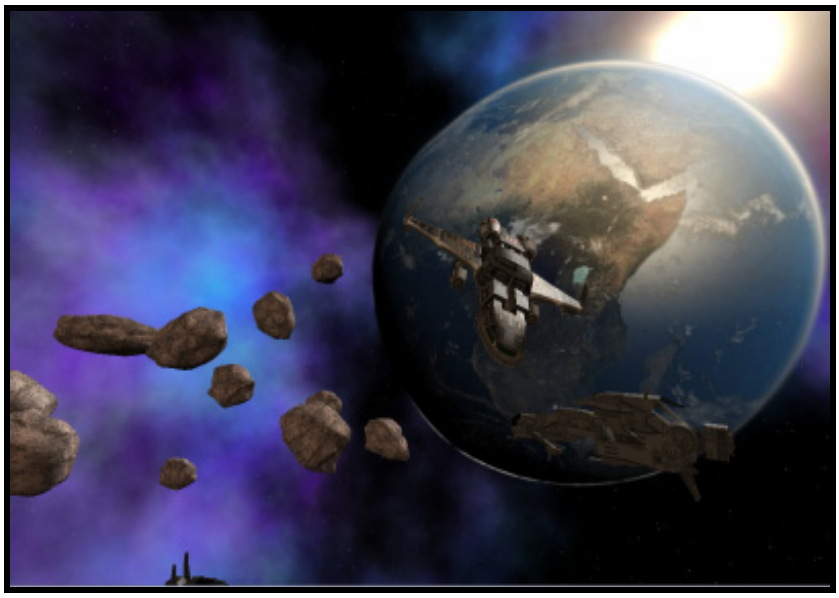
Workbook Eight:
Meshes



Lab Project 8.1: Using ID3DXMesh

The two lab projects in this workbook will be dedicated to the construction of a reusable mesh class. This class will wrap D3DX mesh functionality and employ a sub-system to conveniently handle resource management (the loading and storing of textures and materials). The D3DX library does provide us with an easy means for loading geometry from X files with a single function call, but the burden of loading textures still remains with the application. Our aim is to make this task as easy as possible while avoiding textures being loaded more than once, when they are referenced by more than one mesh.

A mechanism for the efficient rendering of multi-mesh scenes will also need to be put in place. We can not always allow meshes to render themselves independently from the rest of the scene if proper batch rendering is to be achieved. If multiple meshes have subsets that share the same properties, then the scene should be able to intelligently render all subsets from all meshes that share the same device states with a single render call. This will minimize device state changes, an ingredient that is crucial for good performance.

While the D3DX library greatly assists in the loading of X files, if you wish to import your geometry from another source, this has to be accomplished at a lower level. Vertex buffers, index buffers and attribute buffers will need to be locked and manually filled with the data that the application has parsed from a custom file format. An IWF file is an example of such a custom format that we may wish to load and store in a D3DXMesh. This will ensure that the data can benefit from its optimization features. Our wrapper class should expose an interface so that the adding of vertices, indices and attributes is done in as painless a manner as possible. We will expose functions such as AddVertex, AddIndex and AddAttribute which will allow us to build a mesh gradually from data imported from external resources. The vertex buffer, index buffer and attribute buffer will only be constructed internally by the mesh object once all data has been added. This will ensure that we are not locking and unlocking the buffers every time we add a small amount of data (which is highly inefficient).

Finally, as our mesh class is essentially wrapping the underlying ID3DXMesh interface, geometry loaded into our mesh will automatically benefit from the collection of optimization features provided by D3DX.

After the construction of our mesh class you will be able to do the following:

- Populate meshes manually with geometry and attribute data
- Load X files and/or IWF files created by the GILES™ level editor
- Optimize mesh data for rendering
- Create a reusable mesh container class for future applications
- Deal with key aspects of scene state management
    - Texture/material sharing across multiple meshes and their subsets
    - Elimination of redundant texture loading

The CTriMesh Class

The CTriMesh class will be employed as a means for wrapping D3DX mesh functionality. The management of multiple mesh objects and the resources they use will be handled at the scene level. This is made possible by the way each CTriMesh object registers its required resources with the scene texture/material database. To understand this fully, it makes sense for us to look at the CTriMesh class first and later examine the code to the CScene class.

Implementation Goals:

The first design issue to be tackled is texture loading. It does not make sense for the mesh object to manage the loading of textures as the scene or application class is typically the object aware of the limitations of the device. In our demo applications, the scene class is aware of the available texture formats compiled from the device enumeration process. The scene class will therefore be responsible for loading and storing any textures that our mesh objects will use.

Allowing the scene to store and manage the textures used by the mesh is not a trivial design decision. The D3DXLoadMeshFromX function, which our wrapper class will use internally, does not perform any texture loading on our behalf. It returns an array of D3DXMATERIAL structures describing the texture filename and material properties used by each subset in the mesh. We know for example, that the first element in this array contains the texture filename and the material that should be used to render subset 0. The second element in this array contains the attribute properties for subset 1 and so on.

A naive first approach might be to place some code in the mesh wrapper class that, on return from the D3DXLoadMeshFromX function, simply loads all textures referenced in this array. While it is true that every subset in a mesh uses a different texture/material pair, it is probable that multiple subsets in the mesh will use the same texture. In this instance, multiple D3DXMATERIAL structures would contain the same texture filename. Blindly loading the texture referenced by each attribute in this array would cause the same texture to be loaded multiple times. Video memory is a precious resource and we certainly cannot accept this wasteful outcome.

To solve this problem we will expose a function in our mesh class that will allow an external object (our scene class in particular) to register a texture loading callback function. After the scene has instantiated a CTriMesh (but before it has loaded any data into it), it will register a texture loading function with the mesh. The mesh will store a pointer to this function for use when loading data from an X file. When the application calls the CTriMesh::LoadFromX function, the function will first load the geometry using the D3DXLoadMeshFromX function. Then, the mesh object will loop through each of the attributes returned from D3DXLoadMeshFromX and pass each texture filename to the texture loading callback function. The callback (which in our application is a static method of the CScene class), will then check its internal texture list to see if the texture has been previously loaded. If it has, then the function will simply return the texture pointer back to the mesh class for storage. If it has not been loaded, then the CScene class will load the texture using a suitable format and add it to its internal texture database. This pointer is then returned back to the mesh object for storage. After the loading process, a mesh object will thus maintain an attribute array internally which contains one element for each subset in the mesh. Each

element will contain a D3DMATERIAL9 structure and a pointer to an IDirect3DTexture9 interface. The original textures are managed and stored at the scene level.

Using this mechanism, we will solve our redundant texture problem. If multiple subsets use the same texture, their corresponding elements in the internal attribute array will simply contain the same pointer to a texture in the scene texture list. This is true even if multiple meshes use the same texture. They will all contain pointers to the same scene owned texture.

This system provides us with a way to make the mesh essentially self-managed when it comes to rendering since it maintains an internal list of attributes (texture pointer and material) for each subset. This means the mesh object has all the information it needs to render all of its subsets without intervention from the application. The CTriMesh::Draw function will be simple to write. It will loop through each attribute in the attribute array, set the texture and material contained therein, and then use the ID3DXMesh::DrawSubset function to render the appropriate subset in the underlying ID3DXMesh.

This system of self-management makes the mesh class very easy to use. Meshes can be loaded without wasting memory on redundant textures. And, with a single function call, the mesh can be batch rendereded efficiently with mesh triangles that are optimized and sorted by attribute.

This type of self-contained object that requires little external intervention is great for rapid prototyping. It allows an application (such as a mesh viewer) to load and render the mesh with only a few function calls. Each mesh is a self-contained object which will manage its own rendering and therefore is unaware of other mesh objects in the scene. Of course, while easy and intuitive, this approach is not always ideal for all situations, so our mesh class will need to expose an additional mode of operation. The purpose of this additional mode will be to make the mesh more state change friendly when the scene contains multiple meshes.

The importance of batch rendering to minimize state changes has been discussed many times in our studies together. The problem with the self-contained rendering system discussed thus far is that it fails to account for other objects in the scene which may share the same states. It is certainly not uncommon for two or more unique meshes to have cases of identical texture and/or material information at the subset level. For optimal performace, we might want to render all subsets from all meshes which share the same attributes together, to minimize texture and material state changes. This would allow us to batch render across mesh boundaries.

In order for this additional mode (non-managed mode) to allow for state batching across mesh boundaries, the task of rendering the mesh can no longer reside in the mesh class itself. It must now be handled by a higher level object, such as the scene, which can maintain a global list of all attributes (texture and material pairs) used by all meshes in the scene. In this mode, the texture loading callback mechanism will be disabled in favor of another callback function. The new function will also belong to the scene class and will be registered with the mesh. It will accept the complete set of attribute data returned from the D3DXLoadMeshFromX function; not just the texture filename. In this case, the mesh object will not maintain the list of attributes used by each of its subsets; the scene will maintain a global list of attributes used by all meshes.

In order for all meshes to reference the same shared list of attributes, a little remapping of mesh attribute buffers is necessary. This will occur after the mesh is loaded and its textures and materials are registered with the scene. To understand why this is required we must remember that each mesh has its own zero based attribute buffer. While two meshes might each have a subset that both use the same texture and material pair, in the first mesh it may be subset 0 and in the second mesh it may be subset 10.

So we need a way for all of the meshes in the scene to have their attributes remapped. If any meshes have subsets that share the exact same attributes, then we should make sure that both of these subsets use the same attribute ID. An attribute ID will then have global meaning across the entire scene. It is fortunate that attribute IDs within a given mesh do not have to be zero based, or even consecutive, otherwise we would be forced to work with local attributes. If we know that after the remapping process, subset 5 describes the same set of attributes across all meshes, the scene can render the matching subset in all meshes together to reduce state changes. The scene will not loop through each mesh and then render each subset as before, as this would require setting the texture and material states for every subset for every mesh. Instead, the scene will loop through each attribute ID in its internal array and then render any subsets from any mesh that share that attribute together. This way, we set the texture and material for a given scene attribute only once.

The task of remapping the attribute buffer to a global list of attributes is performed when the mesh is first loaded. Our mesh object will pass the attribute data (texture filename and material) returned from the D3DXLoadMeshFromX function to the attribute callback function. This function will search an array of attributes to see if the texture/material data passed in already exists. If so, then its index in the array is returned to the mesh class and becomes the new attribute ID for the subset in question. If a matching attribute cannot be found in the scene attribute array, a new texture/material pair is added to the attribute list and the index of this new element is returned to the mesh. As before, the mesh object can then lock its attribute buffer and alter its contents so that the subset now uses this global attribute ID instead.

To better visualize this process, think of a freshly loaded mesh just about to process the attribute for subset 5. The texture and material used by subset 5 is passed by the mesh into the scene's attribute callback function, where the scene object determines that a matching texture and material already exists in its attribute array at element 35. This means that at least one mesh that was previously loaded used this same attribute combination for one of its subsets. The value of 35 would be returned from this function back to the mesh object. The mesh can now lock the attribute buffer and search for all entries with a value of 5 and replace them with the new global attribute value of 35. Using this method, we can rest assured that all meshes sharing the same attribute data, now share the same global attribute ID for the subsets that reference it.

To summarize, we have achieved some important design goals for our mesh class by envisioning a class that supports two different operational modes:

- ***Managed Mode***: In this mode the mesh object manages its own rendering and setting of states. The mesh maintains an internal attribute array which contains a texture and material pointer for each subset in the mesh. An external texture callback function is used for the loading of texture

filenames returned from the D3DXLoadMeshFromX function. The mesh can be rendered in its entirety with a single function call from the application.

Pros *: With the exception of the texture callback function, the mesh is self contained. This makes it very easy to setup and render. This mode is ideal for simple applications and prototyping. In this mode the mesh is very easy to plug into existing applications, requiring virtually no support code to be written to access its features. The obvious exception to this is the texture callback function which will need to be implemented and registered with the mesh object if we wish textures to be loaded from the X file.*

Cons *: The mesh object is not aware of other meshes in the scene and therefore batch rendering across mesh boundaries is not possible. This results in ineffiecient rendering practices when many meshes are in the scene which share matching attributes. This mode is not ideal for using meshes in a typical multi-mesh game environment. Essentially, this mode's greatest strength is in many ways its greatest weakness. By completely encapsulating the rendering of the mesh, the supporting application is denied the chance to use a mesh rendering strategy that it considers optimal.*

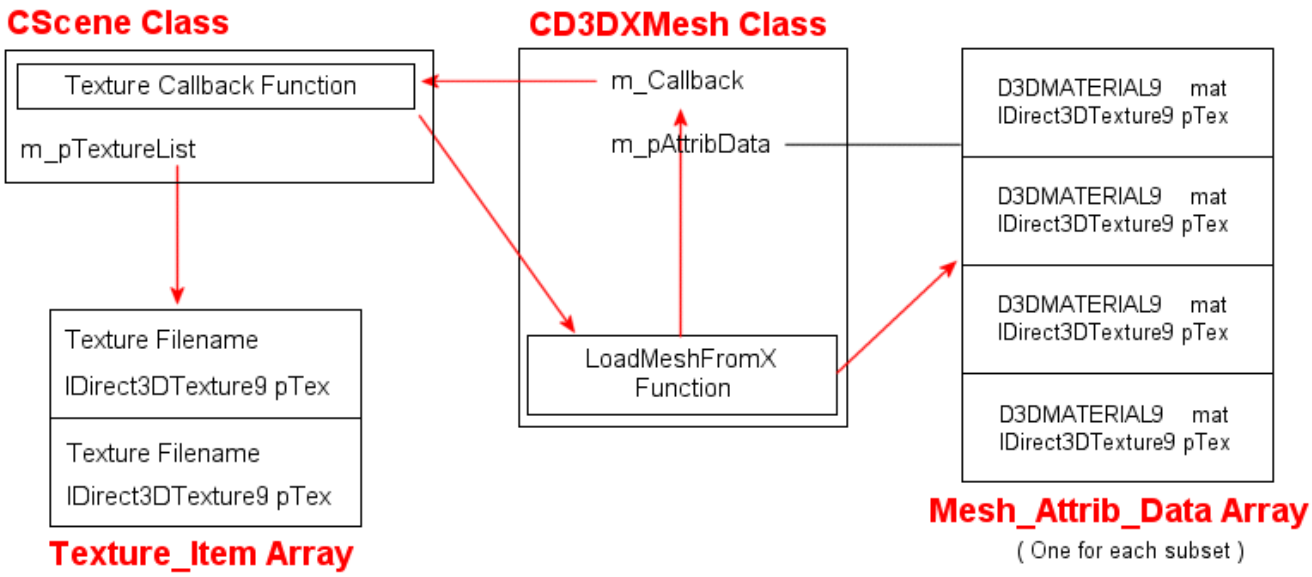
- ***Non-Managed Mode***: In this mode the mesh object no longer maintains an internal list of attributes used by each of its subsets, and is thus incapable of rendering itself. Instead, the meshes pass all attribute data loaded from an X file to an external object (such as our scene class) via an attribute callback function. The scene maintains the attribute list and is responsible for rendering each of the mesh subsets individually. The scene will also be responsible for setting device states before rendering any subset of a given mesh. Attribute data is passed to the scene for registration during X file loading via a callback which returns a global attribute ID for that subset. The mesh will then remap its attribute buffer so that any triangles belonging to the old subset now have attribute IDs matching the newly returned global ID.

Pros *: Multiple meshes in the scene can have subsets that share the same attribute ID. This attribute ID is actually the index of the texture and material pair stored in the scene's attribute array. The scene can batch render across mesh boundaries. All subsets across all meshes that share the same global attribute ID can be rendered together, minimizing state changes between each render call. This mode is much more efficient for a game scene that contains multiple meshes indexing into a global list of attributes. The mesh object itself is ultimately more flexible in this mode because it does not force any rendering semantics on the application. The application is free to render the various subsets of all meshes in the scene how it sees fit.*

Cons *: The only real downside for this mode is that more support code must be implemented in the application. This can make integration of the object into new or existing applications a more time consuming affair. The application must supply the rendering logic and maintain and manage the internal attribute list.*

To better visualize the two modes of operation discussed, some diagrams are presented in the next section. They should aid us in seeing the implementations for the mesh in each of these modes.

Managed Mode



Managed mode is the default mode for the class. As seen above, the CScene class maintains an array of Texture_Items. Each texture item contains the filename and texture pointer for a texture that has already been loaded by a mesh in the scene. This array will be empty prior to any meshes having been instantiated.

The CScene class also has a texture callback function which will be registered with the mesh object. This function pointer will be stored in the mesh so that it can be called by the mesh object during X file loading. The CTriMesh::LoadMeshFromX function is exposed to facilitate the loading of X files. This function will initially use the D3DXLoadMeshFromX function to create the underlying D3DXMesh object and initialize its geometry. The mesh object will then loop though each D3DXMATERIAL structure returned from the D3DXLoadMeshFromX function and will pass the texture filename stored there to the texture callback function. If the texture is not already loaded then it will be and added to the end of the scene's Texture_Item array.

The texture callback function will return a texture pointer back to the mesh object to be stored in the mesh's Mesh_Attrib_Data array. Each element in this array contains a material and a texture pointer for a subset in the mesh. The material data is simply copied straight over from the properties returned from D3DX in the D3DXMATERIAL array. The texture pointer is issued to the mesh by the scene object's texture loading callback.

The diagram shows the core components for the mesh in managed mode so we can see that, because the mesh contains its own attribute list, it has everything it needs to render itself. In order for the mesh to render itself in its entirety, simple code can be employed:

```
void CTriMesh::Draw()
{
    // This function is invalid if there is no managed data
    if ( !m_pAttribData ) return;

    // Render the subsets
    for ( ULONG i = 0; i < m_nAttribCount; ++i )
    {
        // Set the attribute data

        pD3DDevice->SetMaterial( &m_pAttribData[i].Material );
        pD3DDevice->SetTexture( 0, m_pAttribData[i].Texture );

        // Otherwise simply render the subset(s)
        m_pMesh->DrawSubset( i );

    } // Next attribute
}
```

The CTriMesh::Draw function will loop through each subset contained in the mesh and manage the setting of the states stored in its attribute array. Then it calls the ID3DXMesh::DrawSubset function, passing in the number of the subset we wish to render. The ID3DXMesh::DrawSubset function renders all triangles in the underlying D3DXMesh with a matching attribute ID.

In this self-contained mode, any burden for rendering the mesh is largely removed from the application. We might imagine that if a scene has many meshes that are all set to operate in managed mode, the code to render those meshes would be a single call to the CTriMesh::Draw method for each mesh in the scene.

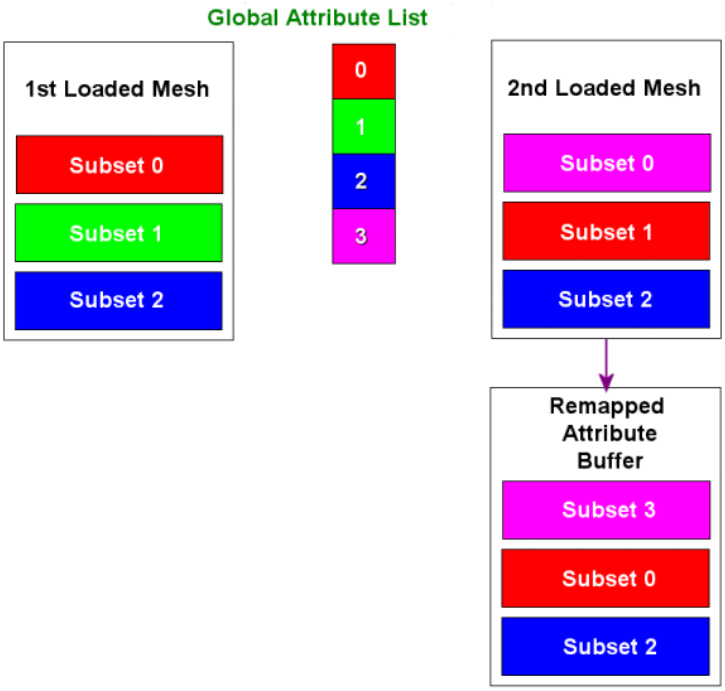
```
for(int i = 0; i < m_NumberOfMeshes; i++)
{
  Meshes[i]->Draw();
}
```

It is the self-contained nature of the managed mode mesh that makes it so quick and easy to plug in and use in existing applications.

Non-Managed Mode

To batch by state across mesh boundaries, we give more rendering control to the scene. In non-managed mode, the mesh does not store an attribute array (this pointer is set to NULL) -- it simply passes the texture/material information loaded from the file to the scene and forgets about it. Subsets will be rendered by the scene using the CTriMesh::DrawSubset function. This function does not update texture or material states and the scene class render method will assume this responsibility. The mesh subset attribute IDs will be remapped to reference a global set of attributes maintained by the scene. This means

that two different meshes that contain matching subset attributes (same texture, same material) can both be remapped to reference the same attribute IDs in the global attribute table. This is demonstrated in the following diagram.

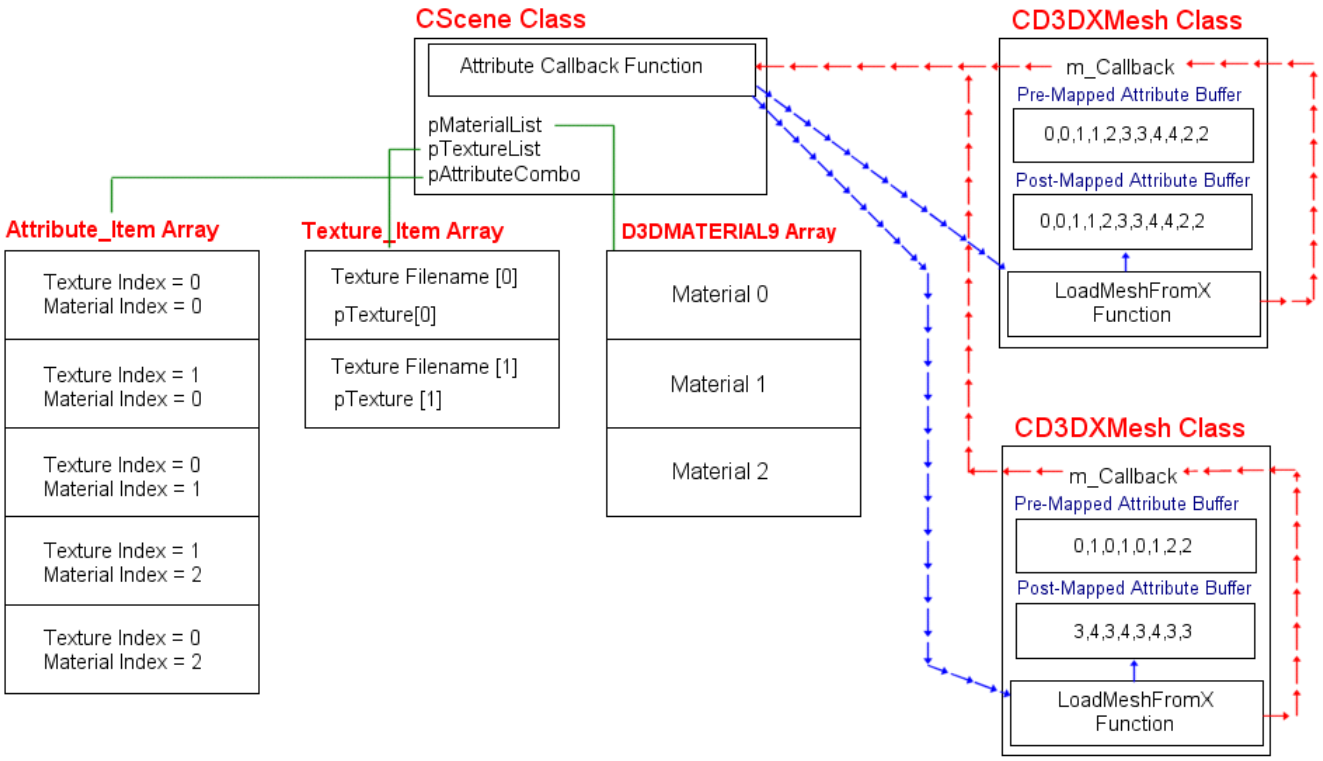


The above image shows how two meshes might initially be loaded. The first and second mesh both have three subsets each. Between both meshes, there are four unique attributes being used. We can see that (using color coding for this example) both the first and second mesh have a red attribute and both also have a blue attribute. The green subset is unique to the first mesh and the purple is unique to the second. We also see that the global attribute list, managed by the scene object, contains those four unique attributes in an ordered array.

While both the first and second mesh contain a red subset, each mesh has a zero based attribute buffer. The attribute buffer values returned from the D3DXLoadMeshFromX function have been assigned differently for the red subset in both meshes. The red material has an attribute ID of 0 in the first mesh and 1 in the second. In order for the scene to render these two subsets together they must both be assigned the same attribute ID. It makes sense to use the index at which the attribute is stored in the global attribute array for this purpose. We can see that the red attribute is stored in this array in position 0. This means any meshes which reference this attribute must use an attribute ID of 0.

The first mesh is basically unaffected because it initially used attribute ID 0 for its red subset. The second mesh however will need to have its attribute buffer locked and altered. Any faces in the red subset of this mesh must have their corresponding attribute ID mapped from 1 (original value when loaded by D3DX) to 0 (the global ID).

The following diagram shows how this model is implemented in our CTriMesh class.



This diagram shows the interaction between the scene and the mesh objects in non-managed mode. The scene class now has to maintain all attribute information used by meshes in the scene; not just textures. The CScene class therefore contains a material array, a texture item array and an attribute combo array. The Texture_Item array is the same one used when dealing with meshes in managed mode. It contans a texture filename, used for testing whether a requested texture has been loaded already, and its accompanying texture pointer. The scene also contains an array of D3DMATERIAL9 structures which will contain all the materials used by all meshes in the scene. These two arrays are maintained by the scene class and logic is put in place to assure that multiple copies of the same texture or material are not loaded more than once. The Attribute_Item_Data array contains what is essentially a scene attribute.

When a non-managed mesh has a subset with an attribute ID of 5 for example, it means the texture and material contained in the 6th element in this array should be used to render that subset. Each element in the scene's attribute array contains a material index and a texture index. These indices are lookups into the Texture_Item array and the D3DMATERIAL9 array. The scene attribute array is analogous to the attribute array that is maintained internally by the mesh object when placed in managed mode. The difference is that this contains the global attribute array for all meshes used in the scene.

In order to place a mesh in non-managed mode we must register an attribute callback function with the mesh, much as we do in managed mode when we register a texture callback function. Registration of this attribute function informs the mesh that it is to be used in non-managed mode. If an attribute callback

has not been registered, the mesh will operate in managed mode and will try to use the texture callback function instead (if one has been registered). The attribute callback function must be registered with the mesh *before* geometry is loaded.

Pay special attention to the two mesh objects shown in the previous diagram as you can see both the pre- and post-mapped attribute buffers. The pre-mapped attribute buffer represents what the mesh looked like just after the D3DXLoadMeshFromX function had been called. Each item is zero based as expected. After the mesh data has been loaded, the mesh object then loops through each attribute and calls the attribute callback function, passing in both the texture and material of each subset. The scene will add the texture and material to its internal arrays and return the index of the corresponding Attribute_Item for that texture/material pair. The attribute buffer of the mesh is then remapped using this information.

Notice how the first mesh that is loaded is also the first to add attributes to the scene. Therefore, the global indices returned for each subset from the callback are the same as the original attribute IDs in the attribute buffer. The second mesh loaded however, has its attribute buffer changed significantly. This is because the attributes it requires were also required by the first mesh and already exist in the scene attribute array.

Once all meshes have been loaded and are known to reference the same global list of attributes, the scene object can render the meshes using whatever method it considers efficient. In the following example we see some code that might be employed by the scene's render function to draw all meshes with minimized texture and material state changes. Batching across mesh boundaries in this fashion is not possible in managed mode.

```
for ( i = 0 ; i < m_TotalNumberOfSceneAttributes; i ++)
{
        pDevice->SetTexture(0, pTextureList[pAttributeCombo[i].TextureIndex]);
        pDevice->SetMaterial(&pMaterialList[pAttributeCombo[i].MaterialIndex]);

        for ( t = 0 ; t < m_NumberOfMeshes ; t ++ )
        {
              m_pMeshes->DrawSubset( i );
        }
}
```

Note that one thing missing from the above code is any mention of the world transform. This is also a device state that must be set.

Often, batching by attribute across mesh boundaries will provide an increase in performance, but this is not always guaranteed. In fact, it is possible that the opposite will happen. If meshes are defined in world space and do not need to be transformed, then batching by attribute across mesh boundaries will provide an increase in performance and is certainly a worthy goal. It is probable however that many of your meshes will be defined in model space and will need to have a world matrix set for them before rendering any of their subsets. If we continue to batch by attribute when this is the case, we are actually increasing the number of times the world matrix has to be set for a mesh. Under normal circumstances, the world matrix would need to be set only once and then the entire mesh rendered. But since attribute

batch rendering requires that we no longer render complete meshes in isolation, the total number of SetTransform calls made per mesh will be equal to its number of subsets:

```
for ( int i=0 ; i<NumberOfAttributes; i ++ )
{
        pDevice->SetTexture ( 0 , pTextureList [ pAttributeCombo[i].TextureIndex]  );
        pDevice->SetMaterial( &pMaterialList  [ pAttributeCombo[i].MaterialIndex] );

        for ( int t =0 ; t < NumberOfMeshes ; t++ )
        {
               pDevice->SetTransform ( D3DTS_WORLD , &pMeshes[t]->Matrix );
               pMeshes[t]->DrawSubset[i];
        }
}
```

SetTransform is a very costly operation and it should not be called superfluously. Usually, the 3D pipeline will try to intelligently predict what information the 3D hardware will need next in its buffers. As a result, the pipeline will often have one or more frames beyond the one you are currently rendering cued up in advance. When the application alters any of its state transform matrices, the pipeline has to flush its buffers. This causes a wait state or stall in the pipeline. In some tests we conducted here in the Game Institute labs, frame rates actually decreased by as much as 50% when batching by attribute on certain scenes because each mesh required multiple world transform sets. This will not always be the case and our non-managed mode mesh provides total flexibility by allowing the scene to choose its preferred rendering strategy. The scene has the ability now to choose how it wishes to batch, by attribute or by transform state.

The scene could choose to render its meshes in the following manner which would assure transform batching over attribute batching. This method would most likely perform better in a scene containing many dynamic meshes.

```
for ( int t = 0 ; t < NumberOfMeshes ; t++ )
{
        pDevice->SetTransform ( D3DTS_WORLD , &pMeshes[t]->Matrix );

        for ( int i=0 ; i<NumberOfAttributes; i ++ )
        {
               pDevice->SetTexture ( 0 , pTextureList [ pAttributeCombo[i].TextureIndex]  );
               pDevice->SetMaterial( &pMaterialList  [ pAttributeCombo[i].MaterialIndex] );

               pMeshes[t]->DrawSubset[i];
        }
}
```

If your scene has many static meshes (which typically lend themselves to being pre-defined in world space), then it would be quicker to batch across mesh boundaries. When a scene consists of both world space meshes and meshes that need a world transform set, you could batch them into two groups and use a different rendering strategey for each group in your core rendering routine.

In our demo applcation we provide the ability for batch render either by attribute or by transform when the mesh objects are in non-managed mode (using a pre-compiler define directive). The scene class

rendering function will test if this define has been set and if so, batch by attribute. Otherwise, batching by transform will be employed.

Source Code Walkthrough

The class declaration for the CTriMesh class is contained in the CObject.h header file. In this listing we have left out the member functions and only show the member variables to improve readability. The class is shown below followed by a description of some of its important member variables.

```
class CTriMesh
{
public:
//-----------------------------------------------------------------------------
// Public Enumerators for This Class.
//-----------------------------------------------------------------------------
 enum CALLBACK_TYPE { CALLBACK_TEXTURE = 0, CALLBACK_EFFECT = 1,
                      CALLBACK_ATTRIBUTEID = 2, CALLBACK_COUNT = 3 };

private:
 //-----------------------------------------------------------------------------
 // Private Variables for This Class
 //-----------------------------------------------------------------------------
    LPD3DXBUFFER  m_pAdjacency;                // Stores adjacency information
    CALLBACK_FUNC m_CallBack[CALLBACK_COUNT];  // References the various callbacks
    LPD3DXMESH    m_pMesh;                     // Physical mesh object
    TCHAR         m_strMeshName[MAX_PATH];       // The filename used to load the mesh


 // Managed Attribute Data
    MESH_ATTRIB_DATA  *m_pAttribData;  // Individual mesh attribute data.
    ULONG              m_nAttribCount; // Number of items in the attribute data array.

 // Mesh creation data.
    ULONG          m_nVertexStride;    // Stride of the vertices
    ULONG          m_nVertexFVF;         // FVF
    ULONG          m_nIndexStride;     // Stride of the indices
    ULONG          m_nVertexCount;     // Number of vertices to use during BuildMesh
    ULONG          m_nFaceCount;       // Number of faces to use during BuildMesh
    ULONG          m_nVertexCapacity;  // We are currently capable of holding this many
                                       // before a grow
    ULONG          m_nFaceCapacity;    // We are currently capable of holding this many
                                       // before a grow
    ULONG          *m_pAttribute;      // Attribute ID's for all faces
    UCHAR          *m_pIndex;          // The face index data
    UCHAR          *m_pVertex;         // The physical vertices.

};
```

LPD3DXBUFFER m_pAdjacency
Unlike ID3DXMesh, the CTriMesh object always maintains a copy of the adjacency information. We use an ID3DXBuffer to store the adjacency data in a uniform and consistent way. Recall that some D3DX functions expect adjacency information as a DWORD array and others expect it as an

ID3DXBuffer. Because the pointer obtained when locking the buffer can be cast to a DWORD pointer, storing it in an ID3DXBuffer makes sure that we account for both cases.

CALLBACK_FUNC m_CallBack[CALLBACK_COUNT]
This is an array which will hold pointers to callback functions. The type CALLBACK_FUNC is a structure defined in the header file Main.h. It contains a void pointer to a function and a void pointer to a context. This second pointer is used to pass arbitrary information to the callback.

```
typedef struct _CALLBACK_FUNC   // Stores details for a callback
{
    LPVOID  pFunction;          // Function Pointer
    LPVOID  pContext;           // Context to pass to the function
} CALLBACK_FUNC;
```

A CTriMesh can hold up to three callback functions for use when loading X files. The first callback (m_CallBack[0]) is used only in managed mode for texture registration. It is where the texture loading callback function pointer will be stored if it has been registered by the application. When the X file is loaded in managed mode, the texture filenames used by each of the mesh attributes will be passed to this function, if it exists. The object that registered the callback with the mesh should make sure that it creates the texture if it does not exist and return a pointer to the texture interface back to the mesh. The returned texture pointer is stored in the mesh MESH_ATTRIB_DATA member. If this callback function is not registered, then no textures will be loaded for the managed mesh. This callback function is not used if the mesh is in non-managed mode and therefore does not need to be registered.

The second array entry (m_CallBack[1]) should contain a pointer to an effect file callback function. X files contain effect file references which the application can use to load the effect file. Effect files are not used in this demo and will be covered in Module III of this series. This callback is only used in managed mode and effect files contained within the X file will be ignored if it is not registered.

The third array entry (m_Callback[2]) essentially switches the mesh into non-managed mode if it contains a function pointer. When this element is not NULL, it should point to an externally defined attribute callback function. For each attribute returned from the D3DXLoadMeshFromX function, the mesh passes in the texture and material to this callback function, if it is defined. The owner of this function should add the material and texture information to its own texture and material lists if they do not already exist and return a new Attribute ID that the mesh class can use to re-map its attribute buffer to point to the global stores. Attribute IDs will then take on global meaning across mesh boundaries and allow for resource sharing. If this function is not registered, then the mesh is assumed to be in managed mode. Registering this function places the mesh in non-managed mode.

There is an enumerated type in the CTriMesh namespace that indexes these callbacks for easy registration:

```
enum CALLBACK_TYPE { CALLBACK_TEXTURE = 0, CALLBACK_EFFECT = 1,
                     CALLBACK_ATTRIBUTEID = 2, CALLBACK_COUNT = 3 };
```

The last member (CALLBACK_COUNT) indicates the total number of callbacks available to the mesh class at present (i.e., it defines the size of the callback array). This allows us to add more callbacks in the future by inserting them into the enum (before the last element) and incrementing CALLBACK_COUNT.

An external object or function that creates a mesh can set the callback function using the CTriMesh::RegisterCallback function:

```
bool   RegisterCallback ( CALLBACK_TYPE Type, LPVOID pFunction, LPVOID pContext );
```

In Lab Project 8.1, the CScene class is responsible for loading the geometry and creating the CTriMesh objects. After a new CTriMesh has been created (but before the data is loaded) the scene registers its callback function as follows:

```
// Create the Mesh
CTriMesh * pNewMesh = new CTriMesh;

// Load in the specified X file
pNewMesh->RegisterCallback(CTriMesh::CALLBACK_ATTRIBUTEID,CollectAttributeID,this);

// Load the X file into the CTriMesh object
pNewMesh->LoadMeshFromX( strFileName, D3DXMESH_MANAGED, m_pD3DDevice );
```

In this example, after we have created the mesh, the scene registers the CALLBACK_ATTRIBUTEID callback function. This places the mesh into non-managed mode. The second parameter is the address of the function that will be called by the mesh for each texture/material combo returned when the X file is loaded. The CScene class includes a function called CScene::CollectAttributeID which accepts a texture and a material and returns a global attribute ID which the mesh can then use to re-map its attribute buffer. This is the also function that is responsible for both creating the texture and adding the texture and the material to the scene database. In this example we do not have to register the texture or effect callback functions since they are only used with managed meshes. Note that the scene passes the 'this' pointer so that it is stored in the context and can be passed to the callback function when called from the mesh. We need to do this because the callback function is a static function shared by all instances of the CScene class. As such, the function can only access static variables. The 'this' pointer allows us to circumvent that problem as we saw in earlier lessons and access the non-static member variables of the instance of the CScene object being used.

The three callback functions each take different parameter lists and return different values. The function signatures are typedef'd at the top of the file CObject.h and are shown below. If you register any of these callback functions, you must make sure the functions you write have the the exact signatures that the mesh expects.

```
typedef LPDIRECT3DTEXTURE9    (*COLLECTTEXTURE )( LPVOID pContext, LPCSTR FileName );

typedef LPD3DXEFFECT          (*COLLECTEFFECT)(LPVOID pContext,
                                               const D3DXEFFECTINSTANCE & EffectInstance );
```

```
typedef ULONG                (*COLLECTATTRIBUTEID)( LPVOID pContext,
                                                    LPCSTR strTextureFile,
                                                    const D3DMATERIAL9 *pMaterial,
                                                    const D3DXEFFECTINSTANCE * pEffectInstance);
```

Continuing our discussion of the CTriMesh member variable…….

LPD3DXMESH m_pMesh
This is a pointer to the ID3DXMesh wrapped by this class.

TCHAR m_strMeshName[MAX_PATH]
This member will hold the filename of the X file used to load the mesh data.

MESH_ATTRIB_DATA *m_pAttribData
When the mesh is in managed mode (i.e. the attribute callback function has not been registered) the mesh will maintain an array of MESH_ATTRIB_DATA structures -- one for each subset. Each structure describes the texture and material used by the corresponding subset. The structure is defined in the header file CObject.h and is shown below for convenience.

```
typedef struct _MESH_ATTRIB_DATA
{
    D3DMATERIAL9        Material;
    LPDIRECT3DTEXTURE9  Texture;
    LPD3DXEFFECT        Effect;
} MESH_ATTRIB_DATA;
```

When the mesh is in non-managed mode, this pointer will be NULL and no texture and material information will be managed by the mesh.

There are two ways that this array can be populated in managed mode. When using the CTriMesh::LoadMeshFromX function, this array will automatically be populated with the material information in the X file and the texture pointers returned from the texture callback function. If we are building the mesh ourselves, then we use the AddVertex, AddFace and AddAttributeData functions to populate the mesh buffers. The AddAttributeData function makes new space at the end of the MESH_ATTRIB_DATA array (in managed mode) which we can then populate with the material and texture information we desire. When rendering the mesh in managed mode (CTriMesh::Draw), this array is used to set the texture and the material for each subset.

ULONG m_nAttribCount
This value describes how many attributes are in the MESH_ATTRIB_ARRAY. This is only used in managed mode and should always be equal to the number of subsets in the mesh.

ULONG m_nVertexStride
ULONG m_nVertexFVF
ULONG m_nIndexStride
These three members store vertex stride, FVF, and index stride (16 vs. 32-bit) information. Our application will set these as soon as it creates a mesh (before calling any loading functions). We set this information by calling CTriMesh::SetDataFormat:

```
void   SetDataFormat ( ULONG VertexFVF,  ULONG IndexStride);
```

SetDataFormat copies over the information into the internal class variables and uses the specified FVF flags to calculate the vertex stride. This function is only used when creating a mesh from scratch. You must make sure you call it before adding any data to the mesh, otherwise mesh creation will fail. We do not call this function when we create the mesh using CTriMesh::LoadMeshFromX because the vertex and index formats will be taken from the X file. Recall that the D3DXLoadMeshFromX function will set the FVF of our vertices when the mesh is created based on the vertex components available in the X file.

UCHAR *m_pVertex
ULONG m_nVertexCount
We simplify the manual creation of mesh data by maintaining temporary system memory vertex, index, and attribute arrays. This allows an application to accumulate data in these arrays and only build the actual ID3DXMesh once all arrays are finalized. Once the mesh is created, these arrays can be freed since the data is no longer needed. The m_nVertexCount member describes the current number of vertices in the temporary array and does not necessarily describe the number of vertices in the underlying mesh. This is especially true after the underlying mesh has been optimized. These variables are not used at all if we create the mesh data using CTriMesh::LoadMeshFromX.

UCHAR *m_pIndex
ULONG *m_pAttribute
ULONG m_nFaceCount
These are also temporary storage bins that will be used during manual mesh filling. m_pIndex will contain three indices per face (m_nFaceCount*3) and the m_pAttribute array will hold m_nFaceCount DWORD values. These are the per-face attribute IDs. These members are not used if the mesh data is created using the CTriMesh::LoadMeshFromX function. When we manually add faces to the mesh we also specify the attribute ID of the face about to be added. The indices and attributes passed for each face are stored in these temporary bins and later used to populate the actual index and attribute buffers of the ID3DXMesh.

ULONG m_nVertexCapacity
ULONG m_nFaceCapacity
These member variables are only used during manual mesh creation. When we use the AddVertex method to add a vertex to the m_pVertex array, the array will need to be resized to accommodate the new addition. Array resizing can be slow, so to avoid doing it for single vertex additions, we use the m_nVertexCapacity member to define a resize threshold. For example, the array will be resized only

after every 100 vertices have been added. If we start with a vertex capacity of 100, we only need to resize the array when we add the 101st vertex. We then resize again by 100 and increase m_nVertexCapacity by 100 also. We can then once again add another 100 vertices before the new vertex capacity of 200 is reached. The vertex array is resized again by 100, and so on. While it might seem wasteful to resize the array by 100 when we may only need to add 1 vertex, this memory is freed as soon as the underlying ID3DXMesh object is created. Thus the memory footprint is only temporary and it does greatly speed up manual mesh creation. m_nFaceCapacity works in precisely the same way but with the index and attribute arrays. These member variables are not used once the underlying ID3DXMesh has been created.

The Methods

Many of the CTriMesh functions are just wrappers around their ID3DXMesh counterparts (e.g., the functions which optimize and render the underlying mesh data). Some may be slightly more complex than others due to the management modes the class supports. This is certainly true in the managed mode code sections where asset management code is introduced to cater for its self-contained design. We will look at the list of the methods next, but only discuss source code for non-wrapper functions where the code inside the function is new or significant.

CTriMesh::CTriMesh()

The mesh class has a default constructor that takes no parameters and initializes all internal variables to NULL or zero.

```
CTriMesh::CTriMesh()
{
    // Reset Variables
    m_pAdjacency      = NULL;
    m_pMesh           = NULL;
    m_pAttribData     = NULL;
    m_nAttribCount    = 0;
    m_nVertexCount    = 0;
    m_nFaceCount      = 0;
    m_pAttribute      = NULL;
    m_pIndex          = NULL;
    m_pVertex         = NULL;
    m_nVertexStride   = 0;
    m_nIndexStride    = 0;
    m_nVertexFVF      = 0;
    m_nVertexCapacity = 0;
    m_nFaceCapacity   = 0;

    ZeroMemory( m_strMeshName, MAX_PATH * sizeof(TCHAR));

    // Clear structures
    for(ULONG i = 0; i < CALLBACK_COUNT; ++i )
        ZeroMemory( &m_CallBack[i], sizeof(CALLBACK_FUNC) );
}
```

Notice how CALLBACK_COUNT is being used to initially clear the function callback array. Initially no callback functions are registered so the mesh is assumed to be in managed mode at this point. However, textures and effect files would be ignored during loading unless a callback is given.

CTriMesh::~CTriMesh()

The destructor calls CTriMesh::Release to free the internal memory used by the object.

```
CTriMesh::~CTriMesh()
{
    Release();
}
```

CTriMesh::Release

The Release function manages the freeing of memory resources used by the mesh. This allows us to release the underlying data and reuse the same CTriMesh object to create another CTriMesh object if desired. This function calls Release for any COM objects it is currently managing so that reference counts are decremented as expected. This allows the COM layer to unload those objects from memory if no other references to those objects remain outstanding.

If the mesh has an internal attribute array defined (which is only the case if the mesh is operating in managed mode), then the reference count of any textures and effect files contained therein are decremented and the attribute array deleted. It is likely that the texture objects will not be released from memory at this point since the scene object will also have a reference to this object. The same texture resource may also be referenced by other meshes in the scene. Only when all meshes have released their claim to a texture and the scene itself releases it, will the reference count hit zero and the texture be unloaded from memory. It is especially important that our mesh class abides by the COM reference counting convention and is well behaved in this regard. If we did not make sure that the reference count of a texture was increased and decreased correctly when a mesh object gains or releases its claim to a texture resource, we might end up with a scenario where meshes in the scene still have pointers to textures which no longer exist.

```
void CTriMesh::Release( )
{
    // Release objects
    if ( m_pAdjacency ) m_pAdjacency->Release();
    if ( m_pMesh      ) m_pMesh->Release();

    // Release attribute data.
    if ( m_pAttribData )
    {
        for ( ULONG i = 0; i < m_nAttribCount; i++ )
        {
            // First release the texture object (addref was called earlier)
            if ( m_pAttribData[i].Texture ) m_pAttribData[i].Texture->Release();
```

```
            // And also the effect object
            if ( m_pAttribData[i].Effect  ) m_pAttribData[i].Effect->Release();

        } // Next Subset

        delete []m_pAttribData;

    } // End if subset data exists

    // Release flat arrays
    if ( m_pVertex    ) delete []m_pVertex;
    if ( m_pIndex     ) delete []m_pIndex;
    if ( m_pAttribute ) delete []m_pAttribute;

    // Clear out variables
    m_pAdjacency      = NULL;
    m_pMesh           = NULL;
    m_pAttribData     = NULL;
    m_nAttribCount    = 0;
    m_nVertexCount    = 0;
    m_nFaceCount      = 0;
    m_pAttribute      = NULL;
    m_pIndex          = NULL;
    m_pVertex         = NULL;
    m_nVertexStride   = 0;
    m_nIndexStride    = 0;
    m_nVertexFVF      = 0;
    m_nVertexCapacity = 0;
    m_nFaceCapacity   = 0;

    ZeroMemory( m_strMeshName, MAX_PATH * sizeof(TCHAR));
}
```

Note that we do not reset the callback array to zero. It may well be the case that you want to free the internal data but re-use the class to create another mesh. In this case, you will probably want to use the previously registered callback functions to load a new mesh.

Loading Mesh Data

Once we have a new CTriMesh object, there are two ways to populate it. We can either load the data from an X file using CTriMesh::LoadMeshFromX or we can use the AddVertex, AddFace, and AddAttributes functions to add the data to the mesh manually. In the latter case we would call CTriMesh::BuildMesh to create the final ID3DXMesh. Because the two methods of mesh population are quite different, we will cover them in two sections.

A. Loading Data from X Files

If we intend to load the mesh from an X file, then before calling CTriMesh::LoadMeshFromX we will want to register one or more callback functions to handle asset management. The following example demonstrates texture callback registration for a managed mode mesh. Registering this function will have no effect for a non-managed mode mesh since the attribute callback function will be called instead in that case.

```
CTriMesh MyMesh;
MyMesh.RegisterCallback ( CTriMesh::CALLBACK_TEXTURE , MyFunction ,  pSomeInfo );
```

pSomeInfo is a pointer to arbitrary data (or NULL) that we wish the mesh class to pass to the callback function when it is called. We will look at the code to this function next.

CTriMesh::RegisterCallback

The mesh class allows for registration of three types of callback functions to handle asset management requirements. When instructed to load an X file, the mesh can use these functions to pass texture names and material properties to the external object that registered the callback. In our application, the CScene class is responsible for loading and storing textures and making sure that we do not load redundant texture copies.

```
bool CTriMesh::RegisterCallback( CALLBACK_TYPE Type, LPVOID pFunction, LPVOID pContext )
{
    // Validate Parameters
    if ( Type > CALLBACK_COUNT ) return false;

    // You cannot set the functions to anything other than NULL
    // if mesh data already exists (i.e. it's too late to change your mind :)
    if ( pFunction != NULL && m_pAttribData ) return false;

    // Store function pointer and context
    m_CallBack[ Type ].pFunction = pFunction;
    m_CallBack[ Type ].pContext  = pContext;

    // Success!!
    return true;
}
```

This function is passed a member of the CALLBACK_TYPE enumerated type (CALLBACK_TEXTURE, CALLBACK_EFFECT or CALLBACK_ATTRIBUTEID) indicating the callback function being set. The first two types are used if the mesh is in managed mode. Registering the CALLBACK_ATTRIBUTEID function places the mesh object into non-managed mode. When in non-managed mode we do not need to register the texture or effect callback functions as they will never be called. Note that CALLBACK_TYPE also serves as the callback array index of each function pointer.

The second parameter is a pointer to the callback function. It will be stored in the callback array. The third parameter allows the external object that is registering the callback to store arbitrary data that it wants passed to the callback function on execution. Because our callback functions are static, the CScene class will pass the ‘this’ pointer as the context to ensure access to non-static members.

CTriMesh::LoadMeshFromX

Once we have registered our callback functions, we call LoadMeshFromX to populate the mesh with data from an X file. The following example shows how this function might be called. Again, it is important to call this function only after callback registration so that attribute data is handled properly.

```
MyMesh.LoadMeshFromX( strFileName, D3DXMESH_MANAGED, m_pD3DDevice );
```

The function takes three parameters: a string containing the filename of the X file, a DWORD containing our desired mesh creation flags, and a pointer to the device that will own the mesh resources. The mesh creation flags are a combination of zero or more D3DXMESH flags that we use when creating an ID3DXMesh. These flags are passed straight to the D3DXLoadMeshFromX function to control properties such as the memory pools used for the vertex and index buffers.

Note as well that we do not need to specify vertex stride, FVF, or index size since the X file will provide that information. If the end result is not to our liking, we can always clone the CTriMesh to a different format after it has been loaded.

> **Note:** Due to the lengthy nature of this function, we will leave out error checking to compact the listing. The complete version can be found in the source code that accompanies this lesson.

```
HRESULT CTriMesh::LoadMeshFromX( LPCSTR pFileName, DWORD Options, LPDIRECT3DDEVICE9 pD3D )
{
    HRESULT      hRet;
    LPD3DXBUFFER pMatBuffer;
    LPD3DXBUFFER pEffectBuffer;
    ULONG        AttribID, i;
    ULONG        AttribCount;
    ULONG       *pAttribRemap  = NULL;
    bool         ManageAttribs  = false;
    bool         RemapAttribs   = false;
```

The first thing the function does is allocate some local variables. It initially assumes the mesh is in non-managed mode by setting the ManageAttribs Boolean to false. We also declare two ID3DXBuffer interface pointers that will passed into D3DXLoadMeshFromX to be filled with material and effect data from the X file.

Since it is possible that the user may have called this function for an object that already has mesh data defined, we call the Release member function (shown earlier) to free up any potential old data.

```
    // Release any old data
    Release();
```

Next we call D3DXLoadMeshFromX with the appropriate parameters (filename, options, etc.). Since we will maintain adjacency data, we pass in the module level CTriMesh::m_pAdjacency member. If the call is successful, the final mesh will be stored in the class variable CTriMesh::m_pMesh.

```
    // Attempt to load the mesh
    D3DXLoadMeshFromX( pFileName, Options, pD3D, &m_pAdjacency, &pMatBuffer,
                       &pEffectBuffer, &AttribCount, &m_pMesh );
```

At this point the mesh has been created and it contains the geometry from the X file. The remainder of the function processes the material and texture information loaded from the file and returned from the D3DXLoadMeshFromX function. In the above code, we can see that it is the pMatBuffer variable which will contain an array of D3DXMATERIAL structures, one for each subset in the mesh. The AttributeCount local variable will contain the number of subsets in the mesh. How we process and store this returned attribute data from this point on depends on whether the mesh is in managed or non-managed mode.

The first thing we do is test wheter the CALLBACK_ATTRIBUTEID callback function is NULL. If it is, then the non-managed mode callback function has not been registered, and this is a managed mesh. In this instance we set the ManageAttribs Boolean to true. Next we store the number of subsets in this mesh in the m_nAttribCount member variable.

```
    // Are we managing our own attributes ?
    ManageAttribs = (m_CallBack[ CALLBACK_ATTRIBUTEID ].pFunction == NULL);
    m_nAttribCount = AttribCount;
```

If this is a managed mesh, then we need to allocate an array of MESH_ATTRIB_DATA structures -- one for each subset. They will be filled with the texture and material information for each subset so that the mesh can render itself using the CTriMesh::Draw function. Once the array is allocated, we initialize it to zero as shown below.

```
    // Allocate the attribute data if this is a manager mesh
    if ( ManageAttribs == true && AttribCount > 0 )
    {
        m_pAttribData  = new MESH_ATTRIB_DATA[ m_nAttribCount ];
        ZeroMemory( m_pAttribData, m_nAttribCount * sizeof(MESH_ATTRIB_DATA));
    } // End if managing attributes
```

If this is a **non**-managed mesh then we will not need to allocate an attribute array as no textures or materials will be stored in the mesh itself. We will however, potentially need to re-map the attribute IDs in the D3DXMesh attribute buffer so that they index into a global pool of resources at the scene level. Therefore, we will allocate a temporary array that will hold new attribute IDs for each mesh subset.

```
    else
    {
        // Allocate attribute remap array
        pAttribRemap = new ULONG[ AttribCount ];

        // Default remap to their initial values.
        for ( i = 0; i < AttribCount; ++i ) pAttribRemap[ i ] = i;
    }
```

By default we initialize each new attribute ID so that it matches the original attribute ID. If the mesh had five subsets for example, this array will be five DWORDS long and each element will initially contain 0

to 4 respectively. This array will be used to store the new values that each ID will have to be mapped to, which will be determined momentarily. Thus if pAttribRemap[2] = 90 after the attribute callback function has been called, this means that the third subset of this mesh uses the texture and material combination that is stored in the 91st position in the scene's global attribute list. We would then loop through the attribute buffer of the ID3DXMesh and change all attribute ID's that currently equal 2 to the new value 90.

Next we retrieve pointers to the material and effects buffers returned from the D3DXLoadMeshFromX function. Although this mesh supports effect instance parsing, we will not be using it in this lesson and effects can be ignored for the time being. Notice how we lock the buffers and cast the pointers to the correct type to step through the data in each buffer.

```
    // Retrieve data pointers
    D3DXMATERIAL       *pMaterials = (D3DXMATERIAL*)pMatBuffer->GetBufferPointer();
    D3DXEFFECTINSTANCE *pEffects   = (D3DXEFFECTINSTANCE*)pEffectBuffer->GetBufferPointer();
```

Now we can loop through each subset/attribute and parse the material data. This is handled differently depending on whether the mesh is in managed or non-managed mode. We will look at the managed case first.

```
    // Loop through and process the attribute data
    for ( i = 0; i < AttribCount; ++i )
    {
```

The material buffer contains D3DXMATERIAL structures that store a D3DMATERIAL9 structure and a texture filename. First we copy the D3DMATERIAL9 into the correct slot in our MESH_ATTRIB_DATA array. For example, if we are currently processing subset 5 in the loop, then we will copy the material into m_pAttribData[4].Material. Since the material information in an X file does not contain an ambient property, we manually set the ambient member to full reflectance (1.0f, 1.0f, 1.0f, 1.0f) after the copy. You should feel free to change this default behavior.

```
        if ( ManageAttribs == true )
        {
            // Store material
            m_pAttribData[i].Material = pMaterials[i].MatD3D;

            // Note : The X File specification contains no ambient material property.
            //        We should ideally set this to full intensity to allow us to
            //        control ambient brightness via the D3DRS_AMBIENT renderstate.
            m_pAttribData[i].Material.Ambient = D3DXCOLOR( 1.0f, 1.0f, 1.0f, 1.0f );
```

To process the texture filename stored in the D3DXMATERIAL buffer for this subset we will use the registered texture callback function. The following code shows how we define a function pointer called CollectTexture to point at the texture callback function in the mesh's callback function pointer array.The CollectTexture variable is a pointer of type COLLECTTEXTURE which we saw defined earlier as a pointer to a function with the desired function signature.

```
            // Request texture pointer via callback
            if ( m_CallBack[ CALLBACK_TEXTURE ].pFunction )
            {
                COLLECTTEXTURE CollectTexture =
                                (COLLECTTEXTURE) m_CallBack[ CALLBACK_TEXTURE ].pFunction;
```

We now call the function using this pointer, passing in the context data that was set for this callback when it was registered. In our case this will be a pointer to the instance of the CScene class that registered the function. We also pass the texture filename for the current subset being processed. The function should return a pointer to an IDirect3DTexture9 interface that will be copied into the Texture member of the mesh attribute array for this subset. In keeping with proper COM standards, we increase the interface reference count of this texture pointer as we make a copy of it.

```
               m_pAttribData[i].Texture = CollectTexture(
                                            m_CallBack[ CALLBACK_TEXTURE ].pContext,
                                            pMaterials[i].pTextureFilename );

                // Add reference. We are now using this
                if ( m_pAttribData[i].Texture ) m_pAttribData[i].Texture->AddRef();

            } // End if callback available
```

Next we need to process the CALLBACK_EFFECT callback function using the same process. We will not register an effect callback function in our application, but we have included the code for future use. When the effect callback function is registered, we pass the context and the D3DXEFFECTINSTANCE structure for this subset. This information was returned from the D3DXLoadMeshFromX function in the pEffects buffer. The effect callback function returns a pointer to an ID3DXEffect interface which is stored in the mesh attribute data array for the current subset. You can feel free to ignore the code that deals with effects until we get to Module III.

```
            // Request effect pointer via callback
            if ( m_CallBack[ CALLBACK_EFFECT ].pFunction )
            {
                COLLECTEFFECT CollectEffect =
                                (COLLECTEFFECT)m_CallBack[ CALLBACK_EFFECT ].pFunction;
                m_pAttribData[i].Effect =
                      CollectEffect( m_CallBack[ CALLBACK_EFFECT ].pContext, pEffects[i] );

                // Add reference. We are now using this
                if ( m_pAttribData[i].Effect ) m_pAttribData[i].Effect->AddRef();

            } // End if callback available

        } // End if attributes are managed
```

At the end of this loop, the managed mesh will have all texture and material information used by each subset stored in the mesh attribute data array. Thus, it can render each of its subsets (including state setting) in a self-contained manner.

If this is not a managed mesh then the chain of events is quite different. We no longer have to allocate a mesh attribute data array and we will not store the texture and material information inside the object.

Management of textures and materials is left to the caller. In order to do this, we must pass the texture filename, the material, and the effect instance to the CALLBACK_ATTRIBUTEID callback function. This function will belong to the CScene class, which maintains a list of all texture and material combinations used by all meshes in the scene. The callback will search its list for a matching attribute (i.e. texture and material combination) and if found, will return the index of this attribute set in its attribute list. If the attribute does not exist in the global list, it will be created and appended. The result is the same from the mesh's perspective -- an index is returned. We store this index in our temporary re-map array so that we can update the subset attribute ID to this new index. After the loop is complete, a non-managed mesh will have an array of re-map values describing what each attribute ID in the mesh attribute buffer should be re-mapped to.

```
        else
        {
            // Request attribute ID via callback
            if ( m_CallBack[ CALLBACK_ATTRIBUTEID ].pFunction )
            {
                COLLECTATTRIBUTEID CollectAttributeID =
                          (COLLECTATTRIBUTEID)m_CallBack[ CALLBACK_ATTRIBUTEID ].pFunction;

                AttribID = CollectAttributeID( m_CallBack[ CALLBACK_ATTRIBUTEID ].pContext,
                                               pMaterials[i].pTextureFilename,
                                               &pMaterials[i].MatD3D,
                                               &pEffects[i] );

                // Store this in our attribute remap table
                pAttribRemap[i] = AttribID;

                // Determine if any changes are required so far
                if ( AttribID != i ) RemapAttribs = true;

            } // End if callback available

        } // End if we don't manage attributes

    } // Next Material
```

We no longer need the material and effect buffers that were returned by D3DXLoadMeshFromX because the callbacks have taken care of loading and storing the resources. Therefore, they can be released.

```
    // Clean up buffers
    if ( pMatBuffer    ) pMatBuffer->Release();
    if ( pEffectBuffer ) pEffectBuffer->Release();
```

If this is a non-managed mesh and attributes require remapping, then we lock the attribute buffer, update it with the new information, and unlock. We can release the remapping data when we are finished because the modifications have now been made to the attribute buffer of the D3DXMesh.. Remember that this buffer contains an attribute ID for each triangle in the mesh.

```
    // Remap attributes if required
    if ( pAttribRemap != NULL && RemapAttribs == true )
    {
        ULONG * pAttributes = NULL;
```

```
        // Lock the attribute buffer
        m_pMesh->LockAttributeBuffer( 0, &pAttributes );

        // Loop through all faces
        for ( i = 0; i < m_pMesh->GetNumFaces(); ++i )
        {
            // Retrieve the current attribute ID for this face
            AttribID = pAttributes[i];

            // Replace it with the remap value
            pAttributes[i] = pAttribRemap[AttribID];

        } // Next Face

        // Finish up
        m_pMesh->UnlockAttributeBuffer( );

    } // End if remap attributes

    // Release remap data
    if ( pAttribRemap ) delete []pAttribRemap;
```

Finally, we store the filename of the X file that was loaded in the m_strMeshName string and return success.

```
    // Copy the filename over
    _tcscpy( m_strMeshName, pFileName );

    // Success!!
    return S_OK;
}
```

The overview of the loading process is not complete until we look at the code for the CScene class callback functions. However, the above function is all that is needed to handle resource management from the perspective of the mesh object. We will look at the CScene callback functions after we have finished examining the methods of CTriMesh.

B. Loading Mesh Data Manually

We have covered the basic loading operations involved for managed and non-managed meshes extracted from X files. Let us now examine the member functions used to manually populate a CTriMesh with vertex, index, and attribute data. An application would need to do this when the mesh data is being created programmatically. More importantly, manual mesh creation will be necessary when the application is loading data from an alternative file format (e.g., IWF, 3DS, etc.). In this section, we will discuss the manual creation methods in the order in which they are normally used so that we get a better feel for the flow of operations.

CTriMesh::SetDataFormat

When we load a mesh from an X file, D3DXLoadMeshFromX chooses the format of our vertices and indices based on the matching data stored in the X file. When creating a mesh manually, before we add any vertex or index data, we must inform the CTriMesh object of the vertex and index formats we will be using. This is the only way it will know how to create the ID3DXMesh. To do this, we call the CTriMesh::SetDataFormat function. This is usually the first function we will call after we have instantiated a mesh object.

```
void CTriMesh::SetDataFormat( ULONG VertexFVF, ULONG IndexStride )
{
    // Store the values
    m_nVertexFVF    = VertexFVF;
    m_nVertexStride = D3DXGetFVFVertexSize( VertexFVF );
    m_nIndexStride  = IndexStride;
}
```

The first parameter is the FVF flag, describing the vertex format for the mesh. The second parameter is the size of the indices (in bytes) we wish to use. This will either be 2 or 4 depending on whether we wish to use 16-bit or 32-bit indices. This function calls the global D3DX helper function D3DXGetFVFVertexSize which returns the size (stride) of a single vertex given the FVF flag passed in.

The following code snippet demonstrates how we might instantiate a mesh which we intend to use for manual data population and set its vertex and index formats:

```
CTriMesh MyMesh;
DWORD  FVFFlags = D3DFVF_XYZ  | D3DFVF_TEX1;
MyMesh.SetDataFormat ( FVFFlags , 2 );
```

Here we are creating a mesh that will have room for a 3D position vector and a single set of 2D texture coordinates stored in each vertex. We also inform the mesh object that we intend to use 16-bit indices.

CTriMesh::AddVertex

This function is used to add vertices to a temporary vertex array maintained by the mesh. It is very much like the functions we have studied in previous lessons that manage array allocation and resizing. In our application, the vertex capacity for the array is initially set to 100 and is increased by 100 every time the limit is reached. This speeds up adding the vertex data to the array and minimizes memory fragmentation. This function adds no data to the ID3DXMesh, as it has not even been created yet. All we are doing is adding vertices to a temporary vertex array which will later be used to create and populate the vertex buffer of the ID3DXMesh. Once the actual ID3DXMesh has been created, this array will no longer be used and may be released.

The first parameter describes the number of vertices we wish to add and the second parameter is a pointer to one or more vertices. We use a void pointer for the vertex data because it must be able to handle arbitrary vertex formats with varying sizes.

```
long CTriMesh::AddVertex( ULONG Count , LPVOID pVertices   )
{
    UCHAR * pVertexBuffer = NULL;

    if ( m_nVertexCount + Count > m_nVertexCapacity )
    {
        // Adjust our vertex capacity (resize 100 at a time)
        for ( ; m_nVertexCapacity < (m_nVertexCount + Count) ; ) m_nVertexCapacity += 100;

        // Allocate new resized array
        if (!(pVertexBuffer = new UCHAR[(m_nVertexCapacity) * m_nVertexStride])) return -1;

        // Existing Data?
        if ( m_pVertex )
        {
            // Copy old data into new buffer
            memcpy( pVertexBuffer, m_pVertex, m_nVertexCount * m_nVertexStride );

            // Release old buffer
            delete []m_pVertex;

        } // End if

        // Store pointer for new buffer
        m_pVertex = pVertexBuffer;

    } // End if a resize is required
```

First we check to see whether the number of vertices passed in will exceed the current capacity of the vertex array. If so, then we need to resize the array. We grow the array by 100 vertices every time the capacity is reached so we will not need to resize again until we have added another 100 vertices.

Note the use of the loop rather than a simple conditional test for the resize. This handles the cases where a user adds large numbers of vertices in one go. If a user wanted to add 500 vertices, we would need to resize the array by 500, not 100. The loop incrementally adds 100 to the current m_nVertexCapcity variable until it is large enough to hold all the vertices required.

Next we use the local temporary pointer pVertexBuffer to allocate a new BYTE array large enough to hold the required number of vertices. Notice that we multiply the vertex capacity by the stride (m_nVertexStride) to get the total number of bytes needed. This is a clear example of why it is important to call SetDataFormat prior to adding vertex data.

If the vertex array currently contains vertex data, then we copy it into the newly allocated buffer and delete the original array because we no longer need it. Finally, we reassign the m_pVertex variable to point to the new vertex array which now contains any previous vertex data and enough room on the end to store the input vertex data. Next we copy over the vertex data passed into this function (it will be appended to the current contents of the buffer). Because we have not yet increased the internal vertex count variable, this tells us exactly where to start adding the new vertices.

```
    // Copy over vertex data if provided
    if ( pVertices )
       memcpy(&m_pVertex[m_nVertexCount*m_nVertexStride],pVertices,Count*m_nVertexStride);
```

Finally, we update the vertex count and return the index of the first newly added vertex in the array. This allows the calling function to retrieve a pointer to the vertex data and use it to start placing vertex data in the correct position.

```
    // Increase Vertex Count
    m_nVertexCount += Count;

    // Return first vertex
    return m_nVertexCount - Count;
}
```

CTriMesh::AddFace

The AddFace function is the second part of the geometry creation functionality. Since the ID3DXMesh object always represents its data as indexed triangle lists, we will follow the same rules when adding mesh indices.

The AddFace function is similar to the AddVertex function in the sense that it is used to temporarily store face data until the ID3DXMesh is created. But there is one important difference -- in addition to the face data (the indices), we must also specify an attribute ID describing the subset the face belongs to. These attribute IDs will be arbitrary values that are used by the application to group faces that have like properties. Therefore, the function must resize the temporary index array as well as the attribute array to allow for one attribute per index buffer triangle.

The first parameter is the number of triangles we wish to add and the second parameter is a void pointer to the index data for the triangles. This buffer should contain Count * 3 indices. The final parameter is the attribute ID we would like associated with the face(s) we are adding. This means that we can add multiple triangles with the same attribute ID in a single call. If we add N faces to the mesh, we will also need to add N attribute ID's to the temporary pAttribute buffer.

```
long CTriMesh::AddFace( ULONG Count, LPVOID pIndices , ULONG AttribID  )
{
    UCHAR * pIndexBuffer = NULL;
    ULONG * pAttributeBuffer = NULL;
```

First we allocate two pointers that can be used to resize the index and attribute arrays. Because the index buffer contains faces and the attribute array holds an attribute ID for each face, these two arrays will always be resized together so that they stay in sync.

Resizing the arrays is identical to the vertex array resize. The default capacity and resize values are both 100.

```
    if ( m_nFaceCount + Count > m_nFaceCapacity )
    {
        // Adjust our face capacity (resize 100 at a time)
        for ( ; m_nFaceCapacity < (m_nFaceCount + Count) ; ) m_nFaceCapacity += 100;
```

We use the local pIndexBuffer pointer to allocate enough memory to hold the required indices. We multiply the desired face count by three to get the desired index count, and then multiply that value by the size of an index (2 or 4 bytes) to get the total number of bytes needed for the new temporary index buffer.

```
        // Allocate new resized array
        if (!( pIndexBuffer = new UCHAR[(m_nFaceCapacity * 3)*m_nIndexStride])) return -1;
```

If there is currently data in the index array then we will copy it into the new index array and release the old array. We reassign m_pIndex to the newly created array.

```
        // Existing Data?
        if ( m_pIndex )
        {
            // Copy old data into new buffer
            memcpy( pIndexBuffer, m_pIndex, (m_nFaceCount * 3) * m_nIndexStride );

            // Release old buffer
            delete []m_pIndex;
        }

        // Reassign pointer to new buffer
        m_pIndex = pIndexBuffer;
```

We do similar resizing to the attribute array.

```
        // Allocate new resized attribute array
        pAttributeBuffer = new ULONG[ m_nFaceCapacity ];
```

If the current attribute array (m_pAttribute) holds any data, then we copy it into the new array. Finally we release the old array and ressign the member variable m_pAttribute.

```
        // Existing Data?
        if ( m_pAttribute )
        {
            // Copy old data into new buffer
            memcpy( pAttributeBuffer, m_pAttribute, m_nFaceCount * sizeof(ULONG) );

            // Release old buffer
            delete []m_pAttribute;
        }

        // Store pointer for new buffer
        m_pAttribute = pAttributeBuffer;

    } // End if a resize is required
```

Next we append the index data passed into the function into the index array.

```
    // Copy over index and attribute data if provided
    if ( pIndices )
      memcpy(&m_pIndex[(m_nFaceCount*3)*m_nIndexStride],pIndices,(Count*3)*m_nIndexStride);
```

Finally, we loop through the new elements added to the attribute array (one for each new face) and set the attribute ID to the value passed into the function. We then increment the mesh face count and return the index of the first newly added face.

```
    for ( ULONG i = m_nFaceCount; i < m_nFaceCount + Count; ++i) m_pAttribute[i] = AttribID;

    // Increase Face Count
    m_nFaceCount += Count;

    // Return first face
    return m_nFaceCount - Count;
}
```

CTriMesh::AddAttributeData

As discussed earlier, a managed mesh maintains an array of MESH_ATTRIB_DATA structures describing the texture and material used by each subset. This allows the mesh to render itself. We use the CTriMesh::AddAttributeData function to add MESH_ATTRIB_DATA information to this array. This function is necessary if we manually create a mesh which is intended for use in managed mode. It allows us to specify the texture and material properties used by each subset to manually build the mesh's internal attribute array. This function serves no purpose for non-managed mode meshes.

The function takes one parameter which describes how many MESH_ATTRIB_DATA structures to make room for in the m_pAttribData array. If the array already contains mesh attribute data then the array will be resized to store the requested number of elements plus the elements already in the array. Array resizing works in exactly the same way as we saw in the last two functions discussed.

```
long CTriMesh::AddAttributeData( ULONG Count  )
{
    MESH_ATTRIB_DATA * pAttribBuffer = NULL;

    // Allocate new resized array
     pAttribBuffer = new MESH_ATTRIB_DATA[ m_nAttribCount + Count ] ;

    // Existing Data?
    if ( m_pAttribData )
    {
        // Copy old data into new buffer
        memcpy( pAttribBuffer, m_pAttribData, m_nAttribCount * sizeof(MESH_ATTRIB_DATA) );

        // Release old buffer
        delete []m_pAttribData;
    }
```

```
    // Store pointer for new buffer
    m_pAttribData = pAttribBuffer;

    // Clear the new items
    ZeroMemory( &m_pAttribData[m_nAttribCount], Count * sizeof(MESH_ATTRIB_DATA) );

    // Increase Attrib Count
    m_nAttribCount += Count;

    // Return first Attrib
    return m_nAttribCount - Count;
}
```

The function returns the index of the first attribute added by the call. The calling function can use the returned value to index into the m_pAttribData function and fill out the required attribute information. Keep in mind that the number of mesh attribute data elements in this array should match the number of subsets in the managed mesh. There is a one-to-one mapping between subsets in the mesh and this array, therefore, the first element you add to this array will be used to describe the texture and material information for the first subset, and so on.

One thing is not immediately clear. The above function allows us to add space in the internal attribute array of the mesh and returns the index of the newly added attribute element. How do we use this to place texture and material information into that array element? The mesh object also exposes a function called GetAttributeData which returns a pointer to the underlying attribute array. You can use this pointer along with the index returned from the above function to access the newly added attribute elements and populate them with texture and material data.

> **Note:** You should not call the AddAttributeData function if you intend to use the mesh in non-managed mode, since this array will only be used in managed mode. In non-managed mode, the application or scene class is responsible for setting a subset's texture and material before rendering it.

Example: Creating a cube using CTriMesh

We have now covered the basic functions we need to add data to the mesh. For illustration, the following code snippet will demonstrate how we might create a simple cube mesh using these functions. This next section is like an instruction manual for how to use the wrapper (as opposed to an examination of the code), and should be helpful in providing insight into the manual creation process.
First the application must allocate a new CTriMesh. In this example we will be using vertices with a 3D position and one set of 2D texture coordinates. After the mesh is created, the SetDataFormat function is called to inform the mesh object of the vertex and index formats intended for use:

```
CTriMesh  CubeMesh;
CubeMesh.SetDataFormat(  D3DFVF_XYZ | D3DFVF_TEX1 , 2 );
```

Because we will be adding one quad at a time in this example (a cube face is two triangles), we will use temporary arrays of 4 vertices and 6 indices to store the information for each face. We will pass this data into the CTriMesh::AddVertex and CTriMesh::AddFace functions respectively:

```
USHORT   Indices[6];
CVertex  Vertices[4];

// Build Front quad ( all quads point inwards in this example)
Vertices[0] = CVertex( -10.0f,  10.0f,  10.0f, 0.0f, 0.0f );
Vertices[1] = CVertex(  10.0f,  10.0f,  10.0f, 1.0f, 0.0f );
Vertices[2] = CVertex(  10.0f, -10.0f,  10.0f, 1.0f, 1.0f );
Vertices[3] = CVertex( -10.0f, -10.0f,  10.0f,, 0.0f, 1.0f );

// Build the front face indices
Indices[0] = 0; Indices[1] = 1; Indices[2] = 3;
Indices[3] = 1; Indices[4] = 2; Indices[5] = 3;
```

We assign the two triangles of this face to attribute 0 using the AddFace function:

```
// Add the vertices and indices to this mesh for front face
CubeMesh.AddVertex( 4, &Vertices );
CubeMesh.AddFace( 2, Indices, 0 );
```

We have now added the indices and the vertices to the mesh. The two triangles we have added belong to subset 0 and describe the front quad of the cube. Now we can re-use our local arrays to store the information for the back quad before adding it to the mesh. These vertices will be added to the mesh as vertices 4 through 7, so we must set the indices with this in mind.

```
// Back Quad
Vertices[0] = CVertex(  10.0f,  10.0f, -10.0f, 0.0f, 0.0f );
Vertices[1] = CVertex( -10.0f,  10.0f, -10.0f, 1.0f, 0.0f );
Vertices[2] = CVertex( -10.0f, -10.0f, -10.0f, 1.0f, 1.0f );
Vertices[3] = CVertex(  10.0f, -10.0f, -10.0f, 0.0f, 1.0f );

// Build the back quad indices
Indices[0] = 4; Indices[1] = 5; Indices[2] = 7;
Indices[3] = 5; Indices[4] = 6; Indices[5] = 7;

// Add the vertices and indices to this mesh
CubeMesh.AddVertex( 4, &Vertices );
CubeMesh.AddFace( 2, Indices, 0 );
```

This face has also been assigned an attribute ID of 0. Thus the two front triangles and the two back triangles of the cube belong to subset 0. As a result, these four triangles will be rendered with the same texture/material combination.

We use the same technique to add the left and right quads. Both will be assigned attribute IDs of 1 so they will belong to the same subset and be rendered with the same texture/material.

```
// Left Quad
Vertices[0] = CVertex( -10.0f,  10.0f, -10.0f, 0.0f, 0.0f );
Vertices[1] = CVertex( -10.0f,  10.0f,  10.0f, 1.0f, 0.0f );
Vertices[2] = CVertex( -10.0f, -10.0f,  10.0f, 1.0f, 1.0f );
Vertices[3] = CVertex( -10.0f, -10.0f, -10.0f, 0.0f, 1.0f );

// Build the left quad indices
Indices[0] = 8; Indices[1] = 9; Indices[2] = 11;
Indices[3] = 9; Indices[4] = 10; Indices[5] = 11;

// Add the vertices and indices to this mesh
CubeMesh.AddVertex( 4, &Vertices );
CubeMesh.AddFace( 2, Indices, 1 );

// Right Quad
Vertices[0] = CVertex(  10.0f,  10.0f,  10.0f, 0.0f, 0.0f );
Vertices[1] = CVertex(  10.0f,  10.0f, -10.0f, 1.0f, 0.0f );
Vertices[2] = CVertex(  10.0f, -10.0f, -10.0f, 1.0f, 1.0f );
Vertices[3] = CVertex(  10.0f, -10.0f,  10.0f,  0.0f, 1.0f );

// Build the right quad indices
Indices[0] = 12; Indices[1] = 13; Indices[2] = 15;
Indices[3] = 13; Indices[4] = 14; Indices[5] = 15;

// Add the vertices and indices to this mesh
CubeMesh.AddVertex( 4, &Vertices );
CubeMesh.AddFace( 2, Indices, 1 );
```

Finally we add the top and bottom quads as subset 2. It should be noted that while we have added the faces to the mesh in subset order, this is not a requirement. You will usually optimize the mesh once it is created so that faces are internally batched by subset anyway.

```
// Top Quad
Vertices[0] = CVertex( -10.0f,  10.0f, -10.0f, 0.0f, 0.0f );
Vertices[1] = CVertex(  10.0f,  10.0f, -10.0f, 1.0f, 0.0f );
Vertices[2] = CVertex(  10.0f,  10.0f,  10.0f, 1.0f, 1.0f );
Vertices[3] = CVertex( -10.0f,  10.0f,  10.0f,  0.0f, 1.0f );

// Build the indices
Indices[0] = 16; Indices[1] = 17; Indices[2] = 19;
Indices[3] = 17; Indices[4] = 18; Indices[5] = 19;

// Add the vertices and indices to this mesh
CubeMesh.AddVertex( 4, &Vertices );
CubeMesh.AddFace( 2, Indices, 2 );

// Bottom Quad
Vertices[0] = CVertex( -10.0f, -10.0f,  10.0f, 0.0f, 0.0f );
Vertices[1] = CVertex(  10.0f, -10.0f,  10.0f, 1.0f, 0.0f );
Vertices[2] = CVertex(  10.0f, -10.0f, -10.0f, 1.0f, 1.0f );
Vertices[3] = CVertex( -10.0f, -10.0f, -10.0f, 0.0f, 1.0f );
```

```
// Build the indices
Indices[0] = 20; Indices[1] = 21; Indices[2] = 23;
Indices[3] = 21; Indices[4] = 22; Indices[5] = 23;

// Add the vertices and indices to this mesh
CubeMesh.AddVertex( 4, &Vertices );
CubeMesh.AddFace( 2, Indices, 2 );
```

At this point the mesh has all vertices stored in its temporary vertex array, all indices in its temporary index array, and the attribute IDs for each face are stored in its temporary attribute array (describing three subsets).

If we were intending to use this mesh as a non-managed mesh then our work would be done (except for loading textures and making sure that they are set before rendering the correct subsets). However, in this example we will assume that this is a managed mesh with attribute IDs in the range of 0 to 2 (i.e., three subsets). Therefore, we need to add the attribute data to the mesh for each subset so that the mesh knows which textures and materials to set when rendering its subsets.

First we call the AddAttributeData function with a value of 3 so that the mesh allocates its MESH_ATTRIB_DATA array to hold 3 elements.

```
// Add the attribute data (We'll let the mesh manage itself)
CubeMesh.AddAttributeData( 3 );
```

Next we call CTriMesh::GetAttributeData which will return a pointer to the internal MESH_ATTRIB_DATA array so we can populate it with meaningful attribute data for each subset.

```
MESH_ATTRIB_DATA *pAttribData;
pAttribData = CubeMesh.GetAttributeData();
```

Now we can loop through each element and store the texture pointer and the material for the subset mapped to that attribute. In this example we are not storing normals in our vertices so will not be using the DirectX lighting pipeline. Therefore, we do not bother setting the material for each subset because it will not be used. We just create a default material and copy it into each attribute.

We also assume that pTexture is an application-owned array containing the three textures used by the subsets. These textures would have been created prior to this code being executed. What is worthy of note is that the texture and attribute callback functions are only used by the mesh when parsing X files. When the application manually creates meshes in this way, it is responsible for creating the required textures and storing their pointers in the mesh attribute array (in managed mode only).

```
D3DMATERIAL9 Material;
ZeroMemory( &Material, sizeof(D3DMATERIAL9));

// Set the attribute data for each subset
for ( ULONG i = 0; i < 3; ++i )
{
    pAttribData[i].Texture  = pTexture[I];
```

```
        pTexture[I]->AddRef();
        pAttribData[i].Material = Material;
    }
```

A good example of mesh creation happens when loading an IWF file. The IWF file contains all the texture filenames which would be initially created and stored by the scene. The IWF file also contains the face data and the material and texture mappings for each face. Using this information, the faces can be added to the mesh one triangle at a time and the attributes can be grouped and added as the subsets of the mesh.

All that is left to do at this point is instruct the CTriMesh object to build its internal ID3DXMesh object using the data we have added to the temporary arrays. Notice that when we call BuildMesh, we pass in the D3DXMESH_MANAGED member of the D3DXMESH enumerated type in this example. This indicates that we would like the mesh vertex and index buffers created in the managed resource pool so that they can support automatic recovery from a lost/reset device.

```
// Build the mesh
CubeMesh.BuildMesh( D3DXMESH_MANAGED, m_pD3DDevice );
```

CTriMesh::BuildMesh

The BuildMesh function should be called after the application has finished manually populating the mesh with data. It should not be called if we have loaded the data from an X file as the underlying mesh will have already been created. BuildMesh will generate the underlying ID3DXMesh object and fill its vertex and vertex buffers with the information currently stored in the temporary arrays. The code is shown next a few lines at a time. This listing has had some of the error checking removed to improve readability, but the error checking is performed in the actual source code.

The function takes three parameters. The first specifes zero or more members of the D3DXMESH enumerated type. This indicates which memory pool we wish the ID3DXMesh we are about to create to use for its vertex and index buffers. The second parameter is a pointer to the Direct3D device object which will own the mesh resources. The optional third parameter, which is set to TRUE by default, is a Boolean variable indicating whether we wish the temporary vertex, index and attribute arrays to be freed from memory after the ID3DXMesh has been successfully created.

Usually you would want to release these temporary storage bins as they are no longer required, but it can be useful to keep them around if the underlying mesh is to be created in the default pool. If the device should become lost, default pool meshes would need to be created again from scratch, requiring your application to manually add all the vertex and index data again. If you do not release the temporary storage bins, then a simple call to the CTriMesh::BuildMesh function is all that is needed to restore the mesh to its former status. The underlying ID3DXMesh will still be released and recreated, but the vertex and index data is still in the temporary bins and is automatically copied over by this function. This convenience obviously comes at the expense of increased memory requirement.

```
HRESULT CTriMesh::BuildMesh(ULONG Options, LPDIRECT3DDEVICE9 pDevice, bool ReleaseOriginals)
{
    HRESULT hRet;
    LPVOID  pVertices   = NULL;
    LPVOID  pIndices    = NULL;
    ULONG  *pAttributes = NULL;
```

If this CTriMesh object already has an ID3DXMesh assigned to it, we should release it. This allows us to call this function to rebuild meshes that have become invalid due to device loss / reset.

```
    // First release the original mesh if one exists
    if ( m_pMesh ) { m_pMesh->Release(); m_pMesh = NULL; }
```

Next we test the m_nIndexStride member variable to see if the user has set the index data format to 32-bit. By default D3DXCreateMeshFVF creates meshes with 16-bit indices, so we will need to modify the mesh creation options to include the D3DXMESH_32BIT flag in that case.

```
    // Force 32 bit mesh if required
    if ( m_nIndexStride == 4 ) Options |= D3DXMESH_32BIT;
```

Now we call D3DXCreateMeshFVF to create the ID3DXMesh. We pass in the number of faces and the number of vertices the mesh will require, followed by the creation options. We also pass the FVF flags that were registered with the mesh and a pointer to the device that will own the mesh. The final parameter is the CTriMesh::m_pMesh pointer that will point to a valid ID3DXMesh interface if the function is successful.

```
     // Create the blank empty mesh
    D3DXCreateMeshFVF(m_nFaceCount,m_nVertexCount,Options,m_nVertexFVF,pDevice,&m_pMesh);
```

At this point our D3DXMesh has been created, but it contains no data. The next step is to lock the mesh vertex, index, and attribute buffers and copy over all of the information stored in our temporary arrays.

```
    // Lock the vertex buffer and copy the data
    m_pMesh->LockVertexBuffer( 0, &pVertices );
    memcpy( pVertices, m_pVertex, m_nVertexCount * m_nVertexStride );
    m_pMesh->UnlockVertexBuffer();

    // Lock the index buffer and copy the data
    m_pMesh->LockIndexBuffer( 0, &pIndices );
    memcpy( pIndices, m_pIndex, (m_nFaceCount * 3) * m_nIndexStride );
    m_pMesh->UnlockIndexBuffer();

    // Lock the attribute buffer and copy the data
    m_pMesh->LockAttributeBuffer( 0, &pAttributes );
    memcpy( pAttributes, m_pAttribute, m_nFaceCount * sizeof(ULONG) );
    m_pMesh->UnlockAttributeBuffer();
```

Finally, if the ReleaseOriginals Boolean parameter was set to TRUE (the default setting) then the temporary storage bins are de-allocated and their pointers are set to NULL. We also set all other variables that describe the data in the temporary arrays to zero.

```
    // Release the original data if requested
    if ( ReleaseOriginals )
    {
        if ( m_pVertex    ) delete []m_pVertex;
        if ( m_pIndex     ) delete []m_pIndex;
        if ( m_pAttribute ) delete []m_pAttribute;

        m_nVertexCount    = 0;
        m_nFaceCount      = 0;
        m_pAttribute      = NULL;
        m_pIndex          = NULL;
        m_pVertex         = NULL;
        m_nVertexCapacity = 0;
        m_nFaceCapacity   = 0;
     }

    // We're done
    return S_OK;
}
```

At this point the mesh has been fully created and populated and can be rendered using the DrawSubset function or the Draw function (if this is a managed mesh).

CTriMesh::Draw

CTriMesh::Draw automates the rendering of a managed mesh and should not be used to render non-managed meshes. A managed mode mesh maintains internal texture and material resources for subset rendering. As this is not the case for non-managed meshes, this function only works in managed mode and will immediately return if called for a non-managed mesh.

```
void CTriMesh::Draw(  )
{
    LPDIRECT3DDEVICE9 pD3DDevice = NULL;

    // If the mesh has not been created yet bail
    if (!m_pMesh ) return;

    // This function is invalid if there is no managed data
    if ( !m_pAttribData ) return;

    // Retrieve the Direct3D device
    m_pMesh->GetDevice( &pD3DDevice );

    // Set the attribute data
    pD3DDevice->SetFVF( GetFVF() );

    // Render the subsets
    for ( ULONG i = 0; i < m_nAttribCount; ++i )
    {
        pD3DDevice->SetMaterial( &m_pAttribData[i].Material );
        pD3DDevice->SetTexture( 0, m_pAttribData[i].Texture );
```

```
        m_pMesh->DrawSubset( i );

    } // Next attribute

    // Release the device
    pD3DDevice->Release();
}
```

After checking for managed mode rendering, the function retrieves the device from the ID3DXMesh. We then loop through each subset of the mesh, set its texture and material, and call ID3DXMesh::DrawSubset with the subset attribute ID. Finally, we release the device interface because ID3DXMesh::GetDevice increments the device reference count before returning the pointer. CTriMesh::GetFVF is a simple wrapper that passes the request through to ID3DXMesh::GetFVF to get the FVF flags the mesh was created with.

CTriMesh::DrawSubset

DrawSubset provides an interface for rendering individual mesh subsets. It can be used in both managed and non-managed modes. It is in fact our only means for rendering a non-managed mesh. A higher level process will be required to set the texture, material, and other states for each subset prior to the render call (as seen in the CTriMesh::Draw function shown above). This allows the scene to batch render subsets from multiple meshes to achieve attribute order rendering across mesh boundaries. This function can also be called to render a subset of a managed mesh, in which case the texture and material of the subset will be set automatically.

```
void CTriMesh::DrawSubset( ULONG AttributeID )
{
    LPDIRECT3DDEVICE9 pD3DDevice = NULL;

    // Set the attribute data if managed mode mesh
    if ( m_pAttribData && AttributeID < m_nAttribCount )
    {
        // Retrieve the Direct3D device
        m_pMesh->GetDevice( &pD3DDevice );

        pD3DDevice->SetMaterial( &m_pAttribData[AttributeID].Material );
        pD3DDevice->SetTexture( 0, m_pAttribData[AttributeID].Texture );

        // Release the device
        pD3DDevice->Release();
    }

    //draw the subset
    m_pMesh->DrawSubset( AttributeID );
}
```

If the mesh attribute data array is defined, then this is a managed mesh and we set the subset texture and material before rendering. If it is non-managed, then we simply call ID3DXMesh::DrawSubset straight away.

CTriMesh::OptimizeInPlace

CTriMesh::OptimizeInPlace is basically a wrapper around the ID3DXMesh::OptimizeInPlace function.

The first parameter is a DWORD containing one or more D3DXMESHOPT flags describing the optimization we wish to perform. The second and third parameters can be set to NULL or can be passed the address of ID3DXBuffer interface pointers that will contain the face and vertex remap information respectively. These should not be allocated buffers since the function will create both buffers for you.

```
HRESULT CTriMesh::OptimizeInPlace( DWORD Flags, LPD3DXBUFFER *ppFaceRemap,
                                   LPD3DXBUFFER *ppVertexRemap)
{
    HRESULT      hRet;
    LPD3DXBUFFER pFaceRemapBuffer = NULL;
    ULONG       *pData = NULL;
```

If the mesh has not yet had its adjacency information generated then we do so at this point. The m_pAdjacency member variable is a pointer to an ID3DXBuffer interface. The GenerateAdjacency function will create the D3DXBuffer object and fill it with adjacency information.

```
    // Generate adjacency if none yet provided
    if (!m_pAdjacency)
    {
        GenerateAdjacency();
    }
```

ID3DXMesh::OptimizeInPlace can be somewhat confusing when it comes to generating remap information. To get the face remap information you pass in a pointer to a pre-allocated DWORD array. To get vertex remap information you just pass a pointer to an ID3DXBuffer and the function will generate and fill it with the relevant data. To avoid confusion, our function will return both face and vertex remapping information in ID3DXBuffer objects. Therefore, we must first create the face remap buffer ourselves, obtain a pointer to its data area, cast it to a DWORD, and pass it into ID3DXMesh::OptimimzeInPlace.

```
    // Allocate the output face remap if requested
    if ( ppFaceRemap )
    {
        D3DXCreateBuffer( GetNumFaces() * sizeof(ULONG), ppFaceRemap );
         pData = (ULONG*)(*ppFaceRemap)->GetBufferPointer();
    }
```

Finally, we call ID3DXMesh::OptimizeInPlace to optimize the mesh data.

```
    // Optimize the data
    m_pMesh->OptimizeInplace(Flags, (DWORD*)m_pAdjacency->GetBufferPointer(),
                             (DWORD*)m_pAdjacency->GetBufferPointer(), pData, ppVertexRemap );
    return S_OK;
}
```

CTriMesh::Optimize

This function is a bit more complex than its predecessor due to the fact that it has to clone the optimized data into a new CTriMesh object. However, unlike ID3DXMesh::Optimize which automatically creates the output mesh, CTriMesh::Optimize expects the application to pass in a pre-created CTriMesh object. This mesh will be the recipient of the optimized data. This affords us the extra flexibility of being able to use a statically allocated or stack allocated CTriMesh object as the output mesh. This might be useful if you wanted to create a temporary optimized mesh locally inside a function, where the output mesh would be allocated on the stack and automatically discarded when the function returns.

The first parameter is a combination of D3DXMESH flags describing how the underlying ID3DXMesh object in the output mesh should be created. This can be combined with one or more D3DXMESHOPT flags describing the optimization to perform. The second parameter is a pointer to a CTriMesh object that will receive the optimized ID3DXMesh object. This CTriMesh pointer should point to an already created CTriMesh object. If the output mesh already contains an underlying ID3DXMesh object, this mesh will be released in favor of the new one that is created by this function.

```
HRESULT CTriMesh::Optimize(ULONG Flags, CTriMesh *pMeshOut, LPD3DXBUFFER *ppFaceRemap,
                           LPD3DXBUFFER *ppVertexRemap , LPDIRECT3DDEVICE9 pD3DDevice  )
{
    HRESULT             hRet;
    LPD3DXMESH          pOptimizeMesh = NULL;
    LPD3DXBUFFER        pFaceRemapBuffer = NULL;
    LPD3DXBUFFER        pAdjacency = NULL;
    ULONG              *pData      = NULL;
```

If the mesh we are cloning does not yet have its adjacency information generated, then we need to generate it, because it is needed in the call to ID3DXMesh::Optimize.

```
    // Generate adjacency if not yet available
    if (!m_pAdjacency)
    {
        hRet = GenerateAdjacency();
    }
```

If the caller did not pass in a pointer to a device, then we will use the device of the current mesh being cloned. Typically you will not want to specify a different device unless you are using multiple devices (very rare). Keep in mind that while the CTriMesh object is not bound to any particular device, its underlying ID3DXMesh object is. The next section of code calls CTriMesh::GetDevice to get a pointer to the current mesh's device if a device pointer was not specified. If a device pointer was passed, then we increment its reference count until we finish using it.

```
    if ( !pD3DDevice )
    {
        // we'll use the same device as this mesh
        // This automatically calls 'AddRef' for the device
        m_pMesh->GetDevice( &pD3DDevice );
```

```
    }
    else
    {
        // Otherwise we'll add a reference here so that we can
        // release later for both cases without doing damage :)
        pD3DDevice->AddRef();
    }
```

ID3DXMesh::Optimize also returns face adjacency information for the optimized mesh, so we create a buffer to store this information. As with the OptimizeInPlace method, we create the face remap buffer if a face remap pointer was passed into the function.

```
    // Allocate new adjacency output buffer
    D3DXCreateBuffer( (3 * GetNumFaces()) * sizeof(ULONG), &pAdjacency );

    // Allocate the output face remap if requested
    if ( ppFaceRemap )
    {
        // Allocate new face remap output buffer
        D3DXCreateBuffer( GetNumFaces() * sizeof(ULONG), ppFaceRemap );
        pData = (ULONG*)(*ppFaceRemap)->GetBufferPointer();
   }
```

At the top of this function we declared a local ID3DXMesh pointer called pOptimizeMesh. This pointer will be fed into the ID3DXMesh::Optimize function and will point to the newly created optimized mesh on function return.

```
   // Attempt to optimize the mesh
   m_pMesh->Optimize( Flags,
                      (ULONG*)m_pAdjacency->GetBufferPointer(),
                      (ULONG*)pAdjacency->GetBufferPointer(),
                       pData, ppVertexRemap, &pOptimizeMesh );
```

At this point we now have an optimized ID3DXMesh, but it is not yet connected to the output CTriMesh object that was passed into this function. So we call the CTriMesh::Attach method for the output CTriMesh object which points the m_pMesh pointer to the optimized ID3DXMesh. Notice that we also pass the face adjacency buffer so that the new mesh has this data resident.

```
    // Attach this D3DX mesh to the output CTriMesh
    // This automatically adds a reference to the mesh passed in.
    pMeshOut->Attach( pOptimizeMesh, pAdjacency );
```

The optimized mesh and its adjacency buffer have now been assigned to the output CTriMesh, which increases the reference count on both. We no longer need to use these interfaces in this function so we release them. This does not destroy the buffer or the mesh because they are now being referenced by the output CTriMesh object.

```
    // We can now release our copy of the optimized mesh and the adjacency buffer
    pOptimizeMesh->Release();
    pAdjacency->Release();
```

If we are optimizing a managed mesh, then the source mesh will have an array of mesh attribute data elements describing the textures and materials used by each subset. If this is the case, we must also copy this data into the mesh attribute array of the output CTriMesh. First we call AddAttributeData to make room for the new attributes and then we copy the data over.

```
    // Copy over attributes if there is anything here
    if ( m_pAttribData )
    {
        // Add the correct number of attributes
         pMeshOut->AddAttributeData( m_nAttribCount ) ;

        // Copy over attribute data
        MESH_ATTRIB_DATA * pAttributes = pMeshOut->GetAttributeData();
        for ( ULONG i = 0; i < m_nAttribCount; ++i )
        {
            MESH_ATTRIB_DATA * pAttrib = &pAttributes[i];

            // Store details
            pAttrib->Material = m_pAttribData[i].Material;
            pAttrib->Texture  = m_pAttribData[i].Texture;
            pAttrib->Effect   = m_pAttribData[i].Effect;

            // Add references so that objects aren't released when either of these
            // meshes are released, or vice versa.
            if ( pAttrib->Texture ) pAttrib->Texture->AddRef();
            if ( pAttrib->Effect  ) pAttrib->Effect->AddRef();
        } // Next Attribute
    } // End if managed

    // Release our referenced D3D Device
    if (pD3DDevice) pD3DDevice->Release();

    // Success!!
    return S_OK;
}
```

Notice that when we copy over each texture pointer (and effect pointer) we make sure that we increase the reference count so that it correctly reflects how many external pointers to the interface are in existence.

CTriMesh::GenerateAdjacency

The CTriMesh class includes a member variable called m_pAdjacency which is a pointer to an ID3DXBuffer interface. This function creates this buffer and calls ID3DXMesh::GenerateAdjacency to calculate the adjacency information. ID3DXMesh::GenerateAdjacency expects a DWORD pointer to an array large enough to hold the adjacency information, so we simply allocate the ID3DXBuffer large enough to hold three DWORDS per face and retrieve the buffer data pointer and cast it before passing it into the function. CTriMesh::GenerateAdjacency accepts an optional epsilon parameter which will default to 0.001. This is used as a tolerance value when comparing vertices in neighboring faces to see if they are joined.

Note: This function should only be called after the underlying ID3DXMesh has been created.

```
HRESULT CTriMesh::GenerateAdjacency( float Epsilon )
{
    HRESULT hRet;

    // Validate Requirements
    if ( !m_pMesh ) return D3DERR_INVALIDCALL;

    // Clear out any old adjacency information
    if (m_pAdjacency) m_pAdjacency->Release();

    // Create the new adjacency buffer
    hRet = D3DXCreateBuffer( GetNumFaces() * (3 * sizeof(DWORD)), &m_pAdjacency );
    if ( FAILED(hRet) ) return hRet;

    // Generate the new adjacency information
    hRet = m_pMesh->GenerateAdjacency( Epsilon, (DWORD*)m_pAdjacency->GetBufferPointer() );
    if ( FAILED(hRet) ) return hRet;

    // Success !!
    return S_OK;
}
```

CTriMesh::Attach

The Attach function is used when a new ID3DXMesh is created and needs to be attached to a pre-existing CTriMesh object. Its parameters are an ID3DXMesh and its adjacency buffer. The adjacency buffer passed is simply stored for later use. If you do not pass the adjacency buffer it will be generated when it is needed. The function returns NULL if a valid mesh pointer is not passed.

```
HRESULT CTriMesh::Attach( LPD3DXBASEMESH pMesh, LPD3DXBUFFER pAdjacency /* = NULL */ )
{
    HRESULT hRet;

    // Validate Requirements
    if ( !pMesh ) return D3DERR_INVALIDCALL;

    // Clear our current data
    Release();
```

We start by calling CTriMesh::Release to release any prior mesh or adjacency data. Since the input mesh is the generic base class, we use a method of the IUnknown interface to determine whether or not the passed pointer is to the correct COM interface (ID3DXMesh in this case). We call the IUnkown::QueryInterface method and pass the interface type we are querying and the m_pMesh pointer. If the mesh is a valid ID3DXMesh COM interface, it will be copied into the m_pMesh pointer and its reference count incremented automatically by the QueryInterface call.

```
    // Store this mesh (ensuring that it really is of the expected type)
```

```
    // This will automatically add a reference of the type required
    hRet = pMesh->QueryInterface( IID_ID3DXMesh, (void**)&m_pMesh );
    if ( FAILED(hRet) ) return hRet;
```

Next, we copy FVF, vertex and index stride, and the adjacency buffer (if it was passed in). We remember to call AddRef to ensure proper reference counting on the buffer. Once done, we return success.

```
    // Calculate strides etc
    m_nVertexFVF    = m_pMesh->GetFVF();
    m_nVertexStride = m_pMesh->GetNumBytesPerVertex();
    m_nIndexStride  = (GetOptions() & D3DXMESH_32BIT) ? 4 : 2;

    // If adjacency information was passed, reference it
    // if none was passed, it will be generated later, if required.
    if ( pAdjacency )
    {
        m_pAdjacency = pAdjacency;
        m_pAdjacency->AddRef();
    }
    // Success!!
    return S_OK;
}
```

CTriMesh::CloneFVF

A useful feature of the ID3DXMesh interface is the ability to clone the mesh into a new mesh with a different vertex/index format. This is especially handy when we load data from an X file which may lack certain pieces of information we need or more components than we intend to support.

This function is almost identical to our Optmize function, without the actual optimization call. The function simply clones and attaches the mesh based on the creation options and FVF specified as input parameters. Adjacency data is either copied or generated and attributes are copied if the mesh is in managed mode. The final mesh is passed out using the third parameter.

```
HRESULT CTriMesh::CloneMeshFVF( ULONG Options, ULONG FVF, CTriMesh * pMeshOut,
                                LPDIRECT3DDEVICE9 pD3DDevice /* = NULL */ )
{
    HRESULT             hRet;
    LPD3DXMESH          pCloneMesh  = NULL;

    // Validate requirements
    if ( !m_pMesh || !pMeshOut ) return D3DERR_INVALIDCALL;

    // Generate adjacency if not yet available
    if (!m_pAdjacency)
    {
       GenerateAdjacency();
    }

    // If no new device was passed...
    if ( !pD3DDevice )
```

```
    {
        // we'll use the same device as this mesh
        // This automatically calls 'AddRef' for the device
        m_pMesh->GetDevice( &pD3DDevice );
    }
    else
    {
        // Otherwise we'll add a reference here so that we can
        // release later for both cases without doing damage :)
        pD3DDevice->AddRef();
    }

    // Attempt to clone the mesh
    m_pMesh->CloneMeshFVF( Options, FVF, pD3DDevice, &pCloneMesh );

    // Attach this D3DX mesh to the output mesh
    // This automatically adds a reference to the mesh passed in.
    pMeshOut->Attach( pCloneMesh );

    // We can now release our copy of the cloned mesh
    pCloneMesh->Release();

    // Copy over attributes if there is anything here
    if ( m_pAttribData )
    {
        // Add the correct number of attributes
        if ( pMeshOut->AddAttributeData( m_nAttribCount ) < 0 ) return E_OUTOFMEMORY;

        // Copy over attribute data
        MESH_ATTRIB_DATA * pAttributes = pMeshOut->GetAttributeData();
        for ( ULONG i = 0; i < m_nAttribCount; ++i )
        {
            MESH_ATTRIB_DATA * pAttrib = &pAttributes[i];
            // Store details
            pAttrib->Material = m_pAttribData[i].Material;
            pAttrib->Texture  = m_pAttribData[i].Texture;
            pAttrib->Effect   = m_pAttribData[i].Effect;

            // Add references so that objects aren't released when either of these
            // meshes are released, or vice versa.
            if ( pAttrib->Texture ) pAttrib->Texture->AddRef();
            if ( pAttrib->Effect  ) pAttrib->Effect->AddRef();

        } // Next Attribute

    } // End if managed
    // Release our referenced D3D Device
    if (pD3DDevice) pD3DDevice->Release();

    // Success!!
    return S_OK;
}
```

CTriMesh::WeldVertices

WeldVertices is a thin wrapper around the D3DXWeldVertices function.

```
HRESULT CTriMesh::WeldVertices( ULONG Flags,  const D3DXWELDEPSILONS * pEpsilon  )
{
    HRESULT          hRet;
    D3DXWELDEPSILONS WeldEpsilons;

    // Validate Requirements
    if ( !m_pMesh ) return D3DERR_INVALIDCALL;

    // Generate adjacency if none yet provided
    if (!m_pAdjacency)
    {
        GenerateAdjacency();
    } // End if no adjacency

    // Fill out an epsilon structure if none provided
    if ( !pEpsilon )
    {
        // Set all epsilons to 0.001;
        float * pFloats = (float*)&WeldEpsilons;
        for ( ULONG i = 0; i < sizeof(D3DXWELDEPSILONS) / sizeof(float); i++ )
            *pFloats++ = 1e-3f;

        // Store a pointer (this doesn't get passed back or anything,
        // we're just reusing the empty var)
        pEpsilon = &WeldEpsilons;

    } // End if

    // Weld the vertices
    D3DXWeldVertices( m_pMesh, Flags, pEpsilon, (DWORD*)m_pAdjacency->GetBufferPointer(),
                      (DWORD*)m_pAdjacency->GetBufferPointer(), NULL, NULL );
     // Success!!
    return S_OK;
}
```

This function takes two parameters. The first is a flag containing zero or more members of the D3DXMESH enumerated type. While these are usually used as mesh creation flags and this function does not create a new output mesh, the weld operation will regenerate the vertex and index buffers. This means that we can optionally use the weld function to change the resource pool allocation strategy while we are welding it.

The second parameter is a pointer to a D3DXWELDEPSILONS structure which allows us to specify floating point tolerances for each possible vertex component. Specifying NULL for the second parameter will force the function to use a default comparison tolerance of 0.001 for each vertex component.

The D3DXWeldVertices function needs to be passed the face adjacency information, so if our CTriMesh object has not yet generated it, we tell it to do so. Next we test to see if a valid pointer to a

D3DXWELDEPSILONS structure was passed. If not, then we temporarily create one and set all of its tolerance values to 0.001. Finally, we call D3DXWeldVertices to perform the weld on the ID3DXMesh for this CTriMesh object.

CTriMesh::GetNumFaces

There are a number of CTriMesh functions we will not discuss here. They are all GetXX style functions which are generally one or two lines long. However we will use this one method as an example of the rest.

```
ULONG CTriMesh::GetNumFaces( ) const
{
    // Validation!!
    if ( !m_pMesh )
        return m_nFaceCount;           // Number of faces in temp index buffer
    else
        return m_pMesh->GetNumFaces(); // Number of faces in actual ID3DXMesh
}
```

This is a good example of how all of the GetXX functions work. Since the user may be creating meshes manually, they might call this function before calling the Build function to create the ID3DXMesh object. In this case, the function should return the number of faces that have currently been added to the temporary index arrays (m_nFaceCount). If the ID3DXMesh has been created however, then we call the ID3DXMesh::GetNumFaces function to return the number of faces in the actual mesh object. Many of these GetXX functions are built to work in these two modes.

Introducing IWF External References

In Chapter Five we discussed loading IWF files using the IWF SDK. While most of that code will remain intact, in this project we will be introducing some new IWF chunks that are specific to GILES™.

GILES™ v1.5 and above includes a new data type called a *reference*. References allow the artist to place multiple objects in the scene which all share the same physical mesh data. This is very much like the concept of instances we discussed in Chapter One. We only need to store one copy of the vertices and indices in memory and then have multiple objects reference the data. Any changes to the data will affect all objects that are referencing it. References can point to internal objects such as those created in GILES™ or they can point to external files. In the latter case, the IWF will contain filenames for X files. The X file data is not included within the IWF file, only the name is stored. Thus when we find an external reference, we will load the file ourselves.

External reference objects are very useful because they allow us to use GILES™ to place and position objects in the scene while still keeping the mesh data separate from the scene database. This means that if we need to tweak the polygons in one of our X files, we can do so without having to resave the scene. This also means that we can use GILES™ to create the scene and position placeholder objects in the world even before the artist has finalized the assets. Once the model is complete, it can be dropped into the game folder and (as long as the name has not changed) our code will display the scene properly.

You will recall from earlier lessons how the IWF SDK provides us with a helper function to automate the loading of IWF files. After the file has been loaded, the various components of the scene are stored in lists. As a simple example, all faces are stored in an IWFSurface list, all entities are stored in IWFEntity lists, etc. GILES™ reference objects (internal and external) are implemented as an entity plugin. This means that they will be stored in the entity list created by the IWF SDK loader, not the mesh list, and will require our code to load the mesh data using the given filename or object name contained inside the entity.

Below you can see how the reference entity is laid out in memory inside the entity's data area. This is the structure we will use in our CScene class to access the data stored in a reference entity loaded by the IWF SDK.

```
typedef struct _ReferenceEntity
{
    ULONG       ReferenceType;          // What type of reference is this
    ULONG       Flags;                  // Reserved flags
    ULONG       Reserved1;              // Reserved values
    ULONG       Reserved2;              // Reserved values
    ULONG       Reserved3;              // Reserved values
    char        ReferenceName[1024];    // External file name or internal object name
};
```

Many of these members are reserved for future use and can be ignored. Our focus is on the ReferenceType (0 for internal references, 1 for external references) and the ReferenceName (filename for external refernces, object name for internal references). For internal references, the name will be the

name of another object stored inside the IWF file. In GILES™, we can give objects in the scene unique names so that they can be referenced in this way. To keep things simple, our first application will support only external references (references to X files). This allows us to demonstrate how to use CTriMesh::LoadMeshFromX to load mesh data for each object in the scene stored in an IWF file.

> **Note:** The IWF file distributed with this project (Space_Scene.iwf) is unlike others we have loaded in past lessons. This file contains no actual geometry data. It contains only a skybox and a list of external reference entities. The scene uses four meshes, each stored in its own X file. Three of the meshes are spacecraft models (Cruiser1.x, Cruiser2.x, sfb-04.x). These objects were positioned in the scene using external referencing within GILES, so that only the object world matrix is stored in the IWF file along with its filename. The fourth X file is called Asteroid.x and is referenced many times in the scene. When we load these objects, we will give each reference its own CObject structure and attach the appropriate CTriMesh to each. From that point forward, rendering proceeds as usual.

The CScene Class

Our scene class will have several responsibilities:

- File loading
- Resource management
    - Textures
    - Materials
    - (Attributes)
    - Meshes
- Rendering

For mesh data, the scene class must be prepared to:

- Load externally referenced X files contained in IWF files
- Manually fill meshes with IWF data
- Manually fill meshes with procedural data (i.e. skyboxes, cubes, etc.)

CGameApp::BuildObjects starts asset loading with a call to CScene::LoadSceneFromIWF. This is very similar to previous projects we have developed. As before, we will use the CFileIWF class that ships with the IWF SDK to handle file import automatically on our behalf. This object contains a number of STL vectors for storage of entities, meshes, texture names, materials, etc that were extracted from the file. We will implement a number of processing functions (ProcessMeshes, ProcessEntities, etc.) to extract the relevant data from these vectors and convert them into a format which our scene object accepts.

> **Note:** We will not be covering all of the code in the CScene class since much of it is unchanged from previous applications. There are some small alterations where we might now copy data into a CTriMesh object instead of into a CMesh structure, but that is about the extent of it. We will focus on the major areas that have undergone revision such as data processing and rendering.

An abbreviated version of the CScene class declaration follows. The full declaration can be found in the header file CScene.h.

```
class CScene
{
public:
    //-----------------------------------------------------------------------
    // Constructors & Destructors for This Class.
    //-----------------------------------------------------------------------
     CScene( );
    ~CScene( );

    //-----------------------------------------------------------------------
    // Public Functions for This Class
    //-----------------------------------------------------------------------
    bool                LoadSceneFromIWF( TCHAR * strFileName,
                                          ULONG LightLimit = 0,
                                          ULONG LightReservedCount = 0 );

    bool                LoadSceneFromX  ( TCHAR * strFileName );

    void                Render          ( CCamera & Camera );
    void                RenderSkyBox    ( CCamera & Camera );
```

LoadSceneFromIWF will be called from CGameApp at application startup. This function was available in previous lessons when we discussed light groups (not used in this demo). The LightLimit parameter in the LoadSceneFromIWF file will simply be used to indicate how many lights we wish to load from the IWF file. For example, if the current hardware only supports eight simultaneous lights, we would set this to 8 so that only the first eight lights in the IWF file would be used and the rest would be ignored. In this application, the third parameter is not used but you may remember that it was used in the light group demo to reserve a number of light slots for dynamic lights.

The CScene class also has a LoadSceneFromX function which provides loading of X files which may contain individual or multiple meshes. Unlike LoadSceneFromIWF which gives each mesh loaded its own world matrix, LoadSceneFromX creates a single CTriMesh for the entire scene. If the X file contains multiple meshes, they will be collapsed into a single mesh. We will learn in the next chapter how to load hierarchical X files containing multiple meshes. In that case each mesh will maintain its own world matrix and can be manipulated separately from other meshes stored in the same X file.

CScene also has two render functions: CScene::Render is called by CGameApp::FrameAdvance to draw all scene meshes. CScene::RenderSkyBox draws the skybox and is called prior to mesh rendering to present a nice backdrop for the scene.

The next two methods are static callback functions that the scene will register with the CTriMesh class to handle the loading of textures and materials. We need two callback functions to differentiate the behaviour of managed vs. non-managed mesh modes. A managed mesh will call the CollectTexture function to load textures and store their pointers in its attribute data array. Non-managed meshes call CollectAttributeID to search the scene database for a material/texture combo index which is used to re-map the mesh attribute buffer to reference the global resource list.

```
    //-----------------------------------------------------------------------
    // Static Public Functions for This Class
    //-----------------------------------------------------------------------
    static LPDIRECT3DTEXTURE9   CollectTexture ( LPVOID pContext, LPCTSTR FileName );
    static ULONG                CollectAttributeID (
                                        LPVOID pContext, LPCTSTR strTextureFile,
                                        const D3DMATERIAL9 * pMaterial,
                                        constD3DXEFFECTINSTANCE *pEffectInstance=NULL);
```

In our demo project, we will use a managed mesh for our skybox because it shares no resources with other objects in the scene and non-managed meshes for all scene geometry. This should give you a good sense of how these types can be used in the same application.

After LoadSceneFromIWF calls CFileIWF::Load, the IWF file data is stored in the CFileIWF object's internal vectors. As in previous demos, the scene will then extract the required data from those vectors using a series of ProcessXX functions.

```
    private:
    //-----------------------------------------------------------------------
    // Private Functions for This Class
    //-----------------------------------------------------------------------
    bool                ProcessMeshes     ( CFileIWF & pFile );
    bool                ProcessVertices   ( CTriMesh * pMesh, iwfSurface * pFilePoly );
    bool                ProcessIndices    ( CTriMesh * pMesh, iwfSurface * pFilePoly,
                                            ULONG AttribID = 0, bool BackFace = false);
    bool                ProcessMaterials  ( const CFileIWF& File );
    bool                ProcessTextures   ( const CFileIWF& File );
    bool                ProcessEntities   ( const CFileIWF& File );
    bool                ProcessReference  ( const ReferenceEntity& Reference,
                                            const D3DXMATRIX & mtxWorld );
    bool                ProcessSkyBox     ( const SkyBoxEntity& SkyBox );
    long                AddMesh           ( ULONG Count = 1 );
    long                AddObject         ( ULONG Count = 1 );
```

Note that there is an AddObject function and an AddMesh function which are used for adding CObject structures to the scene's CObject array and CTriMesh objects to the scene's CTriMesh array respectively.

The first new class member variable is a single statically allocated CTriMesh object that will be used to store a skybox. Our scene will contain one skybox (at most) and it will be manually created when necessary. GILES™ does not export geometry information for a skybox, only the six texture file names needed to create the effect. So when the scene class encounters a skybox entity in the IWF file, it will extract the texture file names, create the textures and then create the cube faces manually. The SkyBox mesh will be created in managed mode so that it will render in a self-contained manner.

```
    //-----------------------------------------------------------------------
    // Private Variables for This Class
    //-----------------------------------------------------------------------
    CTriMesh            m_SkyBoxMesh;               // Sky box mesh.
```

The scene class will also be responsible for resource allocation and management regardless of the mesh management mode selected. A callback mechanism is provided to allow scene loading functions access to texture resources when needed. In order to avoid duplicating texture resources, the scene will store the texture pointer along with its filename in the structure shown below.

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

The global resource pool for textures is stored in the CScene class as a single array:

```
    TEXTURE_ITEM       *m_pTextureList [MAX_TEXTURES];      // Array of texture pointers
```

In the case of a managed mesh, the scene will return a pointer to the texture back to the mesh loading function after it has added the texture to this array. The managed mesh can store the texture pointer in its attribute data array for later use. In the case of a non-managed mesh, the texture will still be added to this array by the callback function, but the pointer will not be returned to the mesh loader. Instead, the index of a texture/material combination will be returned and used to re-map the mesh attribute buffer.

A similar strategy to avoid duplication is used for material resources, but only for non-managed meshes. Managed meshes will store their own materials internally, although as an exercise, you should be able to quickly modify the class to reference the global material pool and reduce memory requirements. The scene material list will contain all materials used by non-managed meshes.

```
    D3DMATERIAL9         m_pMaterialList[MAX_MATERIALS];    // Array of material structures.
```

The scene class also uses a D3DLIGHT9 array to store the lights that were loaded from the IWF file. To simplify the code, this particular demo project does not use light groups, so only the first N lights are loaded from the file (where N is the total number of simultaneous lights supported by the device). You could certainly add light group support as an exercise however.

```
    D3DLIGHT9            m_pLightList   [MAX_LIGHTS];          // Array of light structures
```

The next three variables describe how many items are in the arrays just discussed.

```
    ULONG                m_nTextureCount;           // Number of textures stored
    ULONG                m_nMaterialCount;          // Number of materials stored
    ULONG                m_nLightCount;             // Number lights stored here
```

The space scene IWF file used in this project contains three space ships and a number of asteroids. The scene object will thus contain an array of CObject structures for each object in the scene that needs to be rendered. The CObject structure contains a world matrix and a pointer to a CTriMesh. Recall that multiple CObject's may reference the same CTriMesh. This is the approach used for our asteroids and is a technique referred to as mesh instancing. The scene also contains an array of CTriMesh objects for the

actual geometry data used by the scene objects. For example, the CTriMesh array will contains 4 meshes (3 space ship meshes and 1 asteroid mesh).

```
    CObject           **m_pObject;             // Array of objects storing meshes
    ULONG               m_nObjectCount;        // Number of objects currently stored
    CTriMesh          **m_pMesh;               // Array of loaded scene meshes
    ULONG               m_nMeshCount;          // Number of meshes currently stored
```

Non-managed meshes are not aware of the texture or material each of its subsets is using, since this information is managed at the scene level. The mesh object attribute buffer IDs will simply reference a texture and material combination stored in the CScene ATTRIBUTE_ITEM array shown next.

```
    ATTRIBUTE_ITEM      m_pAttribCombo[MAX_ATTRIBUTES]; // Table of attribute combinations
    ULONG               m_nAttribCount;                 // Number of attributes.
```

The ATTRIBUTE_ITEM structure represents a unique texture/material pair. This allows us to batch subsets across mesh boundaries when they share the same attribute combination (used by non-managed meshes only).

```
typedef struct _ATTRIBUTE_ITEM
{
    long TextureIndex;          // Index into the texture array
    long MaterialIndex;         // Index into the material array
} ATTRIBUTE_ITEM;
```

When a matching attribute item is found during the mesh loading callback function for non-managed meshes, an index into this global array is returned and stored in the mesh attribute buffer. If a match was not found, a new ATTRIBUTE_ITEM is created and inserted into the list and its index returned. The global texture and material arrays are used to avoid resource duplication as mentioned previously (note that the ATTRIBUTE_ITEM members are indices into the scene global texture and material arrays).

Finally, the scene stores a default material that will be used for any faces that do not have a material explicitly assigned in the file.

```
    D3DMATERIAL9        m_DefaultMaterial;   // A plain white material for null cases.
};
```

Now that we have examined the CScene member variables, let us look at the member functions. We will try to do this in the approximate order that they would be called so that we can better understand how the scene is built.

CScene::LoadSceneFromIWF

This function manages the loading of IWF files from disk. We pass in the filename of the IWF file we wish to load (with absolute or relative path) as well as the device light limit and a count for reserved light slots for dynamic lights. This project will not use light groups so we can ignore the third parameter.

The first thing the function does is instantiate a CFileIWF object provided us by the IWF SDK. We then strip off the filename portion of the input string and store the path portion in the m_strDataPath member variable. This path will be used later to load textures that are stored in the same folder.

```
bool CScene::LoadSceneFromIWF(TCHAR *strFileName, ULONG LightLimit, ULONG ReservedCount  )
{
    CFileIWF File;

    // Retrieve the data path
    if ( m_strDataPath ) free( m_strDataPath );
    m_strDataPath = _tcsdup( strFileName );

    // Strip off the filename
    TCHAR * LastSlash = _tcsrchr( m_strDataPath, _T('\\') );
    if (!LastSlash) LastSlash = _tcsrchr( m_strDataPath, _T('/') );
    if (LastSlash) LastSlash[1] = _T('\0'); else m_strDataPath[0] = _T('\0');
```

The next step is loading the IWF file from disk using the CFileIWF::Load function.

```
    // File loading may throw an exception
    try
    {
        // Attempt to load the file
        File.Load( strFileName );
```

At this point, all of the meshes, materials, texture names, and entities have been loaded and are stored in a series of STL vectors internal to the CFileIWF object. We now record the input maximum light count in a scene member variable.

```
        // Store values
        m_nLightLimit     = LightLimit;
```

The remainder of the function calls the ProcessXX member functions to extract the stored data into application defined data types. The first two calls (ProcessMaterials/ProcessTextures) extract the scene materials and textures and store them in their respective global arrays.

Interestingly, the space scene IWF file that accompanies this demo does not contain material or texture information because it does not store any actual mesh data. Instead, all meshes are stored in the IWF file as external reference entities (X file names). The materials and texture names used by this scene will be stored in the referenced X files, so technically they do not need to be loaded for the IWF. However, we make the calls anyway to allow for scenes that do store such information. They simply load textures and materials stored in the CFileIWF object and add them to the scene texture and material arrays.

```
        // Process the materials and textures first (order is important)
        if (!ProcessMaterials( File )) return false;
        if (!ProcessTextures( File )) return false;
```

As you might expect, the same logic holds true for geometry data. Since the IWF file in this demo does not store any meshes, the function will actually wind up doing nothing, but we implement it anyway to allow for cases where mesh data is exported.

```
        // Process the pure mesh data
        if (!ProcessMeshes( File )) return false;
```

Now we are ready to parse the entities stored in the CFileIWF::m_vpEntityList. This job falls to the CScene::ProcessEntities function. While we have looked at this function in previous demos, its only responsibility was extracting scene lights. This time however, we have two new entity types that we will need to write code for: references and skyboxes. In this demo, our reference entities will contain the filename for the X file(s) we wish to load to create the actual meshes in the scene. The skybox entity will store a list of the six required textures required.

```
        // Copy over the entities we want from the file
        if (!ProcessEntities( File )) return false;
```

Now that we have extracted all of the information that was stored in the IWF file, we can release the memory used by the CFileIWF STL vectors.

```
        // Allow file loader to release any active objects
        File.ClearObjects();
```

If the m_nLightCount member variable is still set to 0 at this point, then the IWF file contained no lighting information. To ensure that we are able to see our objects in this case, we setup four default directional lights and add them to the scene light array. This is not something that you must do but we do it here for convenience.

```
        // If no lights were loaded, lets default some half-way decent ones
        if ( m_nLightCount == 0 )
        {
            // Set up an arbitrary set of directional lights
            ZeroMemory( m_pLightList, 4 * sizeof(D3DLIGHT9));
            m_pLightList[0].Type = D3DLIGHT_DIRECTIONAL;
            m_pLightList[0].Diffuse = D3DXCOLOR( 1.0f, 0.92f, 0.82f, 0.0f );
            m_pLightList[0].Specular = D3DXCOLOR( 1.0f, 0.92f, 0.82f, 0.0f );
            m_pLightList[0].Direction = D3DXVECTOR3( 0.819f, -0.573f, 0.0f );

            m_pLightList[1].Type = D3DLIGHT_DIRECTIONAL;
            m_pLightList[1].Diffuse = D3DXCOLOR( 0.4f, 0.4f, 0.4f, 0.0f );
            m_pLightList[1].Specular = D3DXCOLOR( 0.6f, 0.6f, 0.6f, 0.0f );
            m_pLightList[1].Direction = D3DXVECTOR3( -0.819f, -0.573f, -0.0f );

            m_pLightList[2].Type = D3DLIGHT_DIRECTIONAL;
            m_pLightList[2].Diffuse = D3DXCOLOR( 0.8f, 0.8f, 0.8f, 0.0f );
            m_pLightList[2].Specular = D3DXCOLOR( 0.0f, 0.0f, 0.0f, 0.0f );
            m_pLightList[2].Direction = D3DXVECTOR3( 0.0f, 0.707107f, -0.707107f );

            m_pLightList[3].Type = D3DLIGHT_DIRECTIONAL;
            m_pLightList[3].Diffuse = D3DXCOLOR( 0.6f, 0.6f, 0.6f, 0.0f );
            m_pLightList[3].Specular = D3DXCOLOR( 0.0f, 0.0f, 0.0f, 0.0f );
```

```
            m_pLightList[3].Direction = D3DXVECTOR3( 0.0f, 0.707107f, 0.707107f );

            // We're now using 4 lights
            m_nLightCount = 4;

        } // End if no lights

    } // End Try Block

    // Catch any exceptions
    catch (...)
    {
        return false;

    } // End Catch Block

    // Success!
    return true;
}
```

CScene::ProcessMaterials

This function (called by the CScene::LoadSceneFromIWF) loops through each material in the CFileIWF::m_vpMaterialList vector and copies the values into the scene material list. The internal material counter is incremented for each material copied.

```
bool CScene::ProcessMaterials( const CFileIWF& File )
{
    ULONG i;

    // Loop through and build our materials
    for ( i = 0; i < File.m_vpMaterialList.size(); i++ )
    {
        // Retrieve pointer to file material
        iwfMaterial * pFileMaterial = File.m_vpMaterialList[i];

        // Retrieve pointer to our local material
        D3DMATERIAL9 * pMaterial = &m_pMaterialList[i];

        // Copy over the data we need from the file material
        pMaterial->Diffuse  = (D3DCOLORVALUE&)pFileMaterial->Diffuse;
        pMaterial->Ambient  = (D3DCOLORVALUE&)pFileMaterial->Ambient;
        pMaterial->Emissive = (D3DCOLORVALUE&)pFileMaterial->Emissive;
        pMaterial->Specular = (D3DCOLORVALUE&)pFileMaterial->Specular;
        pMaterial->Power    = pFileMaterial->Power;

        // Increase internal vars
        m_nMaterialCount++;
        if ( m_nMaterialCount >= MAX_MATERIALS ) break;

    } // Next Material

    // Success!
    return true;
}
```

CScene::ProcessTextures

This function (called by CScene::LoadScene from IWF) loops through the CFileIWF::m_vpTextureList and uses the stored filenames to load the texture resources. This function does not need to worry about duplicates because IWF files only store unique texture filenames. As each texture is loaded, its IDirect3DTexture9 interface pointer and filename is stored in the scene objects m_pTextureList array.

The first thing we do is zero out the scene TEXTURE_ITEM array.

```
bool CScene::ProcessTextures( const CFileIWF& File )
{
    ULONG          i;
    TCHAR          Buffer[MAX_PATH];
    TEXTURE_ITEM * pNewTexture = NULL;

    ZeroMemory( m_pTextureList, m_nTextureCount * sizeof(TEXTURE_ITEM*));
```

Now we will loop through each TEXTURE_REF in m_vpTextureList. The TEXTURE_REF structure is defined in libIWF.h and contains the texture filename and possibly even the actual texture pixel data. We are interested only in the filename (and its string length) for now.

```
for ( i = 0; i < File.m_vpTextureList.size(); i++ )
    {
        // Retrieve pointer to file texture
        TEXTURE_REF * pFileTexture = File.m_vpTextureList[i];

        // Skip if this is an internal texture (not supported by this demo)
        if ( pFileTexture->TextureSource != TEXTURE_EXTERNAL ) continue;
```

As we have found a potentially valid texture, we will create a new TEXTURE_ITEM to store the filename and texture we are about to create. We add the item to the CScene::m_pTextureList array.

```
        // No texture found, lets create our texture data and store it
        pNewTexture = new TEXTURE_ITEM;
        ZeroMemory( pNewTexture, sizeof(TEXTURE_ITEM) );
```

The TEXTURE_REF structure stores only the filename and not the full path. This allows us to store the texture in any folder desired. We extracted the path information during the LoadSceneFromIWF function, so we can add the path string to the texture file name to get the full path for the texture for loading. We use D3DXCreateTextureFromFileEx to load the texture.

```
        // Now build the full path
        _tcscpy( Buffer, m_strDataPath );
        _tcscat( Buffer, pFileTexture->Name );

        // Create the texture (use 3 mip levels max) (Ignore error value)
        D3DXCreateTextureFromFileEx( m_pD3DDevice, Buffer, D3DX_DEFAULT, D3DX_DEFAULT, 3, 0,
                                     m_fmtTexture,D3DPOOL_MANAGED,D3DX_DEFAULT,D3DX_DEFAULT,
                                     0, NULL, NULL, &pNewTexture->Texture );
```

We copy the texture filename into the TEXURE_ITEM, add the structure to our global array, and then increment our counter.

```
        // Duplicate the filename for future lookups
        pNewTexture->FileName = _tcsdup( pFileTexture->Name );

        // Store this item
        m_pTextureList[ m_nTextureCount++ ] = pNewTexture;
        if ( m_nTextureCount >= MAX_TEXTURES ) break;

    } // Next Texture

    // Success!
    return true;
}
```

CScene::ProcessMeshes

ProcessMeshes extracts geometry from CFileIWF::m_vpMeshList and creates CTriMesh objects for each mesh. The m_vpMeshList vector contains one iwfMesh structure for each mesh extracted from the file (see libIWF.h).

```
bool CScene::ProcessMeshes( CFileIWF & pFile )
{
    HRESULT     hRet;
    CTriMesh * pNewMesh = NULL;
    long        i, j, MaterialIndex, TextureIndex;
    ULONG       AttribID = 0;

    // Loop through each mesh in the file
    for ( i = 0; i < pFile.m_vpMeshList.size(); i++ )
    {
        iwfMesh * pMesh = pFile.m_vpMeshList[i];

        // Allocate a new CTriMesh
        pNewMesh = new CTriMesh;
        if ( !pNewMesh ) return false;
```

We are going to populate mesh data buffers manually, so we need to tell the mesh about our vertex components and our index format so that it can properly allocate its vertex/index buffers. In this project, VERTEX_FVF is defined in CObject.h as:

```
#define VERTEX_FVF      D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1
```

```
        // Set the mesh's data format
        pNewMesh->SetDataFormat( VERTEX_FVF, sizeof(USHORT) );
```

Now that we have created the CTriMesh, we can loop through each surface in the iwfMesh and add the geometry data. Note that we skip surfaces marked with the SURFACE_INVISIBLE flag.

```
        for ( j = 0; j < pMesh->SurfaceCount; j++ )
```

```
        {
            iwfSurface * pSurface = pMesh->Surfaces[j];

            // Skip if this surface is flagged as invisible
            if ( pSurface->Style & SURFACE_INVISIBLE ) continue;
```

First we extract the texture and material indices used by the face setting the indices to –1 if some error has occurred (e.g., the texture index has a larger value than the total number of textures stored in the IWF file). This would mean the IWF file has been created incorrectly. We do not halt execution but instead assign an idex of –1 indicating that the scene's default (white) material will need to be assigned to these faces.

```
           // Determine the indices we are using.
           MaterialIndex = -1;
           TextureIndex  = -1;
          if ( (pSurface->Components & SCOMPONENT_MATERIALS) && pSurface->ChannelCount > 0 )
             MaterialIndex = pSurface->MaterialIndices[0];

          if ( (pSurface->Components & SCOMPONENT_TEXTURES ) && pSurface->ChannelCount > 0 )
             TextureIndex  = pSurface->TextureIndices[0];

          if ( MaterialIndex >= m_nMaterialCount ) MaterialIndex = -1;
          if ( MaterialIndex >= m_nTextureCount  ) TextureIndex  = -1;
```

With the material and texture indices stored in temporary local variables (MaterialIndex and TextureIndex), we use them to get a pointer to the material in the scene material array and the texture filename for the texture stored in the scene texture array (shown below).

```
        LPCTSTR        strTextureFile = NULL;
        D3DMATERIAL9 * pMaterial      = NULL;

        // Retrieve information to pass to support functions
        if ( MaterialIndex >= 0 ) pMaterial = &m_pMaterialList[MaterialIndex];
        if ( TextureIndex  >= 0 ) strTextureFile = m_pTextureList[TextureIndex]->FileName;
```

Bear in mind when looking at the above code that when this function has been called, the ProcessMaterials and ProcessTextures function have already been executed. Therefore, all textures and materials stored inside the IWF file are already in the scene texture and material lists at this point. This means the texture and material index stored in each IWFSurface indexes correctly into the scene's texture and material arrays. This is because they will have been added to the scene in the same order as the materials and textures listed inside the IWF file. The texture and material indices stored in each IWFSurface are therefore still valid when used to access the scene textures and materials.

Now that we have a pointer to the material and the texture filename used by this face, we pass this information into the CScene::CollectAttributeID function. The function will look for a matching material/texture combination in its ATTRIBUTE_ITEM array and if one is not found, create a new entry, loading and creating any textures and materials as necessary. This is the same function that is used as a callback function for non-managed mode meshes when loading X file data.

The function returns the index of the ATTRIBUTE_ITEM which describes the global attribute ID that will be assigned to this face when it is added to the CTriMesh. Notice that we also pass in the 'this' pointer because CollectAttributeID is a static function.

```
        // Collect an attribute ID
        AttribID = CollectAttributeID( this, strTextureFile, pMaterial );
```

Now that we have the attribute index for this material/texture combination, we can add the face indices to the CTriMesh index buffer. For each triangle added (because this may be an N-gon), we also copy the attribute ID into the CTriMesh attribute buffer so that all triangles that share the same material and texture will belong to the same subset. This is all handled by the ProcessIndices function. We pass in a pointer to our new CTriMesh, a pointer to the iwfSurface that contains the index data for the face we wish to add, and the face attribute ID that we have just generated.

```
        // Process the indices
        if (!ProcessIndices( pNewMesh, pSurface, AttribID ) ) break;
```

If the surface has the SURFACE_TWO_SIDED flag set, then the level designer wants this surface to be visible from both its front and back sides. As our application uses back face culling, we add the face to the mesh again, this time passing in TRUE as the final parameter. This call reverses the winding order of the face before it adds it to the index buffer the second time. We have essentially created two polygons that share the same position in 3D space but face in opposing directions.

```
        if ( pSurface->Style & SURFACE_TWO_SIDED )
        {
            // Two sided surfaces have back faces added manually
            if (!ProcessIndices( pNewMesh, pSurface, AttribID, true ) ) break;
        }
```

Now we call ProcessVertices to add the vertex data to the new mesh.

```
           // Process vertices
           if (!ProcessVertices( pNewMesh, pSurface ) ) break;

    } // Next Surface
```

After the CScene::ProcessVertices and CScene::ProcesIndices have been called, the CTriMesh will have had all the vertex, index and attribute data added to its temporary storage bins.

If one of the processing functions failed, then the loop variable 'j' will be smaller than the surface count of the iwfMesh. If this is the case, then we release the new CTriMesh and return failure as we have data corruption.

```
        // If we didn't reach the end, then we failed
        if ( j < pMesh->SurfaceCount ) { delete pNewMesh; return false; }
```

We next instruct the CTriMesh object to build its underlying ID3DXMesh with a call to the CTriMesh::BuildMesh function. We pass in the D3DXMESH_MANAGED creation flag indicating our desire for allocating index and vertex buffers in the managed resource pool.

```
        // We're done, attempt to build this mesh
        hRet = pNewMesh->BuildMesh( D3DXMESH_MANAGED, m_pD3DDevice );
        if ( FAILED(hRet) ) { delete pNewMesh; return false; }
```

Next we clean and optimize the CTriMesh. We start with the CTriMesh::Weld function, passing in a 0.0 floating point tolerance so that only exact duplicated vertices are merged. Then we call the CTriMesh::OptimizeInPlace function to perform compaction, attribute sorting, and vertex cache optimizations on the underlying D3DXMesh data.

```
        // Optimize the mesh of possible
        pNewMesh->WeldVertices( 0 );
        pNewMesh->OptimizeInPlace( D3DXMESHOPT_VERTEXCACHE );
```

We now call the CScene::AddMesh() function, which is a simple utility function to resize the scene CTriMesh array (m_pMesh). This call returns the index of the newly added element, which we use to store the CTriMesh pointer to the mesh we have just created.

```
        // Store this new mesh
        if ( AddMesh( ) < 0 ) { delete pNewMesh; return false; }
        m_pMesh[ m_nMeshCount - 1 ] = pNewMesh;
```

Although we have added the mesh to the scene, we still require a higher level object to allow for mesh instancing, so we allocate a new CObject structure, passing our new mesh into the constructor. Note that while the CObject maintains a world matrix, meshes stored in IWF files exported from GILES™ are all defined in world space. So the CObject world matrix should just be set to identity, and this is done implicitly by the constructor.

```
        // Now build an object for this mesh (standard identity)
        CObject * pNewObject = new CObject( pNewMesh );
        if ( !pNewObject ) return false;
```

Finally, we add this new CObject to the scene level CObject array (m_pObject). The AddObject() member function resizes the array and returns the index where we will copy the pointer .

```
        // Store this object
        if ( AddObject() < 0 ) { delete pNewObject; return false; }
        m_pObject[ m_nObjectCount - 1 ] = pNewObject;

    } // Next Mesh

    // Success!!
    return true;
}
```

CScene::ProcessVertices

ProcessVertices is called by ProcessMeshes (discussed above) for each surface in the mesh. Its job is to add the vertices stored in the iwfSurface to the newly created CTriMesh object. Note that they are not copied straight into the vertex buffer because the ID3DXMesh will not have been created yet. Instead they are copied into the CTriMesh::pVertices array. Only after all vertices have been added and the underlying ID3DXMesh has been created will we fill the vertex buffer with this data.

```
bool CScene::ProcessVertices( CTriMesh * pMesh, iwfSurface * pFilePoly )
{
    ULONG i, VertexStart = pMesh->GetNumVertices();
    CVertex * pVertices = NULL;

    // Allocate enough vertices
    if ( pMesh->AddVertex( pFilePoly->VertexCount ) < 0 ) return false;
    pVertices = (CVertex*)pMesh->GetVertices();
```

We start by retrieving the number of vertices currently stored in the mesh. This tells us where we want to start appending our new vertex data. Next we call the CTriMesh::AddVertex function passing in the number of vertices in the iwfSurface. This function resizes the mesh's temporary vertex array if necessary to make room for the new data. We then get a pointer to the mesh vertex array so that we can start copying the vertex data.

```
    // Loop through each vertex and copy required data.
    for ( i = 0; i < pFilePoly->VertexCount; i++ )
    {
        // Copy over vertex data
        pVertices[i + VertexStart].x      = pFilePoly->Vertices[i].x;
        pVertices[i + VertexStart].y      = pFilePoly->Vertices[i].y;
        pVertices[i + VertexStart].z      = pFilePoly->Vertices[i].z;
        pVertices[i + VertexStart].Normal = (D3DXVECTOR3&)pFilePoly->Vertices[i].Normal;

        // If we have any texture coordinates, set them
        if ( pFilePoly->TexChannelCount > 0 && pFilePoly->TexCoordSize[0] == 2 )
        {
            pVertices[i + VertexStart].tu = pFilePoly->Vertices[i].TexCoords[0][0];
            pVertices[i + VertexStart].tv = pFilePoly->Vertices[i].TexCoords[0][1];
        } // End if has tex coordinates
    } // Next Vertex

    // Success!
    return true;
}
```

CScene::ProcessIndices

ProcessIndices remains relatively unchanged from our prior projects so we will only show the code that deals with face indices in indexed triangle list format. This means we simply copy the indices from the passed iwfSurface into the mesh. Please refer to Chapter Five for code that converts the other formats.

```
bool CScene::ProcessIndices( CTriMesh * pMesh, iwfSurface * pFilePoly,
                             ULONG AttribID, bool BackFace )
{
    ULONG    i, Counter, VertexStart, FaceStart, IndexStart;
    USHORT * pIndices = NULL;

    // Validate parameters
    if (!pMesh || !pFilePoly) return false;
```

Before we add indices to the new CTriMesh object, we need to retrieve the mesh vertex count so that we can correctly offset the zero-based index values. Keep in mind that this function is called before the vertices are processed, so if we did not do this, regardless of how many vertices we added, each triangle would refernce the first three vertices. We also also fetch the number of triangles currently stored in the CTriMesh so that we can correctly calculate the offset in the mesh index array for appending the new indices.

```
    // Store current Mesh vertex and face count
    VertexStart = pMesh->GetNumVertices();
    FaceStart   = pMesh->GetNumFaces();
    IndexStart  = FaceStart * 3;
```

If the iwfSurface has a non-zero index count then the face includes indices. If not, then the function will need to generate indices for the appropriate polygon type (see Chapter Five). All we have to do for the current case we are looking at is copy the index data.

```
    // Generate indices
    if ( pFilePoly->IndexCount > 0 )
    {
        ULONG IndexType = pFilePoly->IndexFlags & INDICES_MASK_TYPE;

        // Interpret indices (we want them in tri-list format)
        switch ( IndexType )
        {
             case INDICES_TRILIST:

                // We can do a straight copy (converting from 32bit to 16bit)
                if ( pMesh->AddFace( pFilePoly->IndexCount / 3, NULL, AttribID ) < 0 )
                       return false;

                pIndices = (USHORT*)pMesh->GetFaces( );

                // Copy over the face data
                for ( i = 0; i < pFilePoly->IndexCount; ++i )
                       pIndices[i + IndexStart] = pFilePoly->Indices[i] + VertexStart;

             break;
```

After the indices have been added, we test the BackFace Boolean passed into the function. If TRUE, then we need to reverse the order of the indices just added and re-copy them. This approach is used when we encounter a two sided face in the IWF file.

```
  // We now have support for adding the same polygon again, but in
  // reverse order to add a renderable back face if disabling culling
  // is not a viable option.
  if ( BackFace == true )
  {
        // If we specified back faces, invert all the indices recently added
               pIndices  = (USHORT*)pMesh->GetFaces( );
        for ( i = IndexStart; i < pMesh->GetNumFaces() * 3; i+=3 )
        {
               USHORT iTemp  = pIndices[i];
               pIndices[i]   = pIndices[i+2];
               pIndices[i+2] = iTemp;

        } // Next Tri

  } // End if back face

    // Success!
    return true;
}
```

CScene::CollectAttributeID

This function is called from two possible places. CTriMesh::LoadMeshFromX calls it to process a material/texture combination extracted from an X file. CScene::ProcessMeshes calls it to process a material/texture combination used by an iwfSurface. The function is responsible for making sure that textures and materials are not duplicated and it returns the global attribute ID for subsets in non-managed meshes.

The first parameter is a void pointer to a context. Remember that this function was declared as a static member function so that it can double as a callback function for non-managed mode meshes when needed. We use the context parameter in the ProcessMeshes function to pass the 'this' pointer so that there can be access to non-static member variables. The next three parameters are what we will match against when searching for the appropriate combination of attributes.

```
ULONG CScene::CollectAttributeID( LPVOID pContext, LPCTSTR strTextureFile,
                                  const D3DMATERIAL9 * pMaterial,
                                  const D3DXEFFECTINSTANCE * pEffectInstance )
{
    TEXTURE_ITEM * pNewTexture = NULL;
    ULONG          i;
    long           TextureMatch = -1, MaterialMatch = -1;
    bool           TexMatched = false, MatMatched = false;

    // Validate parameters
    if ( !pContext ) return 0;

    // Retrieve the scene object
    CScene *pScene = (CScene*)pContext;
```

We cast the context pointer to a CScene pointer so that we have access to the texture and materials arrays for the correct scene object instance. We can now loop through the scene ATTRIBUTE_ITEM array to see if an element with the passed texture/material combination already exists. The process should be relatively easy to follow, so we will not explain it in great detail. We start by checking texture filenames and then the material pointers.

```
    // Loop through the attribute combination table to see if one already exists
    for ( i = 0; i < pScene->m_nAttribCount; ++i )
    {
        ATTRIBUTE_ITEM * pAtrItem = &pScene->m_pAttribCombo[i];
        long TextureIndex  = pAtrItem->TextureIndex;
        long MaterialIndex = pAtrItem->MaterialIndex;

        TEXTURE_ITEM   * pTexItem = NULL;
        D3DMATERIAL9   * pMatItem = NULL;

        // Retrieve pointers
        if ( TextureIndex  >= 0 ) pTexItem = pScene->m_pTextureList[ TextureIndex ];
        if ( MaterialIndex >= 0 ) pMatItem = &pScene->m_pMaterialList[ MaterialIndex ];

        // Neither are matched so far
        TexMatched = false;
        MatMatched = false;

        // If both sets are NULL, this is a match, otherwise perform the real match
        if ( pTexItem == NULL && strTextureFile == NULL )
            TexMatched = true;

        else if ( pTexItem != NULL && strTextureFile != NULL )
        if ( _tcsicmp( pTexItem->FileName, strTextureFile ) == 0 ) TexMatched = true;

        if ( pMatItem == NULL && pMaterial == NULL )
            MatMatched = true;
        else if ( pMatItem != NULL && pMaterial != NULL )

        if ( memcmp( pMaterial, pMatItem, sizeof(D3DMATERIAL9) ) == 0 ) MatMatched = true;

        // Store the matched indices in case we can use the later on
        if ( TexMatched ) TextureMatch = TextureIndex;
        if ( MatMatched ) MaterialMatch = MaterialIndex;
```

If we have both a texture match and a material match, then we return the index of this attribute item to the caller. The CTriMesh::LoadMeshFromX function uses this index to remap its attribute buffer from mesh local attribute IDs to scene attribute IDs.

```
        // If they both matched up at the same time, we have a winner
        if ( TexMatched == true && MatMatched == true ) return i;

    } // Next Attribute
```

If we cannot find a match on both parameters (texture and material), then we will need to add a new ATTRIBUTE_ITEM structure to the scene array reflecting these attribute properties. We want to look

for individual matches on texture or material first to avoid duplicating data, so we will traverse those arrays again and record the index of the matches if we find them.

```
    ATTRIBUTE_ITEM AttribItem;

    if ( MaterialMatch < 0 && pMaterial != NULL )
    {
        for ( i = 0; i < pScene->m_nMaterialCount; ++i )
        {
            // Is there a match ?
            if ( memcmp( pMaterial, &pScene->m_pMaterialList[i], sizeof(D3DMATERIAL9)) == 0 )
                 MaterialMatch = i;

            // If we found a match, bail
            if ( MaterialMatch >= 0 ) break;

        } // Next Attribute Combination

    } // End if no material match

    if ( TextureMatch < 0 && strTextureFile != NULL )
    {
        for ( i = 0; i < pScene->m_nTextureCount; ++i )
        {
            if (!pScene->m_pTextureList[i] || !pScene->m_pTextureList[i]->FileName) continue;

            // Is there a match ?
            if ( _tcsicmp( strTextureFile, pScene->m_pTextureList[i]->FileName ) == 0 )
                 TextureMatch = i;

            // If we found a match, bail
            if ( TextureMatch >= 0 ) break;

        } // Next texture

    } // End if no Texture match
```

If matches are found (for either type), we simply copy the index value from the local variable into the new ATTRIBUTE_ITEM. When a match is not found, we must load the data into our global lists and record those indices instead. Materials will simply be copied, textures will require loading.

```
    // Now build the material index, or add if necessary
    if ( MaterialMatch < 0 && pMaterial != NULL )
    {
        AttribItem.MaterialIndex = pScene->m_nMaterialCount;
        pScene->m_pMaterialList[ pScene->m_nMaterialCount++ ] = *pMaterial;
    } // End if no material match
    else
    {
        AttribItem.MaterialIndex = MaterialMatch;
    } // End if material match
```

If a matching texture was not found in the scene texture array then we need to create a new texture with a call to the CollectTexture function. This function (also used as the texture callback function for managed mode meshes in the CTriMesh::LoadMeshFromX function) will load the texture if it does not yet exist

and will add it to the end of the scene texture array. If a texture was found, we copy this index into the attribute item structure as shown below.

```
    // Now build the texture index, or add if necessary
    if ( TextureMatch < 0 && strTextureFile != NULL )
    {
        // We know it doesn't exist, but we can still use
        // collect texture to do the work for us.
        AttribItem.TextureIndex = pScene->m_nTextureCount;
        CollectTexture( pScene, strTextureFile );

    } // End if no texture match
    else
    {
        AttribItem.TextureIndex = TextureMatch;

    } // End if Texture match
```

Finally, we add the new item to the scene m_pAttribCombo array and return the index of this new attribute.

```
    // Store this new attribute combination item
    pScene->m_pAttribCombo[ pScene->m_nAttribCount++ ] = AttribItem;

    // Return the new attribute index
    return pScene->m_nAttribCount - 1;
}
```

CScene::CollectTexture

This function searches the scene texture array to determine whether a texture with the passed texture filename already exists. If so, a pointer to this texture item is returned. Otherwise, a new texture is created, inserted into the array, and its pointer is returned.

CollectTexture is called from a number of places in our code. If we wish to load mesh data from an X file into a managed mode CTriMesh, we can register this function as the texture callback so that texture filenames returned from D3DXLoadMeshFromX can be mapped to textures in our scene texture array. The CollectAttributeID function, which can serve as a callback for non-managed meshes, calls this function to load its textures as well. We have just seen how ProcessMeshes calls the CollectAttributeID function (which in turn calls CollectTexture) to add the textures in an IWF file to the scene texture list.

```
LPDIRECT3DTEXTURE9 CScene::CollectTexture( LPVOID pContext, LPCTSTR FileName )
{
    HRESULT        hRet;
    TEXTURE_ITEM * pNewTexture = NULL;

    // Validate parameters
    if ( !pContext || !FileName ) return NULL;

    // Retrieve the scene object
```

```
    CScene *pScene = (CScene*)pContext;
```

Once we have a pointer to the scene instance, we can loop through its texture array and compare each texture name with the texture name passed into the function. If a match is found then the texture already exists and we immediately return a pointer to the texture.

```
    // Loop through and see if this texture already exists.
    for ( ULONG i = 0; i < pScene->m_nTextureCount; ++i )
    {
        if ( _tcsicmp( pScene->m_pTextureList[i]->FileName, FileName ) == 0 )
            return pScene->m_pTextureList[i]->Texture;

    } // Next Texture
```

If we exit the loop, then we know that a texture with a matching filename does not yet exist. Therefore, we need to create and initialize a new TEXTURE_ITEM structure that will be added to the texture array. Note that the texture filename passed into the function will not contain any path information. So in order to load the image file into a texture, we build the complete filename string with the full path. We will store the texture filename without a path in the TEXTURE_ITEM structure for future searches.

```
    // No texture found, so lets create and store it.
    pNewTexture = new TEXTURE_ITEM;
    if (!pNewTexture) return NULL;
    ZeroMemory( pNewTexture, sizeof(TEXTURE_ITEM) );

    // Build filename string
    TCHAR Buffer[MAX_PATH];
    _tcscpy( Buffer, pScene->m_strDataPath );
    _tcscat( Buffer, FileName);

    // Create the texture (use 3 mip levels max)
    hRet = D3DXCreateTextureFromFileEx(pScene->m_pD3DDevice, Buffer, D3DX_DEFAULT,
                                       D3DX_DEFAULT, 3, 0, pScene->m_fmtTexture,
                                       D3DPOOL_MANAGED, D3DX_DEFAULT, D3DX_DEFAULT,
                                       0, NULL, NULL, &pNewTexture->Texture );

    if (FAILED(hRet)) { delete pNewTexture; return NULL; }

    // Duplicate the filename for future lookups
    pNewTexture->FileName = _tcsdup( FileName );
```

Finally we add the new TEXTURE_ITEM to the scene array, increment the texture count, and return the new texture pointer.

```
    // Store this item
    pScene->m_pTextureList[ pScene->m_nTextureCount++ ] = pNewTexture;

    // Return the texture pointer
    return pNewTexture->Texture;
}
```

CScene::ProcessEntities

Our current project will support three different entity types: lights, skyboxes, and references. This function is responsible for navigating the CFileIWF::m_vpEntityList vector and extracting these types as they are encountered.

The first part of the function tests for light entities. The light entity is one of the IWF standard entity types and it is defined in the IWF SDK as an entity with an ID of 16:

```
#define ENTITY_LIGHT                          0x0010
```

```
bool CScene::ProcessEntities( const CFileIWF& File )
{
    ULONG      i, j;
    D3DLIGHT9  Light;
    USHORT     StringLength;
    bool       SkyBoxBuilt = false;

    // Loop through and build our lights & references
    for ( i = 0; i < File.m_vpEntityList.size(); i++ )
    {
        // Retrieve pointer to file entity
        iwfEntity * pFileEntity = File.m_vpEntityList[i];

        // Skip if there is no data
        if ( pFileEntity->DataSize == 0 ) continue;
```

We will extract light information from the entity data area directly into a D3DLIGHT9 structure and add the new light to our scene light array. In this project we will only process the light if we have not yet added the maximum number of lights to the array, otherwise it will be ignored.

```
        // Decide what to do here (skip all lights if we've already reached our limit)
        if ( (m_nLightCount < m_nLightLimit && m_nLightCount < MAX_LIGHTS)
              && pFileEntity->EntityTypeMatches( ENTITY_LIGHT ) )
        {
            LIGHTENTITY * pFileLight = (LIGHTENTITY*)pFileEntity->DataArea;
```

GILES™ can export ambient lights, but they are not particularly relevant in the DirectX lighting pipeline, so if the current light is an ambient light we will skip it

```
            // Skip if this is not a valid light type (Not relevant to the API)
            if ( pFileLight->LightType == LIGHTTYPE_AMBIENT ) continue;
```

Extracting the lighting information is very simple due to the fact that the LIGHTENTITY structure used by CFileIWF is almost identical to the D3DLIGHT9 structure.

```
            // Extract the light values we need
            Light.Type      = (D3DLIGHTTYPE)(pFileLight->LightType + 1);
            Light.Diffuse   = D3DXCOLOR(pFileLight->DiffuseRed, pFileLight->DiffuseGreen,
                                        pFileLight->DiffuseBlue, pFileLight->DiffuseAlpha);
```

```
            Light.Ambient   = D3DXCOLOR (pFileLight->AmbientRed, pFileLight->AmbientGreen,
                                         pFileLight->AmbientBlue, pFileLight->AmbientAlpha);

            Light.Specular  = D3DXCOLOR (pFileLight->SpecularRed, pFileLight->SpecularGreen,
                                         pFileLight->SpecularBlue,pFileLight->SpecularAlpha);
```

Every entity stores a world matrix, so we extract the bottom row to get the light position in world space.

```
            Light.Position  = D3DXVECTOR3(pFileEntity->ObjectMatrix._41,
                                          pFileEntity->ObjectMatrix._42,
                                          pFileEntity->ObjectMatrix._43 );
```

We also extract the look vector which describes the direction the light is pointing in world space.

```
            Light.Direction  = D3DXVECTOR3(pFileEntity->ObjectMatrix._31,
                                           pFileEntity->ObjectMatrix._32,
                                           pFileEntity->ObjectMatrix._33 );
```

The remaining light data is extracted and we add the light to our light array, incrementing the counter.

```
            Light.Range            = pFileLight->Range;
            Light.Attenuation0     = pFileLight->Attenuation0;
            Light.Attenuation1     = pFileLight->Attenuation1;
            Light.Attenuation2     = pFileLight->Attenuation2;
            Light.Falloff          = pFileLight->FallOff;
            Light.Theta            = pFileLight->Theta;
            Light.Phi              = pFileLight->Phi;

            // Add this to our array
            m_pLightList[ m_nLightCount++ ] = Light;

        } // End if light
```

Testing for skyboxes and reference entities requires a slightly different approach because these types are not defined by the IWF standard -- they are custom entity types defined by GILES™. The IWF specification provides applications the ability to specify their own entity types with their own entity IDs and chunk IDs provided they do not conflict with IDs reserved by the IWF standard.

To address the possibility of third party entities sharing the same IDs with each other, the IWF specification records an author ID in addition to the entity chunk IDs. For example, the skybox and reference entities both have the author ID for GILES™ embedded in those chunks. GILES™ uses a 5 BYTE author ID embedded in its custom chunks. Each byte is the ASCII code for the letters of its name. We define this array in CScene.cpp and we can use it to test if an entity is a custom GILES™ entity.

```
const UCHAR AuthorID[5] = { 'G', 'I', 'L', 'E', 'S' };
```

The iwfEntity class contains a helper function called EntityAuthorMatches. We pass in an array length and an array of bytes which will be compared against the entity's author ID, returning true if they match.

If the entity is not a light entity or a GILES™ custom entity (skybox/reference) we will ignore it.

```
        else if ( pFileEntity->EntityAuthorMatches( 5, AuthorID ) )
        {
```

At this point we do not know if this will be a skybox entity of a reference entity, so we will create two local variables that can be used to hold the information for both.

```
            SkyBoxEntity SkyBox;
            ZeroMemory( &SkyBox, sizeof(SkyBoxEntity) );

            ReferenceEntity Reference;
            ZeroMemory( &Reference, sizeof(ReferenceEntity ));
```

The SkyBoxEntity and ReferenceEntity structures are both defined in CScene.h.

```
typedef struct _ReferenceEntity
{
    ULONG       ReferenceType;          // What type of reference is this
    ULONG       Flags;                  // Reserved flags
    ULONG       Reserved1;              // Reserved values
    ULONG       Reserved2;              // Reserved values
    ULONG       Reserved3;              // Reserved values
    char        ReferenceName[1024];    // External file name

} ReferenceEntity;
```

The entity ID for the GILES™ reference entity is 0x203 (CScene.h).

```
#define CUSTOM_ENTITY_REFERENCE 0x203
```

Most of the members of the reference entity are reserved for later use. We are interested only in the reference type (0 for internal reference, 1 for external reference) and the ReferenceName. In this project we will work with external references only. The name of the referenced X file will be stored in the ReferenceName member.

The skybox entity stores a reserved DWORD followed by six texture filenames. When our application encounters one of these entities, we create the scene skybox mesh (m_SkyBoxMesh) as an inward facing cube, and then load the six textures using these filenames and map them to the cube faces.

```
typedef struct _SkyBoxEntity
{
    ULONG       Flags;                  // Reserved flags
    char        Textures[6][256];       // 6 Sets of external texture names
} SkyBoxEntity;
```

The entity ID for the GILES™ skybox entity is 0x202 (CScene.h).

```
#define CUSTOM_ENTITY_SKYBOX    0x202
```

To determine which entity type we have found, we retrieve a pointer to the entity data area and check its ID.

```
            // Retrieve data area
            UCHAR * pEntityData = pFileEntity->DataArea;
```

If it is a GILES™ reference entity then we extract the data into the temporary ReferenceEntity structure and pass it to CScene::ProcessReference for additional processing. Notice that the ProcessReference function accepts the ReferenceEntity and the entity world matrix. This matrix stores the world space position and orientation of the object that will ultimately be added to the scene for this reference. Every IWF entity contains a matrix describing the position and orientation of the entity in the scene. The following code shows how to extract all the entity information and pass it along to the ProcessReference function.

```
            switch ( pFileEntity->EntityTypeID )
            {
                case CUSTOM_ENTITY_REFERENCE:

                    // Copy over the the reference data
                    memcpy( &Reference.ReferenceType, pEntityData, sizeof(ULONG) );
                    pEntityData += sizeof(ULONG);

                    memcpy( &Reference.Flags, pEntityData, sizeof(ULONG) );
                    pEntityData += sizeof(ULONG);

                    memcpy( &StringLength, pEntityData, sizeof(USHORT) );
                    pEntityData += sizeof(USHORT);

                    if ( StringLength > 0 )
                       memcpy( Reference.ReferenceName, pEntityData, StringLength );
                    pEntityData += StringLength;

                    memcpy( &Reference.Reserved1, pEntityData, sizeof(ULONG) );
                    pEntityData += sizeof(ULONG);

                    memcpy( &Reference.Reserved2, pEntityData, sizeof(ULONG) );
                    pEntityData += sizeof(ULONG);

                    memcpy( &Reference.Reserved3, pEntityData, sizeof(ULONG) );
                    pEntityData += sizeof(ULONG);

                    // Process this reference (returns false only on fatal error)
                    if(!ProcessReference(Reference,(D3DXMATRIX&)pFileEntity->ObjectMatrix))
                        return false;

                    break;
```

For skybox entities we use a similar approach. We copy the data into the temporary SkyBoxEntity structure and pass it to ProcessSkyBox for final processing. Note that we only process the sky box entity if a sky box does not currently exist. This was done for simplicity only, so feel free to change this behavior to maintain an array of skyboxes that can be modified as you wish. Usually, a given scene will have one sky box defined, but this is not always necessarily the case.

```
                case CUSTOM_ENTITY_SKYBOX:
```

```
                    // We only want one skybox per file please! :)
                    if ( SkyBoxBuilt == true ) break;
                    SkyBoxBuilt = true;

                    // Copy over the skybox data
                    memcpy( &SkyBox.Flags, pEntityData, sizeof(ULONG) );
                    pEntityData += sizeof(ULONG);

                    // Read each of the 6 face textures
                    for ( j = 0; j < 6; ++j )
                    {
                        memcpy( &StringLength, pEntityData, sizeof(USHORT) );
                        pEntityData += sizeof(USHORT);

                        if ( StringLength > 0 )
                          memcpy( SkyBox.Textures[j], pEntityData, StringLength );
                        pEntityData += StringLength;

                    } // Next Face Texture

                    // Process this skybox (returns false only on a fatal error)
                    if ( !ProcessSkyBox( SkyBox ) ) return false;

                    break;

            } // End Entity Type Switch

        } // End if custom entities

    } // Next Entity

    // Success!
    return true;
}
```

CScene::ProcessReference

As mentioned previously, the space scene for this project stores all scene objects except for the skybox as external reference entities. These entities contain a world matrix and the filename for an X file which the object will use as its mesh.

The first thing this function does is check the scene mesh array to see if the mesh has already been loaded. This is done because there may be many reference entities which reference the same X file and we would prefer to avoid duplication, to keep memory footprint as small as possible. If the mesh already exists, then we will simply create a new CObject, instance the CTriMesh pointer, and then use the reference matrix for the object's world matrix. If the mesh does not currently exist, then we will need to create a new CTriMesh object and load the mesh data from the X file using the CTriMesh::LoadMeshFromX function, passing in the file name stored in the reference. It is in this function that we will see, for the first time, the registration of callback functions with the CTriMesh object. As discussed, these functions will automatically add the texture and materials in the X file to the scene level texture and material arrays.

The first thing the code does is test the ReferenceType member of the entity. If it is not set to 1 then this is an internal reference and we will ignore this entity and return. If it is set to 0, then we have an external reference, so we will build the full path and filename for the referenced X file (the reference entity will not contain any path information, only the name of the file).

```
bool CScene::ProcessReference(const ReferenceEntity& Reference, const D3DXMATRIX & mtxWorld)
{
    HRESULT     hRet;
    CTriMesh * pReferenceMesh = NULL;

    if (Reference.ReferenceType != 1) return true;

    // Build filename string
    TCHAR Buffer[MAX_PATH];
    _tcscpy( Buffer, m_strDataPath );
    _tcscat( Buffer, Reference.ReferenceName );
```

We now search the scene mesh array for a mesh with the same name as the file name. If a match is found, we will store a pointer to this matching mesh for now.

```
    // Search to see if this X file has already been loaded
    for ( ULONG i = 0; i < m_nMeshCount; ++i )
    {
        if (!m_pMesh[i]) continue;
        if ( _tcsicmp( Buffer, m_pMesh[i]->GetMeshName() ) == 0 ) break;
        } // Next Mesh

    // If we didn't reach the end, this mesh already exists
    if ( i != m_nMeshCount )
    {
        // Store reference mesh.
        pReferenceMesh = m_pMesh[i];

    } // End if mesh already exists
```

If the mesh is not already loaded then we will need to create a new CTriMesh object and call its LoadMeshFromX function to load the X file data into the mesh. Before we do that though, we register the CScene::CollectAttributeID function, which automatically places the mesh into non-managed mode. When the mesh loads the X file data, each texture and material loaded from the X file will be passed to the CollectAttributeID function. We saw earlier how this function adds the texture and material to the scene texture and material arrays and returns the index for this texture/material combo back to the mesh. The mesh will then use this index to remap its attribute buffer.

```
    else
    {
        // Allocate a new mesh for this reference
        CTriMesh * pNewMesh = new CTriMesh;

        // Load in the externally referenced X File
        pNewMesh->RegisterCallback(CTriMesh::CALLBACK_ATTRIBUTEID, CollectAttributeID, this);
        pNewMesh->LoadMeshFromX( Buffer, D3DXMESH_MANAGED, m_pD3DDevice );
```

Once the mesh is loaded, we weld its vertices and perform an in-place vertex cache optimimzation. This performs the attribute sort as well.

```
        // Attempt to optimize the mesh
        pNewMesh->WeldVertices( 0 );
        pNewMesh->OptimizeInPlace( D3DXMESHOPT_VERTEXCACHE );
```

We call the CScene::AddMesh function to make room for the new mesh in the scene mesh array and add the new mesh pointer to the end of the list.

```
        // Store this new mesh
        if ( AddMesh( ) < 0 ) { delete pNewMesh; return false; }
        m_pMesh[ m_nMeshCount - 1 ] = pNewMesh;

        // Store as object reference mesh
        pReferenceMesh = pNewMesh;

    } // End if mesh doesn't exist.
```

To position the mesh in the scene, we create a new CObject and store the mesh pointer and the entity world matrix inside the newly allocated CObject structure. The object is added to our global list and the function is complete. The CScene::AddObject function is very much like the CScene::AddMesh function in that it resizes the scene's CObject array making room for a new entry at the end.

```
    // Now build an object for this mesh (standard identity)
    CObject * pNewObject = new CObject( pReferenceMesh );
    if ( !pNewObject ) return false;

    // Copy over the specified matrix
    pNewObject->m_mtxWorld = mtxWorld;

    // Store this object
    if ( AddObject() < 0 ) { delete pNewObject; return false; }
    m_pObject[ m_nObjectCount - 1 ] = pNewObject;

    // Success!!
    return true;
}
```

CScene::ProcessSkyBox

ProcessSkyBox (called by CScene::ProcessEntities) adds six cube faces (12 triangles) to the scene skybox mesh and loads the six textures whose names are stored in the skybox entity passed into the function. This function provides some useful insight into populating a CTriMesh manually. It also demonstrates creation of a managed-mode mesh.

When we wish to create a managed-mode mesh from X file data, we would normally register the CollectTexture callback with the mesh so that the mesh loading function can pass the texture filenames found in the X file to the function and get back texture pointers for its own internal storage. The

managed mesh stores the texture pointer along with the matching material for each of its subsets in its internal attribute data array. In this particular case (skybox) we will not be loading the data from an X file, so we will not need to register the callback. We will manually add the texture and material information for each mesh subset to the mesh's internal attribute array ourselves.

The first thing we need to do is call CTriMesh::SetDataFormat to tell the mesh about our desired vertex/index formats. Because we will be adding vertices and indices to the cube mesh one face at a time (i.e. two triangles at a time) each cube face will consist of 4 vertices and 6 indices. We will use two temporary arrays, an index array and a vertex array, to build the cube faces as we go along. The front face case is examined first.

```
bool CScene::ProcessSkyBox( const SkyBoxEntity & SkyBox )
{
    MESH_ATTRIB_DATA * pAttribData;
    USHORT             Indices[6];
    CVertex            Vertices[4];

    // Set the mesh data format
    m_SkyBoxMesh.SetDataFormat( VERTEX_FVF, sizeof(USHORT) );

    // Build Front quad (remember all quads point inward)
    Vertices[0] = CVertex(-10.0f, 10.0f, 10.0f, D3DXVECTOR3(0.0f, 0.0f, 0.0f), 0.0f, 0.0f);
    Vertices[1] = CVertex( 10.0f, 10.0f, 10.0f, D3DXVECTOR3(0.0f, 0.0f, 0.0f), 1.0f, 0.0f);
    Vertices[2] = CVertex( 10.0f,-10.0f, 10.0f, D3DXVECTOR3(0.0f, 0.0f, 0.0f), 1.0f, 1.0f);
    Vertices[3] = CVertex(-10.0f,-10.0f, 10.0f, D3DXVECTOR3(0.0f, 0.0f, 0.0f), 0.0f, 1.0f);

    // Build the skybox indices
    Indices[0] = 0; Indices[1] = 1; Indices[2] = 3;
    Indices[3] = 1; Indices[4] = 2; Indices[5] = 3;
```

To add the indices and vertices for the quads to the mesh we call the AddVertex and AddFace functions. Notice that we are adding 2 triangles for the face and that we pass in an attribute ID of 0. Since each quad will have a different texture mapped to it, each will belong to a different subset. We will incremement this attribute ID for each quad that we add, such that the front face has an attribute ID of 0, the back face has an attribute ID of 1, and so on. The resulting mesh of the cube will contain 12 triangles and six subsets. Each subset represents one cube face and contains two triangles.

```
    // Add the vertices and indices to this mesh
    m_SkyBoxMesh.AddVertex( 4, &Vertices );
    m_SkyBoxMesh.AddFace( 2, Indices, 0 );
```

We repeat the process for the remaining five faces of the cube. Keep in mind when examining the vertex data that these cube faces are inward facing, so the vertex winding order will reflect this fact. Note as well that the zero length normal may seem strange, but the skybox will be rendered with lighting disabled anyway so the normal information will not be used. This allows us to use the same vertex flags for the skybox as the rest of the objects in our scene. Thus, we avoid calling IDirect3DDevice9::SetFVF in the scene's main render function to change vertex formats every time we render the skybox. Alternatively, you can use a vertex format with position and UV coordinates only if you prefer.

```
    // Back Quad
    Vertices[0] = CVertex( 10.0f, 10.0f,-10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 0.0f, 0.0f );
    Vertices[1] = CVertex(-10.0f, 10.0f,-10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 1.0f, 0.0f );
    Vertices[2] = CVertex(-10.0f,-10.0f,-10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 1.0f, 1.0f );
    Vertices[3] = CVertex( 10.0f,-10.0f,-10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 0.0f, 1.0f );

    // Build the skybox indices
    Indices[0] = 4; Indices[1] = 5; Indices[2] = 7;
    Indices[3] = 5; Indices[4] = 6; Indices[5] = 7;

    // Add the vertices and indices to this mesh
    m_SkyBoxMesh.AddVertex( 4, &Vertices );
    m_SkyBoxMesh.AddFace( 2, Indices, 1 );

    // Left Quad
    Vertices[0] = CVertex(-10.0f, 10.0f,-10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 0.0f, 0.0f );
    Vertices[1] = CVertex(-10.0f, 10.0f, 10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 1.0f, 0.0f );
    Vertices[2] = CVertex(-10.0f,-10.0f, 10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 1.0f, 1.0f );
    Vertices[3] = CVertex(-10.0f,-10.0f,-10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 0.0f, 1.0f );

    // Build the skybox indices
    Indices[0] = 8; Indices[1] = 9; Indices[2] = 11;
    Indices[3] = 9; Indices[4] = 10; Indices[5] = 11;

    // Add the vertices and indices to this mesh
    m_SkyBoxMesh.AddVertex( 4, &Vertices );
    m_SkyBoxMesh.AddFace( 2, Indices, 2 );

    // Right Quad
    Vertices[0] = CVertex( 10.0f, 10.0f, 10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 0.0f, 0.0f );
    Vertices[1] = CVertex( 10.0f, 10.0f,-10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 1.0f, 0.0f );
    Vertices[2] = CVertex( 10.0f,-10.0f,-10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 1.0f, 1.0f );
    Vertices[3] = CVertex( 10.0f,-10.0f, 10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 0.0f, 1.0f );

    // Build the skybox indices
    Indices[0] = 12; Indices[1] = 13; Indices[2] = 15;
    Indices[3] = 13; Indices[4] = 14; Indices[5] = 15;

    // Add the vertices and indices to this mesh
    m_SkyBoxMesh.AddVertex( 4, &Vertices );
    m_SkyBoxMesh.AddFace( 2, Indices, 3 );

    // Top Quad
    Vertices[0] = CVertex(-10.0f, 10.0f,-10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 0.0f, 0.0f );
    Vertices[1] = CVertex( 10.0f, 10.0f -10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 1.0f, 0.0f );
    Vertices[2] = CVertex( 10.0f, 10.0f, 10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 1.0f, 1.0f );
    Vertices[3] = CVertex(-10.0f, 10.0f, 10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 0.0f, 1.0f );

    // Build the skybox indices
    Indices[0] = 16; Indices[1] = 17; Indices[2] = 19;
    Indices[3] = 17; Indices[4] = 18; Indices[5] = 19;

    // Add the vertices and indices to this mesh
    m_SkyBoxMesh.AddVertex( 4, &Vertices );
    m_SkyBoxMesh.AddFace( 2, Indices, 4 );
    // Bottom Quad
    Vertices[0] = CVertex(-10.0f,-10.0f, 10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 0.0f, 0.0f );
    Vertices[1] = CVertex( 10.0f,-10.0f, 10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 1.0f, 0.0f );
    Vertices[2] = CVertex( 10.0f,-10.0f,-10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 1.0f, 1.0f );
```

```
    Vertices[3] = CVertex(-10.0f,-10.0f,-10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 0.0f, 1.0f );

    // Build the skybox indices
    Indices[0] = 20; Indices[1] = 21; Indices[2] = 23;
    Indices[3] = 21; Indices[4] = 22; Indices[5] = 23;

    // Add the vertices and indices to this mesh
    m_SkyBoxMesh.AddVertex( 4, &Vertices );
    m_SkyBoxMesh.AddFace( 2, Indices, 5 );
```

At this point the skybox mesh contains the six quads (12 triangles) needed and has an attribute buffer with values ranging from 0 to 5. Thus we have six subsets, with two triangles per subset.

To make the mesh self-contained, we populate its internal attribute data array with the appropriate texture and material for each subset. We allocate space in the mesh MESH_ATTRIB_DATA array for 6 elements, one for each subset, by calling AddAttributeData. We follow this with a call to GetAttributeData to return a MESH_ATTRIB_DATA pointer to the first element in the array. We will then populate the array with the appropriate subset rendering resources. Note that the skybox will use the same default white material for each subset. CScene::CollectTexture is used to load and/or retrieve the texture pointer based on the filenames stored in the skybox entity. Thus, these skybox textures will exist in the scene's main texture array like all the others.

```
    // Add the attribute data (We'll let the skybox manage itself)
    m_SkyBoxMesh.AddAttributeData( 6 );
    pAttribData = m_SkyBoxMesh.GetAttributeData();

    D3DMATERIAL9 Material;
    ZeroMemory( &Material, sizeof(D3DMATERIAL9));
    Material.Diffuse = D3DXCOLOR( 1.0, 1.0, 1.0, 1.0f );
    Material.Ambient = D3DXCOLOR( 1.0, 1.0, 1.0, 1.0f );

    // Build and set the attribute data for the skybox
    for ( ULONG i = 0; i < 6; ++i )
    {
        pAttribData[i].Texture  = CollectTexture( this, SkyBox.Textures[i] );
        pAttribData[i].Material = Material;
        if ( pAttribData[i].Texture ) pAttribData[i].Texture->AddRef();
    } // Next Texture
```

We also remember to call AddRef for each texture interface since we are making a copy of the pointer.

At this point the mesh contains all subset attribute information as well as the vertex and index data (in its temporary system memory vertex and index arrays). The underlying skybox ID3DXMesh has not yet been created, so this is our final step before returning to the caller.

```
    // Build the mesh
    m_SkyBoxMesh.BuildMesh( D3DXMESH_MANAGED, m_pD3DDevice );

    return true;
}
```

We have now discussed all functions of significance with respect to loading the IWF file and populating the scene's mesh, object and resource arrays. As it stands, the scene has everything it needs to render. Let us now look at how the CScene::Render function (called by CGameApp::FrameAdvance) renders all of its meshes.

CScene::Render

The CScene::Render function is responsible for rendering all the meshes which comprise the scene. In this application, all meshes, with the exception of one, are being used in non-managed mode. So this function has to be responsible for setting the states before rendering the subsets of each mesh. The exception is the scene's skybox mesh which is utilizing managed mode. The CScene::RenderSkyBox function is called to set the skybox's position in the world and render it. We will study the RenderSkyBox function shortly.

Depending on the type of scene being rendered, it may be more or less efficient to batch render based on either attribute or transform. We dicussed the various pros and cons of each approach earlier. To allow our code to easily be adjusted to use both rendering strategies, a pre-compiler define is used to control how this function is compiled. Using this technique, we can actually instruct the compiler, as our code is processed, to include or exclude certain sections of the code from the final build.

If we define DRAW_ATTRIBUTE_ORDER with a value of 1, the code that batch renders the scene based on attributes will be compiled. This will render the entire scene based on subsets minimizing texture and material state changes. This would be a suitable strategy if many of your meshes exist in world space and do not need to be transformed.

If we define DRAW_ATTRIBUTE_ORDER with a value of 0, then an alternative version of the render code will be compiled. This time the code will batch render on a per object basis, minimizing transform state changes.

In this demonstration we set DRAW_ATTRIBUTE_ORDER to 0 by default so that it renders on a per object basis by default. Feel free to change this value to 1 so that you can benchmark the different rendering strategies with different scenes.

The code will be discussed a section at a time. This function is longer than would have been the case had a single rendering strategy been used, but it is still pretty straightforward. Which of the two versions of the rendering code actually gets compiled is controlled by the directive just mentioned.

```
void CScene::Render( CCamera & Camera )
{
    ULONG i, j;
    long  MaterialIndex, TextureIndex;

    if ( !m_pD3DDevice ) return;

    // Setup some states...
```

```
    m_pD3DDevice->SetRenderState( D3DRS_NORMALIZENORMALS, TRUE );

    // Render the skybox first !
    RenderSkyBox( Camera );
```

The first thing the code does is set the D3DRS_NORMALIZENORMALS render state to true. This render state forces the pipeline to ensure that vertex normals are unit length before being used for the lighting calculations in the DirectX pipeline. While the normals of a mesh would almost certainly be unit length to begin with, using a world matrix that scales vertices can cause the normals of the vertex to be scaled when the vertex is transformed by the pipeline. If such a scaling matrix is being used, the vertex normals will no longer be unit length, resulting in incorrect lighting calculations. This render state addresses that potential problem.

The first thing we render is the skybox mesh by issuing a call to CScene::RenderSkyBox. The skybox must be rendered first so that its faces are rendered behind all other scene objects (making the skybox scenery appear distant).

The next section of code activates the lights used by the scene. Placing this code here makes sure that if the user loads a new scene with more (or fewer) lights than are currently active, they will automatically be set the next time the scene is rendered. This also makes sure that if the device is reset, the lights are also automatically reset. This code is placed here for simplicity and would be moved outside the main render loop in a real world situation (light states should not be needlessly set every time a frame is rendered).

```
    // Enable lights
    for ( i = 0; i < m_nLightCount; ++i )
    {
        m_pD3DDevice->SetLight( i, &m_pLightList[i] );
        m_pD3DDevice->LightEnable( i, TRUE );
    }
```

Next we define DRAW_ATTRIBUTE_ORDER to 0 so that the scene is rendered per object instead of per subset. By changing this value to 1, you can recompile the code and force the alternative rendering strategy to be used instead. It should be noted that both rendering strategies are mutually exclusive. That is, only one of the possible two sections of rendering code can be compiled into the application at any given time.

```
// For this demo
#define DRAW_ATTRIBUTE_ORDER 0
```

If we have not set DRAW_ATTRIBUTE_ORDER to 0 then it means we wish to compile the code that batch renders the scene across mesh boundries. This is done by looping through every attribute in the scene global list and rendering any meshes that contain a subset with a matching global attribute ID. This minimimzes texture changes but increases the number of times we must set the world matrix. As we are looping through the scene attributes in the outer loop and objects in the inner loop, the world matrix for a particular mesh must be set many times, once for each subset.

The code loops through each of the scene attributes and extracts from the attribute array the texture index and the material index used by that attribute. It uses the material index to get a pointer to the correct material in the scene's material array and sets it as the current material on the device. If the current scene attribute does not contain a material, then we set a default material. We also use the retrieved texture index to bind the correct texture to texture stage 0. If the current scene attribute contains no texture, we set texture stage 0 to NULL.

```
#if ( DRAW_ATTRIBUTE_ORDER != 0 )

    // Loop through each scene owned attribute
    for ( j = 0; j < m_nAttribCount; j++ )
    {
        // Retrieve indices
        MaterialIndex = m_pAttribCombo[j].MaterialIndex;
        TextureIndex  = m_pAttribCombo[j].TextureIndex;

        // Set the states
        if ( MaterialIndex >= 0 )
            m_pD3DDevice->SetMaterial( &m_pMaterialList[ MaterialIndex ] );
        else
            m_pD3DDevice->SetMaterial( &m_DefaultMaterial );

        if ( TextureIndex >= 0 )
            m_pD3DDevice->SetTexture( 0, m_pTextureList[ TextureIndex ]->Texture );
        else
            m_pD3DDevice->SetTexture( 0, NULL );
```

Now that we have the texture and the material set for this attribute, we will loop through every object in the scene, get a pointer to its CTriMesh object, set its world matrix and FVF flags, and draw the current subset. Of course, the subset we are currently processing may not exist in the mesh and it will result in a no-op and quickly return from the function.

```
        // Process each object
        for ( i = 0; i < m_nObjectCount; ++i )
        {
            CD3DXMesh * pMesh = m_pObject[i]->m_pMesh;
            if ( !pMesh ) continue;

            // Setup the per-mesh / object details
            m_pD3DDevice->SetTransform( D3DTS_WORLD, &m_pObject[i]->m_mtxWorld );
            m_pD3DDevice->SetFVF( pMesh->GetFVF() );

            // Render all faces with this attribute ID
            pMesh->DrawSubset( j );

        } // Next Object

    } // Next Attribute
```

The above code repeats for every scene attribute until all subsets of all non-managed meshes have been rendered.

If DRAW_ATTRIBUTE_ORDER is set to 0, alterative rendering code will be compiled instead. This code will render on a per object basis (which proved to be much faster in our example application). This is because it minimizes potential FVF changes and the number of times we have to set the world matrix, thus reducing stalls in the pipeline.

This approach (really just a re-ordering of the code shown above), loops through each of the scene objects and sets its FVF flags and associated world matrix. The next step loops through each of the scenes attributes, setting the material and texture for that attribute. Finally, all meshes are instructed to render any subsets which have matching subset IDs.

```
#else

    // Process each object
    for ( i = 0; i < m_nObjectCount; ++i )
    {
        CD3DXMesh * pMesh = m_pObject[i]->m_pMesh;
        if ( !pMesh ) continue;

        // Setup the per-mesh / object details
        m_pD3DDevice->SetTransform( D3DTS_WORLD, &m_pObject[i]->m_mtxWorld );
        m_pD3DDevice->SetFVF( pMesh->GetFVF() );

        // Loop through each scene owned attribute
        for ( j = 0; j < m_nAttribCount; j++ )
        {
            // Retrieve indices
            MaterialIndex = m_pAttribCombo[j].MaterialIndex;
            TextureIndex  = m_pAttribCombo[j].TextureIndex;

            // Set the states
            if ( MaterialIndex >= 0 )
                m_pD3DDevice->SetMaterial( &m_pMaterialList[ MaterialIndex ] );
            else
                m_pD3DDevice->SetMaterial( &m_DefaultMaterial );

            if ( TextureIndex >= 0 )
                m_pD3DDevice->SetTexture( 0, m_pTextureList[ TextureIndex ]->Texture );
            else
                m_pD3DDevice->SetTexture( 0, NULL );

            // Render all faces with this attribute ID
            pMesh->DrawSubset( j );

        } // Next Object

    } // Next Attribute

#endif // !DRAW_ATTRIBUTE_ORDER != 0
```

Before exiting, we disable any currently active lights so that if we load another IWF scene which uses fewer lights than the current scene, the current scene's lights do not remain active and influence the new scene. Of course, this could be done outside the render loop, and certainly would be in a real world situation. We placed it here for simplicity in our simple application.

```
    // Disable lights again (to prevent problems later)
    for ( i = 0; i < m_nLightCount; ++i ) m_pD3DDevice->LightEnable( i, FALSE );

}
```

CScene::RenderSkyBox

The CScene::RenderSkyBox function builds a world matrix for the skybox that will translate it to the camera's current position. This ensures that the camera remains at the center of the box at all times. We simply place the camera position in the fourth row (the translation vector) of the world matrix and render the cube.

```
void CScene::RenderSkyBox( CCamera & Camera )
{
    // Bail if there is no sky box
    if ( m_SkyBoxMesh.GetNumFaces() == 0 ) return;

    D3DXMATRIX mtxWorld, mtxIdentity;
    D3DXMatrixIdentity( &mtxWorld );
    D3DXMatrixIdentity( &mtxIdentity );

    // Generate our sky box rendering origin and set as world matrix
    D3DXVECTOR3 CamPos = Camera.GetPosition();
    D3DXMatrixTranslation( &mtxWorld, CamPos.x, CamPos.y + 1.3f, CamPos.z );
    m_pD3DDevice->SetTransform( D3DTS_WORLD, &mtxWorld );
```

We disable lighting and depth buffering since neither is appropriate for this type of effect. Lighting would create visible seams and strange coloring. Depth buffering is both unnecessary and undesirable -- it is unnecessary because we do not want to perform thousands of per-pixel depth tests when we know that the depth buffer is currently clear, and it is undesirable because we do not want to write depth values that may potentially occlude other scene objects (the skybox is only a background and should never occlude anything).

```
    // Set up rendering states for the sky box
    m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, FALSE );
    m_pD3DDevice->SetRenderState( D3DRS_ZENABLE, D3DZB_FALSE );
```

When rendering a skybox, we set the texture addressing modes for both the U and V axes to D3DTADDRESS_CLAMP. This prevents pixels on the edge of each quad from being bilinearly filtered with pixels on the opposite side of the texture, creating a visible seam.

```
    m_pD3DDevice->SetSamplerState( 0, D3DSAMP_ADDRESSU, D3DTADDRESS_CLAMP );
    m_pD3DDevice->SetSamplerState( 0, D3DSAMP_ADDRESSV, D3DTADDRESS_CLAMP );
```

Since the skybox is a managed mesh, it will handle setting all texture and render states. It contains the textures used by each of its faces in its internal attribute array which we populated ealier. We simply need to call the self-contained CTriMesh::Draw function.

```
    // Render the sky box
    m_SkyBoxMesh.Draw( );
```

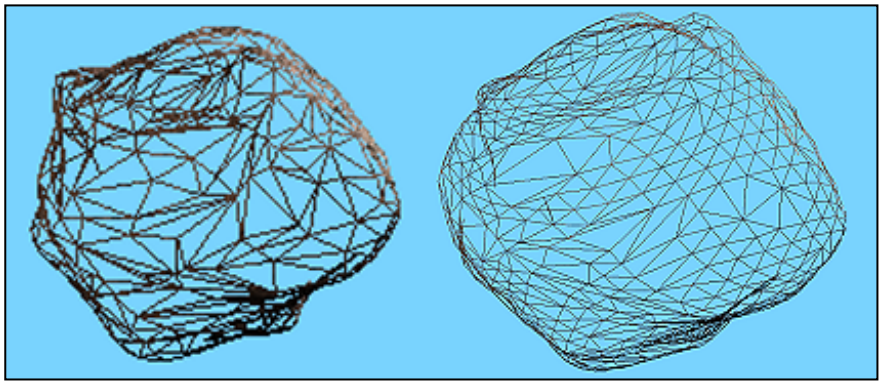
Finally, we reset all render and sampler states and the device world matrix before we return so that we do not influence how the rest of the scene is rendered.

```
    // Reset our states
    m_pD3DDevice->SetSamplerState( 0, D3DSAMP_ADDRESSU, D3DTADDRESS_WRAP );
    m_pD3DDevice->SetSamplerState( 0, D3DSAMP_ADDRESSV, D3DTADDRESS_WRAP );
    m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_ZWRITEENABLE, TRUE );
    m_pD3DDevice->SetTransform( D3DTS_WORLD, &mtxIdentity );
}
```

Lab Project 8.2: Progressive Meshes

This lab project will add Progressive Mesh support to the code we studied in Lab Project 8.1.

To accomplish our objective, we will slightly modify our CTriMesh class so that it has the ability to be used as a static or progressive mesh. Most of the functionality for the mesh class (and indeed the entire application) will be similar to the code studied in the last project.



The CTriMesh (see CObject.h) object will now store two pointers: an ID3DXMesh pointer and an ID3DXPMesh pointer. If the m_pPMesh pointer is active then the progressive mesh will be used by all drawing and optimization functions. If not, the standard mesh will be used. This allows us to easily use the same mesh for both regular and progressive meshes and change from one to the other easily within a CTriMesh object.

```
class CTriMesh
{
  …

  LPD3DXMESH          m_pMesh;                        // Physical mesh object
  LPD3DXPMESH         m_pPMesh;                       // Physical PMesh object
```

We will add a few wrapper functions to set the current level of detail for the underlying progressive mesh. CTriMesh::SetNumVertices and CTriMesh::SetNumFaces simply pass on the request to the underlying ID3DXPMesh if it exists. They do nothing if the progressive mesh has not been generated.

```
  HRESULT           SetNumFaces        ( ULONG FaceCount );
  HRESULT           SetNumVertices     ( ULONG VertexCount );
```

We also include two more wrapper functions for trimming the base level of detail of the underlying progressve mesh. Once again, these functions do nothing if the progressive mesh has not been generated.

```
  HRESULT           TrimByFaces        ( ULONG NewFacesMin, ULONG NewFacesMax );
  HRESULT           TrimByVertices     ( ULONG NewVerticesMin, ULONG NewVerticesMax );
```

CTriMesh also provides a function to generate its progressive mesh using its underlying ID3DXMesh as the input for progressive mesh generation. This call essentially places the CTriMesh object into progressive mesh mode. The underlying ID3DXMesh interface must have been created before switching

to progressive mode. This is because the regular ID3DXMesh is used as the input mesh for D3DX progressive mesh generation.

```
 HRESULT           GeneratePMesh( CONST LPD3DXATTRIBUTEWEIGHTS pAttributeWeights,
                                  CONST FLOAT *pVertexWeights, ULONG MinValue,
                                  ULONG Options, bool ReleaseOriginal = true );
```

Cloning and optimizing are also supported for both standard and progressive meshes.

```
HRESULT           CloneMeshFVF ( ULONG Options, ULONG FVF, CTriMesh * pMeshOut,
                                  MESH_TYPE MeshType = MESH_DEFAULT,
                                  LPDIRECT3DDEVICE9 pD3DDevice = NULL );

HRESULT           Optimize     ( ULONG Flags, CTriMesh * pMeshOut,
                                  MESH_TYPE MeshType = MESH_DEFAULT,
                                  LPD3DXBUFFER * ppFaceRemap = NULL,
                                  LPD3DXBUFFER * ppVertexRemap = NULL,
                                  LPDIRECT3DDEVICE9 pD3DDevice = NULL );
```

Notice that we use a new parameter in both function calls. This is a member of the MESH_TYPE enumerated type. By default, this parameter is set to MESH_DEFAULT, which means that the output mesh will be the same type as the source mesh being cloned. We can also specify standard or progressive flags to convert between one mesh type and the other. This allows us, for example, to take a CTriMesh in progressive mesh mode and clone from another CTriMesh object in regular mesh mode and vice versa.

When cloning a CTriMesh in regular mesh mode out to a CTriMesh in progressive mesh mode, the underlying progressive mesh in the cloned object will use the regular mesh in the source object as its base (highest level of detail) geometry. When cloning a CTriMesh in progressive mesh mode to a CTriMesh in regular mesh mode, the underlying ID3DXMesh in the cloned object will contain the geometry taken from the progressive mesh of the source object at its *current* level of detail. This allows us to create a progressive CTriMesh object and simplify the data by altering its current detail setting, before cloning it back out to a new CTriMesh in regular mesh mode. The source mesh can then be released, leaving us with a simplified standard CTriMesh that can be rendered without the runtime overhead a progressive mesh would incur. If we need the ability to alter the geometric detail of a CTriMesh at runtime, we will want to use a CTriMesh in progressive mesh mode for rendering.

MESH_TYPE is now part of the CTriMesh namespace and is defined in CObject.h as:

```
enum MESH_TYPE     { MESH_DEFAULT = 0, MESH_STANDARD = 1, MESH_PROGRESSIVE = 2 };
```

Finally, CTriMesh now includes some utility wrapper functions for working with the progressive mesh and extracting its current settings. These functions should be self explanatory given our discussion of ID3DXPMesh in the textbook.

```
ULONG             GetMaxFaces       ( ) const;
ULONG             GetMaxVertices    ( ) const;
```

```
ULONG               GetMinFaces       ( ) const;
ULONG               GetMinVertices    ( ) const;
```

If the CTriMesh is not in progressive mesh mode, then all of these functions will return the number of vertices/faces in the regular ID3DXMesh. The existence of an underlying ID3DXPMesh interface takes precedence in such functions, in which case the functions return information about the progressive mesh.

Our application can get a pointer to the underlying ID3DXPMesh interface by calling the GetPMesh function. If the progressive mesh has not been generated, this call will return NULL.

```
LPD3DXPMESH         GetPMesh          ( ) const;
```

While the next two functions are not new to the CTriMesh class, they now behave differently when the mesh object is in progressive mode. In standard mode, these functions return the number of vertices and faces in the temporary arrays used to manually generate the ID3DXMesh when the CTriMesh::BuildMesh function is called. If the ID3DXMesh has already been built, then they return the number of vertices and faces in the underlying D3DXMesh object. But if the mesh object has its progressive mesh generated, then these functions will return the number of vertices and faces used to render the progressive mesh at its current level of detail.

```
ULONG               GetNumVertices    ( ) const;
ULONG               GetNumFaces       ( ) const;
```

The final new function we will add is called SnapshotToMesh. It clones the CTriMesh progressive mesh out to a standard mesh and stores the result in the CTriMesh ID3DXMesh pointer. This call does not destroy the current progressive mesh or change the mode of the CTriMesh object from progressive mesh to standard mesh mode unless the ReleaseProgressive Booelan parameter is set to true. This is handy because it allows us to create a CTriMesh object, generate a progressive mesh for it, lower its level of detail, and clone the current LOD progressive mesh to the underlying standard one. We can then release the progressive mesh, to place the same CTriMesh back into standard mode. We have simply used the progressive mesh in this case to perform a one-time simplification procedure on the mesh data.

```
HRESULT  SnapshotToMesh ( bool ReleaseProgressive = true );
```

If we specify false as the input parameter then the underlying progressive mesh will not be released and the CTriMesh remains in progressive mode. This is also useful since we can attach the simplified mesh to a new CTriMesh if desired. This allows for easy cloning of a progressive mode CTriMesh to a standard CTriMesh. In fact, we use this mechanism in CloneMeshFVF and Optimize to do exactly that.

Before we start examining the underlying code for these new and modified functions in more detail, let us briefly look at some usage scenarios for the updated CTriMesh.

Using CTriMesh to Create a Progressive Mesh

Creating a progressive CTriMesh is a two step process. Step one is exactly the same as our mesh creation code in the last project -- we simply build the mesh so that the underlying ID3DXMesh is created. Step two involves calling CTriMesh::GeneratePMesh to generate the ID3DXPMesh from the underlying ID3DXMesh. This places the CTriMesh in progressive mesh mode and will (by default) release the ID3DXMesh interface. The newly generated ID3DXPMesh will now be used for all rendering. The following example code shows how we might create a progressive CTriMesh using data loaded in from an X file stored on disk:

```
CTriMesh * pNewMesh = new CTriMesh;
pNewMesh->RegisterCallback( CTriMesh::CALLBACK_ATTRIBUTEID, CollectAttributeID, this );
pNewMesh->LoadMeshFromX( 'MyXFile.x', D3DXMESH_MANAGED, m_pD3DDevice );
pNewMesh->WeldVertices( 0 );
pNewMesh->OptimizeInPlace( D3DXMESHOPT_VERTEXCACHE );
```

At this point we have an optimized ID3DXMesh. So far, this has been exactly like the code in Lab Project 8.1. Now we call CTriMesh::GeneratePMesh to place the CTriMesh object into progressive mesh mode. We will pass in NULL in this example for the first two parameters so the default vertex component weighting is used (all vertices are assumed to have the same weight, and therefore will all be assigned the same priority for collapse). We also specify that the underlying progressive mesh should be generated such that it stores enough edge collapse structures to allow us to dynamically simplify down to a face count of 50. This target might not be possible, but the progressive mesh will calculate enough collapse structures to get as close to this number as possible without harming the topology of the mesh to an unacceptable degree.

```
pNewMesh->GeneratePMesh( NULL, NULL, 50, D3DXMESHSIMP_FACE );
```

At this point, the CTriMesh has released its ID3DXMesh interface and now has its m_pPMesh pointer pointing to a valid ID3DXPMesh interface. Finally, we optimize the progressive mesh for rendering.

```
pNewMesh->OptimizeInPlace( D3DXMESHOPT_VERTEXCACHE );
```

CTriMesh::GeneratePMesh

This function will use the standard mesh (CTriMesh::m_pMesh) as the base level of detail for a new progressive mesh. Therefore, the standard mesh should already be a valid ID3DXMesh when this function is called.

```
HRESULT CTriMesh::GeneratePMesh( CONST LPD3DXATTRIBUTEWEIGHTS pAttributeWeights,
                                 CONST FLOAT *pVertexWeights, ULONG MinValue,
                                 ULONG Options, bool ReleaseOriginal /* = true */ )
{
    HRESULT         hRet;
    LPD3DXMESH      pTempMesh      = NULL;
    LPD3DXBUFFER    pTempAdjacency = NULL;

    // Validate Parameters
```

```
    if (!m_pMesh) return D3DERR_INVALIDCALL;
```

The first thing the function does is ensure that m_pMesh is not NULL. If it is, then the function has been called prematurely and we return immediately. If the standard mesh has been created, then we release the current progressive mesh to make room for the new one we are about to create.

```
    // Release previous Progressive mesh if one already exists
    if ( m_pPMesh ) { m_pPMesh->Release(); m_pPMesh = NULL; }
```

To create a progressive mesh, we need adjacency information for the D3DXGeneratePMesh function call. If we have not created the adjacency array previously, we do so now.

```
    // Generate adjacency if not yet available
    if (!m_pAdjacency)
    {
        GenerateAdjacency();
    }
```

Next we need to validate the standard mesh to make sure it does not contain invalid data that would cause the generation of a progressive mesh to fail. We call D3DXValidMesh and pass in a pointer to the standard mesh along with its adjacency data.

```
    // Validate the base mesh
    hRet = D3DXValidMesh( m_pMesh, (DWORD*)m_pAdjacency->GetBufferPointer(), NULL );
```

If the D3DXValidMesh function fails, then the standard mesh contains invalid geometry. We will then attempt to repair the damage using the D3DXCleanMesh function. D3DXCleanMesh does not alter the actual geometry of the mesh being cleaned -- instead it clones the clean data out to a new ID3DXMesh. It also fills a new adjacency buffer, so we must allocate this buffer and pass its pointer into the function.

```
   if ( FAILED(hRet) )
   {
        // Allocate the temporary adjacency buffer
        hRet = D3DXCreateBuffer( m_pAdjacency->GetBufferSize(), &pTempAdjacency );
        if ( FAILED(hRet) ) {  return hRet; }

        // Clean the mesh data storing the new mesh interface pointer in pTempMesh.
        hRet = D3DXCleanMesh( m_pMesh, (DWORD*)m_pAdjacency->GetBufferPointer(), &pTempMesh,
                              (DWORD*)pTempAdjacency->GetBufferPointer(), NULL );

        if ( FAILED(hRet) ) { pTempAdjacency->Release(); return hRet; }
    }
```

The clean mesh is assigned to the local ID3DXMesh pointer pTempMesh. If the function fails, then there is nothing we can do; the data is simply incompatible with progressive mesh generation. If this is the case, we release the temporary adjacency buffer and return.

If the function is successful, pTempMesh will point to the cleaned ID3DXMesh interface pointer and pTempAdjacency will contain the updated face adjacency information.

If m_pMesh was valid to begin with, then we copy the mesh interface pointer and its adjacency buffer into the pTempMesh and pTempAdjacency pointers. This way, whether the mesh needed to be cleaned or not, pTempAdjacency and pTempMesh point to the adjacency buffer and the mesh we will use to create the progressive mesh.

```
    else
    {
        // Simply store common pointers
        pTempAdjacency = m_pAdjacency;
        pTempMesh      = m_pMesh;

        // Add references so that the originals are not released
        m_pAdjacency->AddRef();
        m_pMesh->AddRef();

    } // End if valid mesh
```

We generate the progressive mesh with a call to D3DXGeneratePMesh, passing in the source mesh and other required parameters. We store the new progressive mesh in the CTriMesh::m_pPMesh member pointer when complete and the CTriMesh object is now in progressive mesh mode.

```
  // Generate the progressive mesh
  hRet = D3DXGeneratePMesh(pTempMesh, (DWORD*)pTempAdjacency->GetBufferPointer(),
                           pAttributeWeights, pVertexWeights, MinValue, Options, &m_pPMesh );
```

We can now release pTempMesh and pTempAdjacency since we no longer need them.

```
    // Release all used objects
    pTempMesh->Release();
    pTempAdjacency->Release();
```

Finally, if the ReleaseOriginal Boolean paremeter is set to true (the default) then we release the standard mesh interface (m_pMesh) as well. From this point forward, any member functions we call will work with the underlying progressive mesh instead. Note that we only release the standard mesh if the progressive mesh was successfully generated.

```
    // Release the original mesh if requested, and PMesh generation was a success
    if ( ReleaseOriginal && SUCCEEDED(hRet) ) { m_pMesh->Release(); m_pMesh = NULL; }

    // Success??
    return hRet;
}
```

CTriMesh::CloneMeshFVF

The CloneMeshFVF function has had a fair bit of code added to it to cope with the dual-use nature of the modified CTriMesh class. The MESH_TYPE parameter allows the caller to specify whether the clone will be a progressive mesh (MESH_PROGRESSIVE), a standard mesh (MESH_STANDARD), or the same type as the source mesh (MESH_DEFAULT).

The results differ based on the source mesh type and the requested destination type. The semantics are detailed below:

Source Mesh: MESH_STANDARD
Destination Mesh: MESH_STANDARD -or- MESH_DEFAULT
In this case the output CTriMesh will be a standard ID3DXMesh. The vertices, faces, and attributes will be copied into the output mesh creating a duplicate CTriMesh. Vertex/index format may change depending on the Options and FVF flags passed into the function.

Source Mesh: MESH_STANDARD
Destination Mesh: MESH_PROGRESSIVE
In this case the output CTriMesh will be an ID3DXPMesh. The source mesh determines the base LOD for the progressive mesh. The source mesh remains in standard mode and the output mesh will be in progressive mesh mode. Vertex/index format may change depending on the Options and FVF flags passed into the function.

Source Mesh: MESH_PROGRESSIVE
Destination Mesh: MESH_PROGRESSIVE -or- MESH_DEFAULT
In this case the output CTriMesh will be an ID3DXPMesh. The source progressive mesh is copied in its entirety and the output mesh will have the same face and vertex count as the source. Vertex/index format may change depending on the Options and FVF flags passed into the function.

Source Mesh: MESH_PROGRESSIVE
Destination Mesh: MESH_STANDARD
In this case the output CTriMesh will be an ID3DXMesh. The clone will contain only geometry that is being used at the current level of detail for the source progressive mesh. This is essentially a snapshot of the source mesh at its current LOD. Vertex/index format may change depending on the Options and FVF flags passed into the function.

The first thing the clone function does is make sure that the mesh about to be cloned has a valid source mesh (standard or progressive). If the progressive mesh exists, then the mesh is assumed to be in progressive mesh mode and the standard mesh will be ignored. The progressive mesh will therefore be used as the source mesh in the cloning operation. We also generate face adjacency for the source mesh if it has not been previously generated because it is needed for the cloning operation. Note that the final parameter to this function is a pointer to an IDirect3DDevice9 interface describing the device we would like the output mesh to belong to. Usually we will want this to be the same as the source mesh that is

about to be cloned. If so, we can set this parameter to NULL and the device retrieved from the current mesh will be used.

```
HRESULT CTriMesh::CloneMeshFVF(ULONG Options, ULONG FVF, CTriMesh * pMeshOut,
                               MESH_TYPE MeshType , LPDIRECT3DDEVICE9 pD3DDevice)
{
    HRESULT         hRet;
    LPD3DXBASEMESH  pCloneMesh  = NULL;

    // Validate requirements
    if ( (!m_pMesh && !m_pPMesh) || !pMeshOut ) return D3DERR_INVALIDCALL;

    // Generate adjacency if not yet available
    if (!m_pAdjacency)
    {
        hRet = GenerateAdjacency();
        if ( FAILED(hRet) ) return hRet;

    } // End if no adjacency

    // If no new device was passed...
    if ( !pD3DDevice )
    {
        // we'll use the same device as this mesh
        // This automatically calls 'AddRef' for the device
        pD3DDevice = GetDevice();
    }
    else
    {
        // Otherwise we'll add a reference here so that we can
        // release later for both cases without doing damage :)
        pD3DDevice->AddRef();
    }
```

The next step is to determine the requested destination mesh type and process the request as described above.

If MESH_DEFAULT was passed in then we create a cloned CTriMesh of the same type as the current mesh object being cloned. Progressive meshes take priority, so if the m_pPMesh pointer is not NULL then the mesh is currently in progressive mesh mode and we will need to clone a new progressive mesh using the D3DXClonePMeshFVF function. If the m_pPMesh pointer is NULL then we assume the mesh is in standard mode and clone the standard mesh using D3DXCloneMeshFVF. Note that we store the cloned interface as an ID3DXBaseMesh. This allows us to use the same pointer type for both types.

```
    switch ( MeshType )
    {
        case MESH_DEFAULT:  // Continue to use the same type as 'this'
            if ( m_pPMesh )
            {
                // Attempt to clone the mesh
                hRet = m_pPMesh->ClonePMeshFVF( Options, FVF, pD3DDevice, \
                                                (LPD3DXPMESH*)&pCloneMesh );
                if ( FAILED(hRet) ) { pD3DDevice->Release(); return hRet; }
```

```
            } // End if Progressive
            else
            {
                // Attempt to clone the mesh
                hRet = m_pMesh->CloneMeshFVF( Options, FVF, pD3DDevice, \
                                              (LPD3DXMESH*)&pCloneMesh );
                if ( FAILED(hRet) ) { pD3DDevice->Release(); return hRet; }

            } // End if standard
            break;
```

Processing the request for a standard output mesh is just as easy. Again, we start by checking our progressive mesh first and then fall back to the standard mesh if necessary. Both cases use the CloneMeshFVF to accomplish the objective.

```
        case MESH_STANDARD: // Convert to, or continue to use standard mesh type
            if ( m_pPMesh )
            {
                // Attempt to clone the mesh
                hRet = m_pPMesh->CloneMeshFVF( Options, FVF, pD3DDevice, \
                                               (LPD3DXMESH*)&pCloneMesh );
                if ( FAILED(hRet) ) { pD3DDevice->Release(); return hRet; }
           }
            else
            {
                // attempt to clone the mesh
                hRet = m_pMesh->CloneMeshFVF( Options, FVF, pD3DDevice, \
                                              (LPD3DXMESH*)&pCloneMesh );
                if ( FAILED(hRet) ) { pD3DDevice->Release(); return hRet; }
            }
            break;
```

The final case handles cloning to a progressive mesh. If the progressive mesh exists in the source object, then this becomes a straight cloning operation. If not, then we must convert the standard mesh into progressive form, directly storing the result in the output mesh object.

```
        case MESH_PROGRESSIVE: // Convert to, or continue to use progressive mesh type
            if ( m_pPMesh )
            {
                // Attempt to clone the mesh
                hRet = m_pPMesh->ClonePMeshFVF( Options, FVF, pD3DDevice, \
                                                (LPD3DXPMESH*)&pCloneMesh );
                if ( FAILED(hRet) ) { pD3DDevice->Release(); return hRet; }
            }
            else
            {
                // Attempt to clone the mesh
                hRet = D3DXGeneratePMesh( m_pMesh, (DWORD*)m_pAdjacency->GetBufferPointer(),
                                NULL, NULL, 1, D3DXMESHSIMP_FACE, (LPD3DXPMESH*)&pCloneMesh );
                if ( FAILED(hRet) ) { pD3DDevice->Release(); return hRet; }
           }
        break;

    } // End type switch
```

We can now attach the new D3DXMesh to the output CTriMesh object that was passed into the function and release the base mesh interface pointer because we no longer need it.

```
    // Attach this D3DX mesh to the output mesh
    // This automatically adds a reference to the mesh passed in.
    pMeshOut->Attach( pCloneMesh );

    // We can now release our copy of the cloned mesh
    pCloneMesh->Release();
```

Our final task is to test whether the source CTriMesh has any data in its attribute data array. If it does then we are cloning from a managed CTriMesh and we should copy over the attribute data to the output mesh so that it will have the required textures and materials for each of its subsets.

```
    // Copy over attributes if there is anything here
    if ( m_pAttribData )
    {
        // Add the correct number of attributes
        if ( pMeshOut->AddAttributeData( m_nAttribCount ) < 0 ) return E_OUTOFMEMORY;

        // Copy over attribute data
        MESH_ATTRIB_DATA * pAttributes = pMeshOut->GetAttributeData();
        for ( ULONG i = 0; i < m_nAttribCount; ++i )
        {
            MESH_ATTRIB_DATA * pAttrib = &pAttributes[i];
            // Store details
            pAttrib->Material = m_pAttribData[i].Material;
            pAttrib->Texture  = m_pAttribData[i].Texture;
            pAttrib->Effect   = m_pAttribData[i].Effect;

            // Add references so that objects aren't released when either of these
            // meshes are released, or vice versa.
            if ( pAttrib->Texture ) pAttrib->Texture->AddRef();
            if ( pAttrib->Effect  ) pAttrib->Effect->AddRef();

        } // Next Attribute

    } // End if managed

    // Release our referenced D3D Device
    if (pD3DDevice) pD3DDevice->Release();

    // Success!!
    return S_OK;
}
```

It should be noted that the CTriMesh::Attach function has been altered so that it accepts an ID3DXBaseMesh pointer. It will determine whether it is a standard or progressive mesh using IUnknown::QueryInterface and assign the pointer to the underlying ID3DXMesh or ID3DXPMesh member pointer of the CTriMesh object appropriately.

CTriMesh::Optimize

The optimize function has had some small changes to accommodate our dual-use CTriMesh concept. Recall that this function works like a combination clone/optimize-in-place operation. The output mesh type is once again specified using the MESH_TYPE input parameter. The caller must pass in a pointer to an already instantiated CTriMesh object that will receive the new cloned and optimized D3DX mesh.

The first part of this function is the same as CTriMesh::CloneMeshFVF. It exits if the standard or progressive meshes are not yet created, it generates adjacency information if it does not yet exist, and it retrieves the device from the source mesh if the caller passes NULL for the device parameter.

```
HRESULT CTriMesh::Optimize( ULONG Flags, CTriMesh * pMeshOut, MESH_TYPE MeshType ,
                            LPD3DXBUFFER * ppFaceRemap , LPD3DXBUFFER * ppVertexRemap ,
                            LPDIRECT3DDEVICE9 pD3DDevice )
{
    HRESULT             hRet;
    LPD3DXMESH          pOptimizeMesh = NULL;
    LPD3DXBUFFER        pFaceRemapBuffer = NULL;
    LPD3DXBUFFER        pAdjacency = NULL;
    ULONG              *pData      = NULL;

    // Validate requirements
    if ( (!m_pMesh && !m_pPMesh) || !pMeshOut ) return D3DERR_INVALIDCALL;

    // Generate adjacency if not yet available
    if (!m_pAdjacency)
    {
        hRet = GenerateAdjacency();
        if ( FAILED(hRet) ) return hRet;

    } // End if no adjacency

    // If no new device was passed...
    if ( !pD3DDevice )
    {
        // we'll use the same device as this mesh
        // This automatically calls 'AddRef' for the device
        pD3DDevice = GetDevice( );

    }
    else
    {
        // Otherwise we'll add a reference here so that we can
        // release later for both cases without doing damage :)
        pD3DDevice->AddRef();

    } // End if new device
```

Next we allocate an ID3DXBuffer which we can pass into the Optimize function and it will be filled with the face adjacency information for the new optimized mesh. Also, if the ppFaceRemap parameter is not NULL, then the caller would like to know how the faces were re-mapped from their original positions. In that case, we need to allocate an ID3DXBuffer object to contain this information.

```
    hRet = D3DXCreateBuffer( (3 * GetNumFaces()) * sizeof(ULONG), &pAdjacency );
    if ( FAILED(hRet) ) { pD3DDevice->Release(); return hRet; }

    // Allocate the output face remap if requested
    if ( ppFaceRemap )
    {
        // Allocate new face remap output buffer
        hRet = D3DXCreateBuffer( GetNumFaces() * sizeof(ULONG), ppFaceRemap );
        if ( FAILED(hRet) ) { pD3DDevice->Release(); pAdjacency->Release(); return hRet; }
        pData = (ULONG*)(*ppFaceRemap)->GetBufferPointer();
    } // End if allocate face remap data
```

We now optimize the CTriMesh into a new object. Priority is once again given to the progressive mesh and we fall back to the standard mesh if the progressive mesh has not been created. The mesh that is output from the Optimize call will match the type of the mesh that called the function.

```
    if ( m_pPMesh )
    {
        // Attempt to optimize the progressive mesh
       m_pPMesh->Optimize( Flags, (ULONG*)pAdjacency->GetBufferPointer(), pData,
                           ppVertexRemap, &pOptimizeMesh );
    }
    else
    {
        // Attempt to optimize the standard mesh
     m_pMesh->Optimize( Flags, (ULONG*)m_pAdjacency->GetBufferPointer(),
                       (ULONG*)pAdjacency->GetBufferPointer(),  pData, ppVertexRemap,
                        &pOptimizeMesh );
    }
```

We attach the new mesh to the output CTriMesh object passed into the function.

```
    // Attach this D3DX mesh to the output mesh
    // This automatically adds a reference to the mesh passed in.
    pMeshOut->Attach( pOptimizeMesh, pAdjacency );
```

Our next task is to convert the new mesh (if necessary) into the format requested by the caller.

```
    switch ( MeshType )
    {
        case MESH_DEFAULT:  // Continue to use the same type as 'this'

            // Already a standard mesh, does it need converting ?
            if ( m_pPMesh ) pMeshOut->GeneratePMesh(NULL, NULL, 0, D3DXMESHSIMP_FACE, true);
            break;

        case MESH_PROGRESSIVE: // Convert to, or continue to use progressive mesh type

            // Already a standard mesh, convert it
            pMeshOut->GeneratePMesh( NULL, NULL, 0, D3DXMESHSIMP_FACE, true );
            break;

    } // End type switch
```

Now that the new mesh is attached to the output mesh, we can release the temporary pOptimizeMesh interface and the temporary adjacency buffer. Our final task is to copy the attribute array (if it exists) to accommodate managed meshes.

```
    // We can now release our copy of the optimized mesh and the adjacency buffer
    pOptimizeMesh->Release();
    pAdjacency->Release();

    // Copy over attributes if there is anything here
    if ( m_pAttribData )
    {
        // Add the correct number of attributes
        if ( pMeshOut->AddAttributeData( m_nAttribCount ) < 0 ) return E_OUTOFMEMORY;

        // Copy over attribute data
        MESH_ATTRIB_DATA * pAttributes = pMeshOut->GetAttributeData();
        for ( ULONG i = 0; i < m_nAttribCount; ++i )
        {
            MESH_ATTRIB_DATA * pAttrib = &pAttributes[i];
            // Store details
            pAttrib->Material = m_pAttribData[i].Material;
            pAttrib->Texture  = m_pAttribData[i].Texture;
            pAttrib->Effect   = m_pAttribData[i].Effect;

            // Add references so that objects aren't released when either of these
            // meshes are released, or vice versa.
            if ( pAttrib->Texture ) pAttrib->Texture->AddRef();
            if ( pAttrib->Effect  ) pAttrib->Effect->AddRef();

        } // Next Attribute

    } // End if managed

    // Release our referenced D3D Device
    if (pD3DDevice) pD3DDevice->Release();

    // Success!!
    return S_OK;
}
```

CTriMesh::OptimizeInPlace

OptimizeInPlace has also been slightly modified to accommodate our new progressive mesh functionality, although to a lesser degree than the Optimize call. The only different between this version of the function and the version from Lab Project 8.1 is that it tests to see if the object is in progressive mesh or standard mesh mode. In standard mesh mode, this function simply wraps a call to the ID3DXMesh::OptimizeInPlace function for its internal ID3DXMesh object. Since this function does not exist in the ID3DXPMesh interface, we use the less flexible D3DX optimization function called OptimizeBaseLOD to optimize the underlying progressive mesh if the CTriMesh object is in progressive mesh mode.

```
HRESULT CTriMesh::OptimizeInPlace( DWORD Flags, LPD3DXBUFFER * ppFaceRemap ,
                                   LPD3DXBUFFER * ppVertexRemap  )
{
    HRESULT      hRet;
    LPD3DXBUFFER pFaceRemapBuffer = NULL;
    ULONG       *pData = NULL;

    // Validate Requirements
    if ( (!m_pMesh && !m_pPMesh) ) return D3DERR_INVALIDCALL;

    // Generate adjacency if none yet provided
    if (!m_pAdjacency)
    {
        hRet = GenerateAdjacency();
        if ( FAILED(hRet) ) return hRet;
    }

    // Allocate the output face remap if requested
    if ( ppFaceRemap )
    {
        // Allocate new face remap output buffer
        hRet = D3DXCreateBuffer( GetNumFaces() * sizeof(ULONG), ppFaceRemap );
        if ( FAILED(hRet) ) { return hRet; }
        pData = (ULONG*)(*ppFaceRemap)->GetBufferPointer();
    }

    // Optimize the data
    if ( m_pPMesh )
        hRet = m_pPMesh->OptimizeBaseLOD( Flags, pData );
    else
        hRet = m_pMesh->OptimizeInplace( Flags, (DWORD*)m_pAdjacency->GetBufferPointer(),
                                        (DWORD*)m_pAdjacency->GetBufferPointer(),
                                         pData, ppVertexRemap );
        if ( FAILED( hRet ) ) return hRet;

    // Success!!
    return S_OK;
}
```

CTriMesh::SetNumFaces/SetNumVertices

These functions simply wrap the ID3DXPMesh calls of the same names and allow us to dynamically alter the current LOD of the underlying progressive mesh. The function does nothing if the CTriMesh object has not had its underlying ID3DXPMesh generated.

```
HRESULT CTriMesh::SetNumFaces( ULONG FaceCount )
{
    HRESULT hRet = D3DERR_INVALIDCALL;

    // Set number of faces (this is a no-op if there is no pmesh)
    if ( m_pPMesh ) hRet = m_pPMesh->SetNumFaces( FaceCount );

    // Success??
    return hRet;
}
```

```
HRESULT CTriMesh::SetNumVertices( ULONG VertexCount )
{
    HRESULT hRet = D3DERR_INVALIDCALL;

    // Set number of vertices (this is a no-op if there is no pmesh)
    if ( m_pPMesh ) hRet = m_pPMesh->SetNumVertices( VertexCount );

    // Success??
    return hRet;
}
```

CTriMesh::TrimByFaces

TrimByFaces is a dual-mode function that works differently depending on whether the CTriMesh is in progressive or standard mesh mode. In progressive mode, this call simply wraps the call to ID3DXPMesh::TrimByFaces. This will set new upper and lower boundaries for the progressive mesh. The caller specifies the new maximum number of faces, which must be less than or equal to the current maximum number of faces, and the new minimum number of faces which must be greater than or equal to the current minimum number of faces. This function allows us to trim the dynamic range of the progressive mesh freeing collapse structures from memory allowing the mesh to consume less memory.

If this function is called and the CTriMesh is in standard mode (no progressive mesh exists) then the function is interpreted as a request to simplify the underlying ID3DXMesh standard mesh down to a new level of detail. When this is the case, we call D3DXSimplifyMesh to reduce the input mesh (the current standard mesh) to the requested LOD. Since the output of this call is a new mesh, the old standard mesh is released and the new simplied ID3DXMesh is assigned as the new standard mesh for the CTriMesh.

```
HRESULT CTriMesh::TrimByFaces( ULONG NewFacesMin, ULONG NewFacesMax )
{
    HRESULT hRet = D3DERR_INVALIDCALL;

    // Trim by faces
    if ( m_pPMesh )
    {
        // Drop through to P mesh trimming
        hRet = m_pPMesh->TrimByFaces( NewFacesMin, NewFacesMax, NULL, NULL );
    }
    else if ( m_pMesh )
    {
        LPD3DXMESH pMeshOut = NULL;

        // Generate adjacency if not yet available
        if (!m_pAdjacency)
        {
             GenerateAdjacency();
        }

        // Since there is no PMesh, we can only comply by simplifying
        D3DXSimplifyMesh(m_pMesh, (DWORD*)m_pAdjacency->GetBufferPointer(), NULL, NULL,
                         NewFacesMax, D3DXMESHSIMP_FACE, &pMeshOut);
```

```
        // Release old mesh and store the new mesh
        m_pMesh->Release();
        m_pMesh = pMeshOut;

        // Adjacency will be out of date, update it
       GenerateAdjacency();
   }

    // Success??
    return hRet;
}
```

Note: There is also a CTriMesh::TrimByVertices function which is exactly the same code as above but trims LOD based on vertices instead of faces. Check the source code for this project for a listing of this function.

CTriMesh::GetNumFaces

All of the GetXX functions of the CTriMesh class have been slightly modified to accommodate our dual-use mesh class. We will examine the GetNumFaces call in this example, but you should be aware that all of these functions behave in exactly the same way.

In progressive mesh mode, GetNumFaces returns the number of faces being used by the progressive mesh at its current LOD. In standard mesh mode, there are two possible cases. The first is when the ID3DXMesh has been created. In this case GetNumFaces will return the number of faces in the underlying ID3DXMesh. The second case handles the mesh as it is undergoing manual construction, before it has been converted into a D3DX mesh type. In this case we return the number of faces in the temporary index array.

```
ULONG CTriMesh::GetNumFaces( ) const
{
    // Validation!!
    if ( m_pPMesh )       return m_pPMesh->GetNumFaces();
    else if ( m_pMesh )   return m_pMesh->GetNumFaces();
    else                  return m_nFaceCount;
}
```

CTriMesh::DrawSubset

The DrawSubset function has hardly changed at all. ID3DXMesh and ID3DXPMesh both support the DrawSubset function, so this function only needs to determine whether the progressive mesh has been generated or if we should render using the standard mesh. We use an ID3DXBaseMesh pointer to render the subset to handle both cases.

```
void CTriMesh::DrawSubset( ULONG AttributeID )
{
    LPDIRECT3DDEVICE9 pD3DDevice = NULL;
```

```
    LPD3DXBASEMESH    pMesh      = m_pMesh;

    // Retrieve mesh pointer
    if ( !pMesh ) pMesh = m_pPMesh;
    if ( !pMesh ) return;

    // Set the attribute data if it exisits (which means it is a managed mesh)
    if ( m_pAttribData && AttributeID < m_nAttribCount )
    {
        pD3DDevice = GetDevice( );
        pD3DDevice->SetMaterial( &m_pAttribData[AttributeID].Material );
        pD3DDevice->SetTexture( 0, m_pAttribData[AttributeID].Texture );
        pD3DDevice->Release();
    }

    // simply render the subset(s)
    pMesh->DrawSubset( AttributeID );
}
```

Additional Code Changes

The rest of the changes to our scene viewer can be found in the CScene class. Most changes are relatively minor and you will find most of the code from Lab Project 8.1 intact.

The first change is the addition of one line of code to the ProcessMeshes and ProcessReference functions. Once these functions have created the CTriMesh and generated the underlying ID3DXMesh, we call the new function CTriMesh::GeneratePMesh to generate the underlying progressive mesh (placing it into progressive mesh mode) before adding it to the scene mesh array. The following snippet from ProcessReference illustrates the idea:

```
…
CTriMesh * pNewMesh = new CTriMesh;

// Load in the externally referenced X File
pNewMesh->RegisterCallback( CTriMesh::CALLBACK_ATTRIBUTEID, CollectAttributeID, this );
pNewMesh->LoadMeshFromX( Buffer, D3DXMESH_MANAGED, m_pD3DDevice );

// Attempt to optimize the standard mesh ready fro progressive mesh generation
pNewMesh->WeldVertices( 0 );
pNewMesh->OptimizeInPlace( D3DXMESHOPT_VERTEXCACHE );

// Convert this mesh to a progressive mesh
pNewMesh->GeneratePMesh( NULL, NULL, 50, D3DXMESHSIMP_FACE );

// Optimize the progressive form of the mesh
pNewMesh->OptimizeInPlace( D3DXMESHOPT_VERTEXCACHE );

// Store this new mesh
if ( AddMesh( ) < 0 ) { delete pNewMesh; return false; }
m_pMesh[ m_nMeshCount - 1 ] = pNewMesh;
…
```

In this demo, all of the meshes in our scene mesh array will be progressive meshes with an arbitrarily set minimum LOD of 50 faces. Note that the skybox will not be a progressive mesh.

CScene::CalculateLODRatio

We have added a few new functions to our CScene class to set the current LOD for our meshes prior to rendering. The LOD selected will be based on the distance from the mesh to the camera. As meshes get further away from the camera, they will use fewer faces and vice versa. This is easiest to see if you switch the application to wireframe render mode.

CalculateLODRatio manages this process. The function returns a value between 0.0 and 1.0 based on camera distance. This value will be used to select a new LOD for the number of mesh faces in the CScene::Render function. The [0.0, 1.0] range serves as a percentage between the minimum and maximum number of faces for the mesh.

```
float CScene::CalculateLODRatio( const CCamera & Camera, const D3DXMATRIX & mtxObject ) const
{
    // Retrieve object position.
    D3DXVECTOR3 vecObject = D3DXVECTOR3( mtxObject._41, mtxObject._42, mtxObject._43 );

    // Calculate rough distance to object
    float Distance = D3DXVec3Length( &(vecObject - Camera.GetPosition()) );

    // Calculate the LOD Ratio, from 0.0 to 1.0 within the first 60% of the frustum.
    float Ratio = 1.0f - (Distance / (Camera.GetFarClip() * 0.6f));
    if ( Ratio < 0.0f ) Ratio = 0.0f;

    // We have our value
    return Ratio;
}
```

We pass in the camera and the world matrix for the object we are about to render. The world space position is extracted from the fourth row of the matrix and the distance to the camera is calculated using the D3DXVec3Length function. We then divide the distance by the distance to the camera far plane and multiply by 0.6. This means that any mesh that is over 60% of the distance from the camera, between the near and far planes, will always have a ratio of 0.0. In this case the mesh will be set to its lowest level of detail.

CScene::Render

The CScene::Render function has not been modified much since Lab Project 8.1 so we will examine only the changes that were made. Three new lines were added to the call (highlighted in the following listing) to manage the LOD transition for each mesh.

For each mesh, we calculate its LOD ratio by passing its object world matrix to CalculateLODRatio. The return value is used to calculate the desired face count for the progressive mesh. The ratio interpolates between the minimum and maximum number of faces for the mesh. We then set the mesh LOD with a call to the CTriMesh::SetNumFaces.

```
    for ( i = 0; i < m_nObjectCount; ++i)
    {
        CTriMesh * pMesh = m_pObject[i]->m_pMesh;

        // Calculate LOD ratio
        float Ratio = CalculateLODRatio( Camera, m_pObject[i]->m_mtxWorld );

        // Calculate the number of faces (simple linear interpolation)
        ULONG FaceCount = pMesh->GetMinFaces() +
                          (ULONG)((float)(pMesh->GetMaxFaces()-pMesh->GetMinFaces()) * Ratio);

        // Set the LOD's number of faces
        pMesh->SetNumFaces( FaceCount );

        // Setup the per-mesh / object details
        m_pD3DDevice->SetTransform( D3DTS_WORLD, &m_pObject[i]->m_mtxWorld );
        m_pD3DDevice->SetFVF( pMesh->GetFVF() );

        // Loop through each scene owned attribute
        for ( j = 0; j < m_nAttribCount; j++ )
        {
            // Retrieve indices
            MaterialIndex = m_pAttribCombo[j].MaterialIndex;
            TextureIndex  = m_pAttribCombo[j].TextureIndex;

            // Set the states
            if ( MaterialIndex >= 0 )
                m_pD3DDevice->SetMaterial( &m_pMaterialList[ MaterialIndex ] );
            else
                m_pD3DDevice->SetMaterial( &m_DefaultMaterial );

            if ( TextureIndex >= 0 )
                m_pD3DDevice->SetTexture( 0, m_pTextureList[ TextureIndex ]->Texture );
            else
                m_pD3DDevice->SetTexture( 0, NULL );

            // Render all faces with this attribute ID
            pMesh->DrawSubset( j );

        } // Next Attribute

        // Increase tri-rendering count
        m_nTrisRendered += pMesh->GetNumFaces();

    } // Next Object
```

Notice that while the progressive mesh provided by D3DX is a view *independent* progressive mesh, we calculate the LOD based on view dependant information. This is a common strategy, but not the only way to do it. You might instead enable a menu option for reducing/increasing LOD. Perhaps certain meshes are considered more important than others and the lower priority meshes can be simplified. The

bottom line is that how you choose to modify LOD is up to you. Here we have implemented a distance based approach simply to show how the CTriMesh in progressive mesh mode can be used by an application.

We now have a CTriMesh class that is much more than a simple wrapper. Indeed, it is more like a complete mesh toolkit handling resource management, standard and progressive meshing techniques, optimization and cloning from standard to progressive meshes and vice versa and finally, automated or scene handled rendering strategies are exposed to the application. This functionality will serve us well in future applications as we move forward in the course.