (which is 1.0). This parameter should point to a pre-allocated array that is large enough to hold one DWORD for every vertex in the original/input mesh.

8.11 Global Utility Functions

Before wrapping up this chapter, let us briefly examine some global D3DX functions that can be used with the mesh types we have discussed.

8.11.1 D3DXWeldVertices

This function allows us to weld vertices together in the mesh's vertex buffer. It is essentially another function that can clean and optimize the data for a mesh. This is very useful if the level editor you are using does not generate indices and provides each triangle with its own unique (three) vertices. Consider a quad for example that is stored using 6 vertices (3 for each triangle).



In the case of a D3DX mesh, where all data is stored as indexed triangle lists, the two duplicated vertices at the bottom right and top left corners of the quad could be represented by one vertex (the other discarded) and the indices remapped. Of course, if the two triangles are not supposed to be two halves of the same quad surface and instead had unique textures or colors applied, we definitely would not want to force a vertex weld, as fusing the two vertices into one would cause mesh color/texture distortion.

If two vertices are very close together and share similar or identical properties however, we can use the D3DXWeldVertices function to collapse them into one vertex and reduce the size of our mesh. What this function essentially does is look for vertices that are duplicated and collapse them into a single vertex. This involves removing one of the redundant vertices and changing all indices that reference that vertex to point at the one that remained instead. The function is shown below along with a description of its parameter list.

```
HRESULT D3DXWeldVertices
(
    const LPD3DXMESH pMesh,
    DWORD Flags,
    CONST D3DXWELDEPSILONS* pEpsilon,
    CONST DWORD* pAdjacencyIn,
    DWORD* pAdjacencyOut,
    DWORD* pFaceRemap,
    LPD3DXBUFFER* ppVertexRemap
);
```

const LPD3DXMESH pMesh
This is a standard ID3DXMesh interface pointer to a mesh you want to have welded.

DWORD Flags
A combination of zero or more D3DXMESH flags. Recall that we used these to create a mesh and specify which resource pool the mesh is created in. Since this function does not explicitly generate an output mesh, we can assume that the vertex and index buffers are destroyed and rebuilt because we are able to change resource pools.

CONST D3DXWELDEPSILONS *pEpsilon
To determine whether two vertices should be welded (if they are considered so alike that collapsing them into a single vertex would not significantly alter the appearance of the mesh), a comparison is made between vertex components. If the components match, then the vertices should be welded. To provide tolerance for floating point inaccuracy (ex. 0.998 vs. 0.997), we pass the address of a D3DXWELDEPSILONS structure. It contains a floating point epsilon member for every possible vertex component. This allows us to fine tune exactly how fuzzy the comparisons are when searching for like vertices.

```
typedef struct _D3DXWELDEPSILONS
{
        FLOAT Position;
        FLOAT BlendWeights;
         FLOAT Normal;
         FLOAT PSize;
         FLOAT Specular;
         FLOAT Diffuse;
         FLOAT Texcoord[8];
         FLOAT Tangent;
         FLOAT Binormal;
         FLOAT TessFactor;
} D3DXWELDEPSILONS;
```

Do not concern yourself with the unfamiliar vertex components for now. The vertex structures we will use for the time being require only the components we have seen thus far in the course series.

If we set the Position member to 0.005 and the distance between two vertices is less than 0.005, they will be considered a match (for position). Since we can set tolerance values for each vertex component individually, we might specify a bigger tolerance for the vertex normal test than for the diffuse color. Alternatively, setting the TexCoord[0] member to 0.0 for example means that vertices will only be welded if their first set of texture coordinates are an exact match.

The following code shows how we might fill this structure to perform a weld when each existing vertex component is compared using a tolerance of 0.001.

```
D3DXWELDEPSILONS Epsilons;

 // Set all epsilons to 0.001;
 float *pFloats = (float*)&Epsilons;

 for ( ULONG i = 0; i < sizeof(D3DXWELDEPSILONS) / sizeof(float); i++ )
      *pFloats++ = 1e-3f;
```

CONST DWORD *pAdjacencyIn
This is the adjacency information for the pre-welded mesh.

DWORD* pAdjacencyOut
If we require adjacency information after the mesh has been welded, we can pass a pointer to a DWORD buffer here. It should be large enough to hold 3 DWORDs for every triangle in the pre-optimized mesh.

DWORD* pFaceRemap
LPD3DXBUFFER* ppVertexRemap
These parameters can store face and vertex remapping information to reflect the changes that took place during the operation.

Weld Example:

```
D3DXWELDEPSILONS Epsilons;

// Set all epsilons to 0.001;
float * pFloats = (float*)&Epsilons;
for (ULONG i = 0; i < sizeof(D3DXWELDEPSILONS)/sizeof(float); i++)
     *pFloats++ = 1e-3f;

// Optimize the data
m_Mesh.OptimizeInPlace( D3DXMESHOPT_VERTEXCACHE );

// Weld the Data
m_Mesh.WeldVertices( 0, &Epsilons );
```

8.11.2 D3DXComputeNormals

D3DXComputeNormals calculates vertex normals for D3DX mesh objects. Since the function accepts an ID3DXBaseMesh pointer, we can use it to compute the normals for any derived mesh (ID3DXMesh or ID3DXPMesh).

```
HRESULT D3DXComputeNormals
{
    LPD3DXBASEMESH pMesh,
    const DWORD *pAdjacency
);
```

The mesh must have a vertex buffer created with the D3DFVF_NORMAL flag. The per-vertex normal is generated by averaging face normals for faces that share the vertex. If adjacency is provided, replicated vertices are ignored and "smoothed" over. If adjacency is not provided, replicated vertices will have normals averaged only from the faces explicitly referencing them.

This function is extremely useful, especially if you have loaded an X file without vertex normals and would like to add them to the mesh. Simply clone the mesh with the D3DFVF_NORMAL flag set to create room for vertex normals, and then call D3DXComputeNormals to calculate their values.

8.11.3 D3DXSplitMesh

This function is used to split a mesh into multiple meshes. We pass in a source mesh (ID3DXMesh) and the maximum number of vertices allowed per mesh. If the input mesh is larger than the specified maximum size, it will be split into multiple (current / max) ID3DXMesh objects. The new meshes are returned in an ID3DXBuffer as an array of ID3DXMesh pointers.

```
void D3DXSplitMesh
(
    const LPD3DXMESH pMeshIn,
    const DWORD *pAdjacencyIn,
    const DWORD MaxSize,
    const DWORD Options,
    DWORD *pMeshesOut,
    LPD3DXBUFFER *ppMeshArrayOut,
    LPD3DXBUFFER *ppAdjacencyArrayOut,
    LPD3DXBUFFER *ppFaceRemapArrayOut,
    LPD3DXBUFFER *ppVertRemapArrayOut
);
```

const LPD3DXMESH pMeshIn
This is a pointer to the ID3DXMesh interface that will be split into multiple meshes.

const DWORD *pAdjacencyIn
This is a pointer to a buffer containing the input mesh face adjacency information

const DWORD MaxSize
This is the maximum vertex buffer size per mesh.

const DWORD Options
Here we specify zero or more D3DXMESH flags describing concepts like the resource pool for the smaller meshes that are created by this function.

DWORD *pMeshesOut
This DWORD will contain the number of smaller meshes that were created when the function returns. It tells us the number of ID3DXMesh interface pointers in the ppMeshArrayOut buffer.

LPD3DXBUFFER *ppMeshArrayOut
This is the address of an ID3DXBuffer interface pointer that will contain one or more ID3DXMesh interface pointers for the smaller meshes.

LPD3DXBUFFER *ppAdjacencyArrayOut
This buffer will contain face adjacency information for the meshes created. It is an array of N DWORD pointers (one per mesh) that is followed immediately by the actual data (3 DWORDS per face) referenced by these pointers. For example, if we wanted to find the third adjacency DWORD for the fifth mesh we would do the following:

```
DWORD **pAdjacency = (DWORD**)(ppAdjacencyArrayOut->GetBufferPointer());
DWORD   Adjacency = pAdjacency[4][2];
```

LPD3DXBUFFER *ppFaceRemapArrayOut
If we require the face re-map information for each mesh, we can pass the address of an ID3DXBuffer pointer. When the function returns, the buffer will contain the face re-map information (1 DWORD per face) for each mesh created. This is stored in the same way as the adjacency out information described previously. We could retrieve re-map information for the 10^{th} face in the 4^{th} mesh using the following code:

```
DWORD **FaceRemap = (DWORD**)(ppFaceRemapArrayOut->GetBufferPointer());
DWORD   Remap = FaceRemap[3][9];
```

LPD3DXBUFFER *ppVertRemapArrayOut
If we require the vertex re-map information for each mesh, we can pass in the address of an ID3DXBuffer interface pointer. On function return it will contain 1 DWORD per vertex for each mesh created. This information is also stored in the same way as the adjacency and face remap data. We could retrieve the re-map information for the 100^{th} vertex in the 2^{nd} mesh using the following code:

```
DWORD **VRemap = (DWORD**)(ppVRemapArrayOut->GetBufferPointer());
DWORD   Remap  = VRemap[1][99];
```

8.11.4 D3DXSimplifyMesh

As it turns out, there is also a global function to perform one-time mesh simplification. It is an alternative to using the ID3DXSPMesh interface to simplify a standard mesh.

```
HRESULT D3DXSimplifyMesh
(
    LPD3DXMESH pMesh,
    CONST DWORD *pAdjacency,
    CONST LPD3DXATTRIBUTEWEIGHTS pVertexAttributeWeights,
    CONST FLOAT *pVertexWeights,
    DWORD MinValue,
    DWORD Options,
    LPD3DXMESH *ppMesh
);
```

Most of these parameters should be familiar to you by now. The MinValue parameter is where we pass in the number of vertices or the number of faces we would like the input mesh to be simplified to. This value is interpreted based on the Options parameter, as we saw earlier in the chapter.

The final parameter is the address of a D3DXMesh pointer to store the simplified data. At this point we can release the input mesh and use the output mesh for rendering. This is a quick and easy way to perform one-time simplification at application startup with a single function call.

While this function is very convenient, it does perform the simplification from scratch every time it is called. Therefore, if you wish to clone multiple copies of a simplified mesh, it would be more efficient to use ID3DXSPMesh to perform the simplification once, and then use the interface to clone out the copies.

8.11.5 D3DXIntersect

The D3DXIntersect function determines whether a ray intersects any of the faces in a mesh. This is useful for any number of intersection tasks (collision detection, object picking, etc.). Simply pass in a source mesh and a ray start position and direction vector.

```
HRESULT D3DXIntersect
(
    LPD3DXBASEMESH pMesh,
    CONST D3DXVECTOR3 *pRayPos,
    CONST D3DXVECTOR3 *pRayDir,
    BOOL *pHit,
    DWORD *pFaceIndex,
    FLOAT *pU,
    FLOAT *pV,
    FLOAT *pDist,
    LPD3DXBUFFER *ppAllHits,
    DWORD *pCountOfHits
);
```

LPD3DXBASEMESH pMesh
This is a pointer to the mesh that will be tested for intersection. Since this is a pointer to an ID3DXBaseMesh, we can use this function with both ID3DXMesh and ID3DXPMesh meshes.

CONST D3DXVECTOR *pRayPos
This is a 3D vector describing the model space position of the origin of the ray. For example, if you wanted to test whether a ray from a laser gun has hit a mesh, this vector may initially describe the world position of the gun from which the laser is originating. Since the ray origin must be in the mesh model space, we need to multiply it by the inverse world matrix of the input mesh. If the mesh is already defined in world space, then this step is not necessary since both spaces are the same.

CONST D3DXVECTOR3 *pRayDir
This is a unit length model space direction vector describing the direction the ray is heading with respect to the starting position. If the player look vector was aligned with the world X axis when they fired a laser gun, you might want the laser beam to travel along this same direction. Therefore, you would take the player look vector, multiply it by the inverse of the mesh world matrix to convert it to model space, and then pass it into the function.

BOOL *pHit
If the ray intersects any of the triangles in the mesh, this Boolean will be set to true when the function returns (false otherwise).

DWORD *pFaceIndex
If any faces have been intersected by the ray, this value will contain the zero-based index of the closest triangle that was intersected by the ray. The closest triangle is the triangle that was intersected closest to the ray origin (pRayPos).

FLOAT *pU
If a face has been intersected, this float will contain the barycentric U coordinate for the closest triangle. Barycentric coordinates have two components (U, V) much like texture coordinates. They are also in the range [0, 1]. You can use them to calculate the exact texture coordinate, or diffuse/specular color at the point of intersection because they describe the intersection as how much each vertex is weighted in that intersection. If we have a triangle (v1,v2,v3) then the U coordinate tells us how much v2 is weighted into the result and V tells us how much v3 gets weighted into the result. To find the weight for v1 we simply subtract (v1 = 1.0 – U – V). With these three weights we can interpolate the exact color, texture coordinate, or normal at the exact point of intersection within the face.

FLOAT *pV
If a face has been intersected, this float will contain the barycentric V coordinate for the closest triangle (see previous).

FLOAT *pDist
This is the distance from the ray origin to the intersection point on the closest intersected triangle. Perhaps you would use this distance value to check whether the laser gun in our previous example has hit a face that is considered out of range.

LPD3DXBUFFER *ppAllHits
Because the ray used for intersection testing is infinitely long, it is possible that many faces may be intersected. While the last four parameters return only details for the closest triangle intersected, we can also pass a buffer pointer to be filled with all hits. The function will fill an array of D3DXINTERSECTINFO structures. The D3DXINTERSECTINFO structure contains the face index, barycentric hit coordinates, and a distance from the intersection point to the ray origin. There will be one of these structures in the buffer for each face intersected by the ray.

```
typedef struct _D3DXINTERSECTINFO
{
    DWORD FaceIndex;
    FLOAT U;
    FLOAT V;
    FLOAT Dist;
} D3DXINTERSECTINFO, *LPD3DXINTERSECTINFO;
```

DWORD *pCountOfHits
This parameter will tell us how many faces were hit by the ray (i.e. the number of D3DXINTERSECTINFO structures in the ppAllHits buffer).

8.11.6 D3DXIntersectSubset

We can also run a ray intersection test on specified mesh subsets. This can be used for a wide variety of interesting features. For example, perhaps certain mesh subsets are more vulnerable to attack than others or require a different AI response to collisions. This function is identical to D3DXIntersect except for the fact that it takes the subset attribute ID as an additional parameter.

```
HRESULT D3DXIntersectSubset
(
    LPD3DXBASEMESH pMesh,
    DWORD AttribId,
    const D3DXVECTOR3 *pRayPos,
    const D3DXVECTOR3 *pRayDir,
    BOOL *pHit,
    DWORD *pFaceIndex,
    FLOAT *pU,
    FLOAT *pV,
    FLOAT *pDist,
    LPD3DXBUFFER *ppAllHits,
    DWORD *pCountOfHits
);
```

8.11.7 D3DXIntersectTri

This is a generic ray/triangle intersection function (it does not take a mesh as input). We pass three vectors describing the positions of the triangle vertices and a ray position and direction vector. The triangle is tested against the ray and returns TRUE if the ray intersects, FALSE if not. We can also pass variables for barycentric hit coordinates and distance from the ray origin.

```
BOOL D3DXIntersectTri
(
    const D3DXVECTOR3 *p0,
    const D3DXVECTOR3 *p1,
    const D3DXVECTOR3 *p2,
    const D3DXVECTOR3 *pRayPos,
    const D3DXVECTOR3 *pRayDir,
    FLOAT *pU, FLOAT *pV,
    FLOAT *pDist
);
```

D3DX also provides functions for generating bounding volumes such as spheres and boxes for D3DX mesh objects. These bounding volumes can be used for quick collision detection and frustum culling. We will look at bounding volumes and intersection testing in much more detail a little later in this course.

Conclusion

That wraps up coverage of the three mesh types we wanted to discuss in this chapter. We have certainly seen how D3DX makes what would otherwise be complex coding tasks very simple. In the next chapter we will look at loading mesh hierarchies, which allow us to deal with our scenes in a more organized and flexible fashion. We will also spend a good deal of time exploring the details of the X file format so that we are comfortable with manually parsing such files and creating our own custom templates when needed. This will set the stage for our detailed examination of the D3DX interfaces that support scene animation.