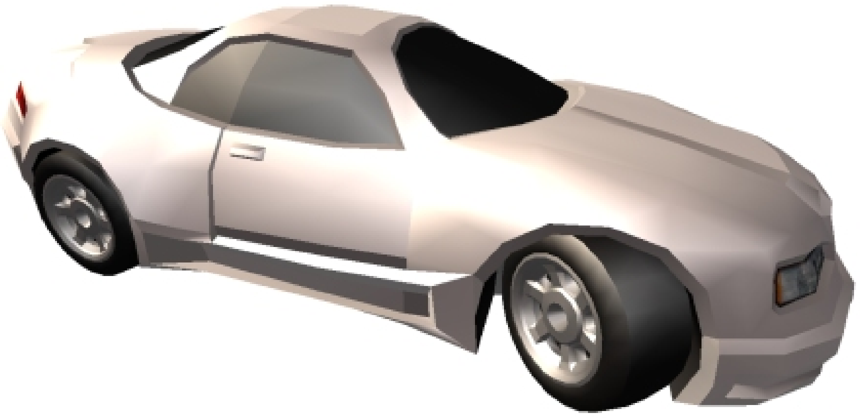
Workbook Nine: Frame Hierarchies



Lab Project 9.1: Multi-Mesh Hierarchies

The focus of this lab project will be on loading and rendering multi-mesh frame hierarchies. Many of the topics discussed in the accompanying textbook will be demonstrated as we progress. For example, we will implement a class called CActor which encapsulates hierarchies and provide the application with an easy means to update and render them. We will also derive our own class from ID3DXAllocateHierarchy to facilitate the automated loading of multi-mesh X files within CActor. In addition, we will explore the relative nature of frame hierarchies by creating a simple animation class (called CAnimation). As discussed in the textbook, objects of this type can be attached to frames in the hierarchy and used to animate/update their matrices in a local fashion. While the demonstration application we create (Figure 9.1) may appear to be a simple mesh viewer, the car is actually a multi-mesh representation. This allows us to apply animations to the wheels separately from the rest of the hierarchy. Finally, the frames in our hierarchy will store pointers to meshes of type CTriMesh. As a result, we can continue to use the mesh resource management system we developed in the previous chapter and render our meshes using the class interface we are now familiar with.

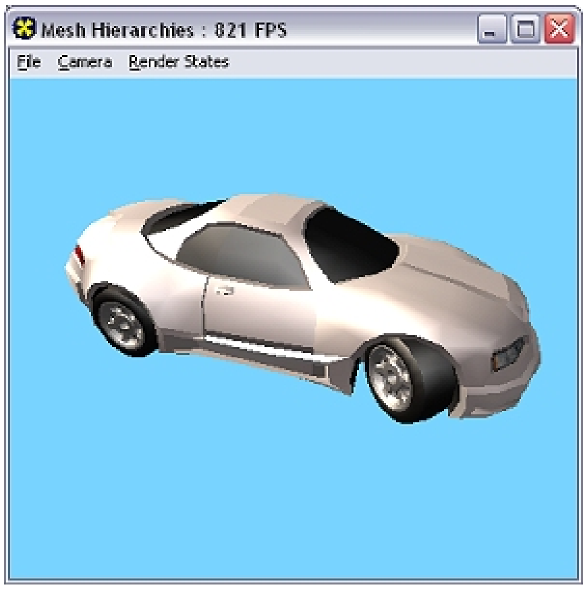


Figure 9.1

This lab project will teach you how to do the following:

- Populate a frame hierarchy from an X file.
- Derive your own class from the ID3DXAllocateHierarchy interface.
- Populate the frame hierarchy from external references inside an IWF file.
- Animate individual frames in the hierarchy.
- Traverse and render a hierarchy.
- Represent meshes in the hierarchy using our own object type (CTriMesh).

Introducing CActor

To encapsulate hierarchies (loaded from an X file in this case), the CActor class will be implemented and employed. If multiple hierarchical X files need to be loaded (e.g., multiple game characters) then a separate CActor object will be instantiated to wrap the frame hierarchy and mesh data for each one. The term 'Actor' is commonly used in many game development SDKs to represent an animation capable object or hierarchy of objects that plays a part in the scene. This is not unlike the way an actor in Hollywood plays a part in a movie. While this is exactly the context in which we are using the name Actor, it should be made clear that our CActor class need not just represent the frame hierarchy for a single multi-mesh model (e.g., a character or an automobile) but may represent an entire scene hierarchy loaded from a single X file.

The CActor class will wrap the call to the D3DXLoadMeshHierarchyFromX function with its own method called LoadActorFromX. This is the function that our application will call to load the X file into the actor. When it returns control back to the application, the hierarchy and mesh data that was contained in the file will be loaded and the CActor will contain a D3DXFRAME derived pointer to the root frame of the hierarchy. The CActor class will also expose functions to traverse the hierarchy and update the frame matrices to generate the absolute matrices for each frame prior to rendering its meshes. Additionally, the interface will expose functions to render the hierarchy and its contained meshes.

In the textbook we learned that in order to use the D3DXLoadMeshHierarchyFromX function, we must derive a class from the ID3DXAllocateHierarchy interface. You will recall that the four member functions of this interface are used as callbacks by the D3DX loading function to allow the application to allocate the memory for the hierarchy frames and mesh containers in the appropriate way. Our ID3DXAllocateHierarchy derived class will be a standalone class that is instantiated and used by the CActor object to pass into the D3DXLoadMeshHierarchyFromX function. The CreateMeshContainer callback function of the CAllocateHierarchy class that we derive will also need access to the CActor object which is currently using it. Therefore, when we derive our class, we will also add a pointer to a CActor object that will be set by its constructor.

The reason the CAllocateHierarchy::CreateMeshContainer function needs to know about the CActor object using it harkens back to the previous lesson where we wanted to have texture and material management for all meshes (contained in all actors) handled at the scene level. You will recall that we implemented a callback mechanism for our CTriMesh class so that when it loaded data from an X file, the returned texture and material buffer was passed off to the scene (via the scene callback functions that had been registered with the mesh) to add the textures and materials to the scene's texture and material arrays. With meshes in non-managed mode, this meant that multiple CTriMeshes in the scene shared the same resources and needed to be rendered by the scene itself. This provided better batching potential. Even in non-managed mode (where the CTriMesh stored its own texture and material information) the textures were still loaded and stored in the scene object to minimize redundancy when multiple managed meshes used the same texture resources. We very much wish to use our CTriMesh class to represent the various meshes stored in our D3DX frame hierarchies so that we can work with an interface that we are comfortable with and continue to benefit from its cloning and optimization functions. Furthermore, we definitely want our meshes to benefit from the managed and non-managed mode resource management systems implemented by that class.

Unlike using our CTriMesh class in isolation, where creation is basically performed by the D3DXLoadMeshFromX function, our meshes will now need to be created manually inside the CreateMeshContainer function of the CAllocator object. Recall that this function is called for each mesh in the hierarchy by D3DX and it is passed the ID3DXMesh interface and the materials the mesh uses. Therefore, we do not wish to use the CTriMesh::LoadMeshFromX function (as we did in the previous lesson) to create our CTriMesh data because the mesh data has already been loaded by D3DX. Instead, our CTriMesh instances will need to be created inside the CreateMeshContainer function and attach the ID3DXMesh data that we are passed. This presents a small problem because previously our application would instantiate a CTriMesh, register the material callbacks, and then call the CTriMesh::LoadMeshFromX function which triggered these callbacks. But now that the CTriMesh::LoadMeshFromX function will no longer be used, our CTriMesh is not created until the hierarchy is being loaded. Therefore, the application cannot register callbacks with meshes that do not yet exist prior to the D3DXLoadMeshHierarchyFromX call made by CActor. The simple solution is to duplicate some of this callback functionality inside CActor. This will allow the application to register the same three callback functions with the CActor instead of CTriMesh. When the CreateMeshContainer function is called, because the CAllocateHierarchy object has a pointer stored for the CActor for which it is being used, it can access the actor's function callback array and perform the same task (handing resources for each mesh off to the scene).

The basic task of the CAllocateHierarchy::CreateMeshContainer function will be as follows:

- Allocate a new CTriMesh object and attach it to the ID3DXMesh that it is passed by D3DX
- If the CActor is in managed mode, allocate space in the CTriMesh's internal attribute table to store the texture pointer and material for each subset that we are passed by D3DX. Copy the material data into the CTriMesh's internal attribute table and use the CActor's registered texture callback function to load the texture and return a texture pointer. These texture pointers are also stored inside CTriMesh's attribute table.
- If the CActor is in non-managed mode then the function will simply send the passed material data to the actor's registered attribute callback function which will store the texture and material data for each subset at the scene level. This function will return a new global attribute ID for each subset in the mesh which the function can use to lock the CTriMesh's index buffer and remap its attribute buffer.
- Store the newly created CTriMesh in the mesh container and return it to D3DX so that it can be attached to the frame hierarchy.

All of the above steps should seem very familiar. This code was seen in the previous chapter inside the CTriMesh::LoadMeshFromX function. All we have done is essentially duplicated it inside the CreateMeshContainer method of the CAllocateHierarchy object. This same code still exists in the CTriMesh::LoadMeshFromX function so that it can continue to be used as a standalone mesh object that does not have to be used with CActor or frame hierarchies. However, when the CTriMesh is part of the CActor's internal hierarchy, their own callback system will not be used and the CActor will manage the callbacks instead.

It is also worth noting that our CActor class can still be used to load an X file even if that X file does not contain a frame hierarchy and contains only a single mesh. In such a case, the CActor's hierarchy will consist of just a single frame (the root) with the mesh attached as an immediate child object. Single mesh CActors can be rendered and updated in exactly the same way, so the difference is invisible to the application using the CActor class.

Application Design

An understanding of the relationships between our various classes is vital. So let us break down the responsibilities for each object:

CScene

As we have come to expect, the scene object handles the loading of scene data and the rendering of that data at a high level. The application uses the CScene::LoadSceneFromX or CScene::LoadSceneFromIWF functions to load the data and set it up correctly. The scene loading methods have now been modified to be aware of actors.

The CScene::LoadSceneFromX function will now allocate a new CActor and register any resource callback functions with the actor before calling the LoadActorFromX function to populate the frame hierarchy with X file data. Each call to this function made by the application will load a new X file and will create a new CActor in the scene's CActor array. In our application, this function is called once to load the automobile representation from the X file.

The CScene::LoadSceneFromIWF (and all its support functions) are largely unchanged. In the previous chapter, it searched for external references stored inside the IWF file and used their names to load an X file into a CTriMesh. Now, this same function will allocate a new CActor when an external reference is found instead. It will then register the scene's resource callbacks with the newly created actor before calling the CActor::LoadActorFromX function to load the X file data.

All the other scene methods that are part of the framework still exist (AnimateObject, Render, etc.) and will be changed slightly to work with actors.

CActor

The CActor class encapsulates the loading, updating, and rendering of a hierarchy.

Its loading function CActor::LoadActorFromX will instantiate a temporary CAllocateHierarchy object and pass it into the D3DXLoadMeshHierarchyFromX function. When this function returns, the CActor's root frame pointer will point to the root frame of the newly loaded frame hierarchy. Any meshes attached to frames in the hierarchy will be of type CTriMesh. The scene will call this function to load any X files (either directly or via external references in an IWF file).

The CActor::RegisterCallback function is virtually identical to its cousin in the CTriMesh class. It allows the application to register callback function pointers. We covered the code to the scene's three resource callback functions in the previous workbook and these are unchanged. This CActor method simply stores the passed function pointers in an array so that they are accessible from the CAllocateHierarchy::CreateMeshContainer function (via its CActor pointer) when the hierarchy is being loaded.

This class will also have a method called DrawActorSubset which will be called by the scene to render **all** CTriMesh subsets in its hierarchy with a matching attribute ID. This function will traverse the hierarchy calling CTriMesh::DrawSubset for each of its contained meshes.

CTriMesh

This class is unchanged from our previous workbook. However, now objects of this type will be stored in the frame hierarchies of the actors to which they belong. In our application, we will be using all the meshes in non-managed mode, so the scene class will be responsible for setting the states before calling CActor::DrawSubset. This allows us to benefit from greater batching potential using global attribute IDs across all CTriMeshes in all CActors currently being used by the scene.

Although not demonstrated in this demo, it is also perfectly legal to use the CActor in managed mode. This is actually the default case when the attribute callback has not been registered with the CActor. When this is the case, each CTriMesh in the frame hierarchy will internally store the material and texture pointer used by the mesh for each of its subsets.

CAllocateHierarchy

The CAllocateHierarchy object is also introduced in this application. It is derived from the ID3DXAllocateHierarchy interface and is instantiated by CActor and passed into the D3DXLoadMeshHierarchyFromX function.

This object will contain four callback functions (CreateFrame, CreateMeshContainer, DestroyFrame, and DestroyMeshContainer) which will be called by the D3DX loading function to allocate the memory for the various hierarchy components as the hierarchy is being constructed. This allows us to customize the mesh and frame data that we are passed. For example, the CreateMeshContainer method provides us the flexibility to store the passed mesh data in a CTriMesh and add that to the mesh container instead. It also allows us to clone the mesh data into another format or optimize its vertex and index data.

Our CAllocateHierarchy class will also have a single member variable: a pointer to a CActor. When the CAllocateHierarchy object is instantiated by the CActor, the pointer to the actor is passed into the constructor and stored in the object. It will be used in the CreateMeshContainer function to access the resource callback functions that have been registered with the actor. This will allow the array of material and texture data that we are passed by D3DX for a given mesh to be registered and stored at the scene level. It will also allow for the re-mapping of each mesh’s attribute buffer into global attribute IDs.

CObject

The CObject class as been used in nearly all of our previous lab projects to represent meshes in the scene. This was necessary because a mesh itself does not store a world matrix. For example, in the last workbook, this object coupled together a CTriMesh with its assigned world matrix. CObject will now have a new member pointer added: a pointer to a CActor. Usually, only one of the pointers will be active at once (either the CActor pointer or the CTriMesh pointer) depending on whether the object represents just a single CTriMesh or a CActor. Although making these two pointers a union would seem the logical choice, there may be times when we might wish the CObject to represent both an actor and a separate mesh that both use the same world matrix. So the design decision was made to allow both pointers to exist simultaneously.

When the application uses the CScene::LoadSceneFromX function, a new CObject will be added to the scene. That object will contain a matrix and an actor. When the application calls CScene::LoadSceneFromIWF, several CObjects will typically be added to the scene. Meshes stored internally in the IWF (such as brushes from GILES™) will be loaded into a CTriMesh and attached to a new CObject and added to the scene's CObject array. Any external references stored in the IWF file will be loaded into a CActor, attached to a new CObject, and added to the scene CObject array. Therefore, when the scene is loaded from an IWF file, multiple CObjects will be added to the scene. Some objects in this array may contain actors (i.e., mesh hierarchies) while other objects may contain only simple CTriMesh pointers. Our scene rendering function will need to be able to render both types of CObject.

We know from previous lessons that the matrix stored in the CObject represents the world matrix that will be sent to the device before rendering the contained mesh. This behavior is still true when the CObject is being used to represent only a single mesh. However, when a CActor is encountered, this matrix is not used in quite the same way. Instead, this matrix will be used to position and orient the root frame of the hierarchy. The scene object will not simply set this matrix as the world matrix on the device since we know that would not work -- the actor might contain multiple meshes which all need to have their absolute world matrices generated and set separately before they are rendered.

The scene object will instead call the CActor::SetWorldMatrix method, passing in the matrix stored in CObject. The CActor will use this passed matrix to combine with the root frame matrix and traverse the tree, generating all absolute world matrices for each frame. As we know from our textbook discussions, combining the CObject matrix with the root frame matrix will affect all the world matrices that are generated below the root frame in the hierarchy. Once all absolute matrices in the hierarchy have been updated, we have all the matrices we need to render any mesh in the hierarchy. When we render a mesh, we simply use the absolute matrix stored in its owner frame as the world matrix that we set on the device. This is done in a separate rendering pass through the hierarchy.

From the application's perspective, things are no different with respect to moving objects around in our scene. When we pass the CObject's matrix into CActor::SetWorldMatrix, we are applying a transformation to the root frame which filters down to all other frames in the hierarchy. Therefore, we can still move the object around in our scene using only the CObject matrix.

The following diagram shows the relationships that exist between the various classes in our application. Notice that the scene stores a list of objects, each object stores an actor and each actor stores the root frame to an entire frame and mesh hierarchy.

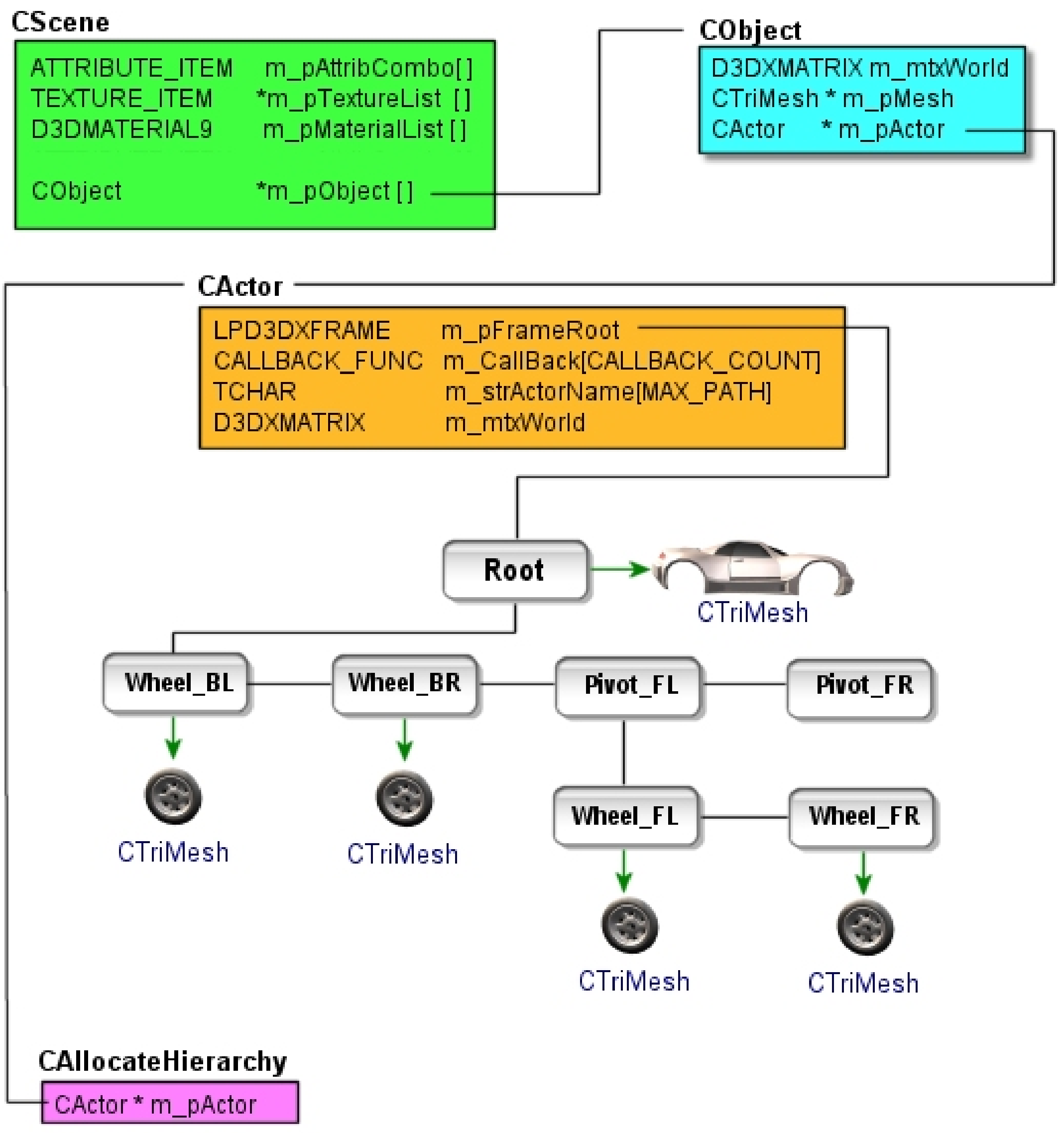


Figure 9.2

The CAllocateHierarchy object in Figure 9.2 is only instantiated temporarily (on the stack) by CActor for the call to D3DXLoadMeshHierarchyFromX.

Notice how the CActor itself also stores its own matrix. In fact, when we call CActor::SetWorldMatrix and pass in the CObject's matrix, this matrix is cached internally by the actor. The reason we do this is that many of the methods exposed by CActor (Draw, UpdateFrames, SetWorldMatrix, etc.) would usually require a total traversal of the hierarchy and a rebuilding of the absolute frame matrices stored in each frame. You may for example, issue several function calls to the actor which would cause the hierarchy to be traversed each time. To rid ourselves of unnecessary traversals, each function (such as SetWorldMatrix) will also except a Boolean parameter indicating whether we wish the function to update the matrices in the hierarchy immediately or whether we simply wish to store the state such that it can be used to update the hierarchy later via another function. This allows us to set the actor world matrix, apply some animation to the relative frame matrices, and then later call

CActor::UpdateFrameMatrices to traverse the hierarchy and rebuild the absolute matrices using the world matrix that was set in a previous call . Using this technique, the first two function calls would not update the hierarchy, but would simply set states. The final call would use these states to update the hierarchy in a single pass using those caches states. Without this mechanism, every call that alters the hierarchy would cause a hierarchy traversal and a rebuild of all absolute frame matrices.

Another thing that may seem strange in the above diagram is that our automobile frame hierarchy looks rather different from the diagrams we looked at in the textbook. There is another level of frames (called *pivots*) between the root and the front wheels. The reason these pivots are necessary is clear when we establish what our application will do. Every time the scene is rendered, the wheel frames of the car will be rotated around their local X axis. We will discuss the class that performs this task in a moment. The process is basically one of building an X axis rotation matrix that rotates some small number of degrees and multiplying it with the relative frame matrices for each wheel. This accumulative step will rotate the wheels a little more around their **local** X axis every cycle of the game loop. This will cause them to spin around like the wheels on a real automobile. The local X axis of each wheel can be perceived as sticking out of the page directly at you if you were viewing the wheel face on (Figure 9.3).

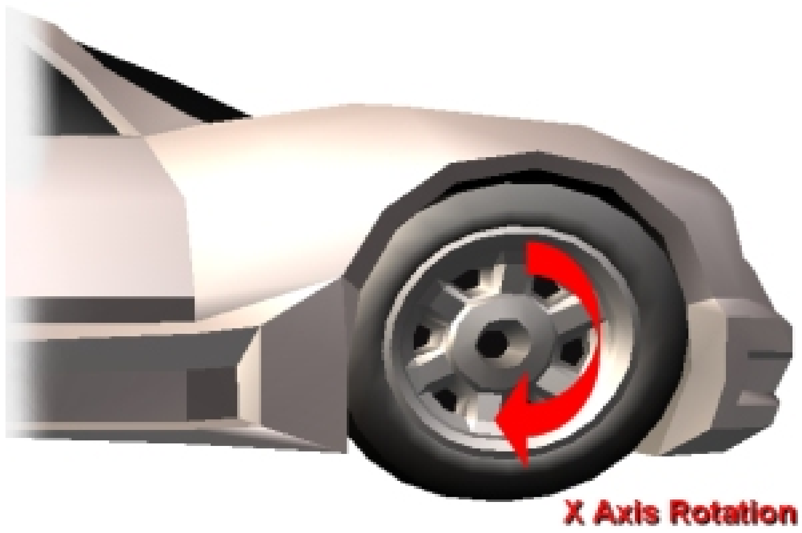


Figure 9.3

The red arrow shows the rotation around the X axis (which we cannot see in this diagram because we would be looking directly down it). It should be clear why rotating the wheel frame matrix X axis is the desired choice. The textbook taught us that applying rotations to the relative matrix of a frame causes a local rotation. Therefore, however much we rotate the root frame and change the orientation of each wheel in the world, each wheel will always rotate around the correct axis: the local X axis.

The problem we have is when we introduce the need for additional local rotations for that wheel frame. In our application, we also wish to give the user the ability to press the comma and period keys to steer the wheels left or right. If we look at Figure 9.4 we can see the local Y axis prior to any local rotations. It is clear that this is the axis we would wish to use for left/right wheel rotation.

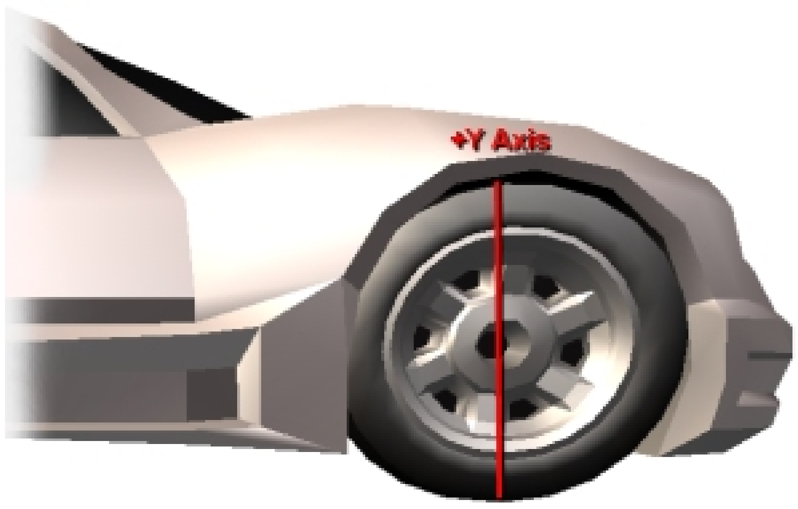


Figure 9.4

In Figure 9.4 we see the local Y axis of the wheel frame prior to any rotation. Just as the local X axis can be thought of as the wheel's right vector, this axis describes the wheels up vector. So far it all looks very promising.

So it would seem at first that our task is a simple one. For every update, we build an X axis rotation matrix and combine it with the wheel matrix causing the wheel to revolve. Then, when the user presses a comma or period key, we simply build a Y axis rotation matrix (with either a negative or positive degree value for left and right respectively) and apply that to the frame matrix of the wheel as well. Right?

Actually, no! The problem is that when we combine matrices with a relative frame matrix we are dealing with its local coordinate system. Imagine for example that the wheel has been rotated 45 degrees about its X axis and we then, based on user input, wanted to apply a Y rotation to make the wheel turn. The problem is, at this point, the entire local coordinate system has been rotated about the local X axis. So the local Y axis (the wheel up vector) is no longer pointing straight up relative to the car body. Instead it is oriented at an angle of 45 degrees around the wheel's local X axis (Figure 9.5). It is clear that applying a Y rotation to this matrix would not have the correct effect at all.

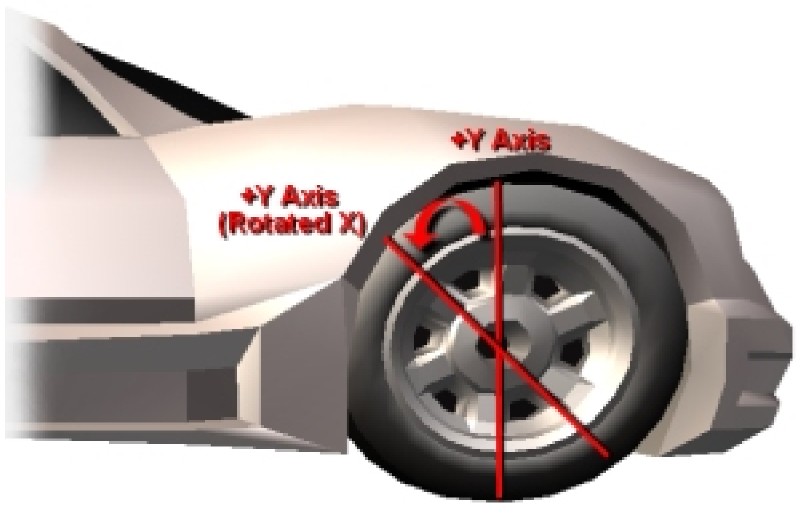


Figure 9.5

So, it seems we cannot use the frame matrix for both X *and* Y rotation. Maybe we could apply the Y rotation to the parent frame instead? No, that would not work either because the parent frame in this case would be the root frame which represents a position at the center of the automobile. While the orientation of the root frame's Y axis absolutely is the axis we need to rotate about, the wheel is offset in this frame of reference from that origin. We would not want the wheel to rotate about the center of the automobile instead of around its own center point. Applying a rotation to the parent would actually rotate the whole car about its Y axis, so that definitely is not the way to go either. We need these wheels to rotate about their own coordinate origins, but using the automobile up vector.

As it happens, there is a very simple way to do what we need and it is commonly used by artists to help game programmers with such tasks. The use of *dummy frames* in the hierarchy, which to the uninitiated may seem to prove no purpose, solves the problem. Two dummy frames exist in our automobile representation and they are positioned above the two front wheels in the hierarchy. Let us examine their purpose.

Figure 9.6 shows the position of the two dummy frames (Pivot_FL and Pivot_FR) in the hierarchy. These frames are positioned in the root frame of reference such that they describe a position at the center of the wheels. They have no default orientation (i.e., identity). In other words, these pivot frames describe the position of the wheels themselves (like the diagrams in the textbook). However, each pivot frame has a child wheel frame. These wheel frames also initially start out with an identity orientation but (and this is the important point) they also have a *zero translation vector*. Since they are defined as children in the pivot frame of reference, but with zero offset, this means that for each front wheel we have two frames on top of each other sharing the same space. To demonstrate this important point, if we were to also assign a wheel mesh to Pivot_FL and render this hierarchy, the mesh at Pivot_FL and the mesh at Wheel_FL would be in the exact same position.

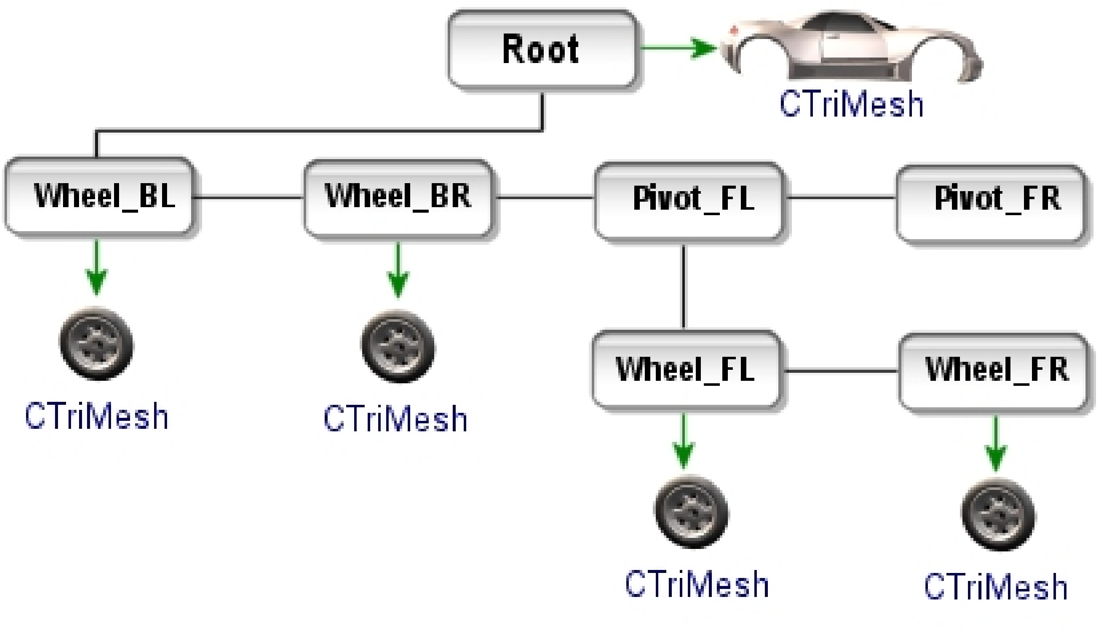


Figure 9.6

The usefulness of these seemingly redundant frames becomes clear when we consider the very important fact that Wheel_FL is defined in the Pivot_FL frame of reference.

If we apply a Y axis rotation to the Pivot_FL frame, we know that any child frames will inherit this rotation. We know for example that when we apply a rotation to the root frame, the four wheel frames will rotate about the root frame Y axis at a radius consistent with their specified offsets. In this case however, the Wheel_FL frame has not been offset from its parent frame. Therefore, we can think of the Pivot_FL frame's Y axis as also passing through the origin of the wheel frame's local coordinate system. Applying a rotation to the pivot will cause the wheel frame (and its mesh) to rotate about its own center point but also around the parent axis. The great thing is that because we have applied the Y axis rotation to the parent frame, we are free to rotate the wheel frame about its local X axis without any fear that its orientation will interfere with (or be corrupted by) other rotation operations.

Our task is simple then. During every update we will apply an X axis rotation to each of the wheel frame matrices, and when the user presses either the comma or period key, we will apply a Y axis rotation to the pivot frames. Problem solved.

CAnimation

The last new class we will introduce in this lesson is called CAnimation. This is a very simple object that will be used to explore hierarchical animation and prepare you for the following chapter where we will explore the DirectX animation controller in detail. The job of this object will be to apply rotations to frames in the hierarchy. There will be one CAnimation object created for each frame that needs to be animated.

The scene will create six animation objects (one for each of the wheels and two more for the pivot frames) and will use the CAnimation::Attach method to attach frame matrices in the hierarchy to the individual animation objects. The CAnimation object simply stores this matrix internally so that it knows which matrix to update when the scene object calls its RotationX, RotationY and RotationZ methods. It is these methods that build a rotation matrix and multiply it with its associated frame matrix. As you can probably imagine, the code to this class is extremely small.

> **Note**: Using this naming convention, an animation is assumed to be either an object or dataset that animates a single frame in the hierarchy. In our example X file, the car has six frames that need to be animated. Therefore, our demo contains six animations.

Source Code Walkthrough – CActor

With a high level discussion of this lab project behind us, we will now examine the code sections that we are not yet familiar with. We will start by looking at the CActor class, which is where most of the stuff that is new to us will be hiding. The class definition is contained in CActor.h and is shown below in its entirety. This should give you a feel for the interface it exposes to the application and the member variables it contains. This will be followed with a description of its member variables.

```
class CActor
{
public:
    enum CALLBACK_TYPE { CALLBACK_TEXTURE=0,
                         CALLBACK_EFFECT = 1,
                         CALLBACK_ATTRIBUTEID = 2,
                         CALLBACK_COUNT = 3 };
    // Constructor
             CActor( );
    virtual ~CActor( );

    bool      RegisterCallback   ( CALLBACK_TYPE Type, LPVOID pFunction, LPVOID pContext );
    HRESULT   LoadActorFromX     ( LPCTSTR FileName, ULONG Options,
                                   LPDIRECT3DDEVICE9 pD3DDevice );
    void      Release            ( );
    void      SetWorldMatrix     ( const D3DXMATRIX * mtxWorld = NULL,
                                        bool UpdateFrames = false );
    void      DrawActor          (  );
    void      DrawActorSubset    ( ULONG AttributeID );

    // Accessor functions
    LPCTSTR       GetActorName      ( ) const;
    CALLBACK_FUNC GetCallback       ( CALLBACK_TYPE Type ) const;
    ULONG         GetOptions        ( ) const;
    LPD3DXFRAME   GetFrameByName    ( LPCTSTR strName, LPD3DXFRAME pFrame = NULL ) const;

    // Private Functions
    private:
    void    DrawFrame           ( LPD3DXFRAME pFrame, long AttributeID = -1 );
    void    DrawMeshContainer   ( LPD3DXMESHCONTAINER pMeshContainer,
                                  LPD3DXFRAME pFrame, long AttributeID = -1 );
    void    UpdateFrameMatrices ( LPD3DXFRAME pFrame, const D3DXMATRIX * pParentMatrix );

    // Member Variables for This Class
    LPD3DXFRAME                   m_pFrameRoot;
    LPDIRECT3DDEVICE9             m_pD3DDevice;
    CALLBACK_FUNC                 m_CallBack[CALLBACK_COUNT];
    TCHAR                         m_strActorName[MAX_PATH];
    D3DXMATRIX                    m_mtxWorld;
    ULONG                         m_nOptions;
};
```

LPD3DXFRAME   m_pFrameRoot

When the actor's hierarchy is constructed in the CActor::LoadActorFromX method, this pointer is passed into the D3DXLoadMeshHierarchyFromX function and used to store the address of the root frame that is returned. This single pointer is the actor object's doorway to the hierarchy and the means by which it begins hierarchy traversal.

Below, you are reminded of the layout of the D3DXFRAME structure:

```
typedef struct _D3DXFRAME
{
    LPSTR Name;
    D3DXMATRIX TransformationMatrix;
    LPD3DXMESHCONTAINER pMeshContainer;
    struct _D3DXFRAME *pFrameSibling;
    struct _D3DXFRAME *pFrameFirstChild;
} D3DXFRAME, *LPD3DXFRAME
```

Because we need to store additional information in our frame structure, such as the absolute world matrix of a frame (and not just the relative matrix provided by the vanilla D3DX version shown above), we will derive our own structure from D3DXFRAME to contain this additional matrix information. The frame structure used throughout our actor's hierarchy is shown below.

```
struct D3DXFRAME_MATRIX : public D3DXFRAME
{
    D3DXMATRIX  mtxCombined;    // Combined matrix for this frame.
};
```

In the textbook, we also covered the D3DXMESHCONTAINER structure that is used throughout the hierarchy to contain mesh data for a frame. You are reminded of this structure again below. It should be very clear to you what each member of this structure is used for after our discussion in the textbook.

```
typedef struct _D3DXMESHCONTAINER
{
    LPSTR Name;
    D3DXMESHDATA MeshData;
    LPD3DXMATERIAL pMaterials;
    LPD3DXEFFECTINSTANCE pEffects;
    DWORD NumMaterials;
    DWORD *pAdjacency;
    LPD3DXSKININFO pSkinInfo;
    struct _D3DXMESHCONTAINER *pNextMeshContainer;
} D3DXMESHCONTAINER, *LPD3DXMESHCONTAINER
```

We will also wish to derive from this structure because we want to represent our meshes in the hierarchy using the CTriMesh object we created in the previous workbook. You will recall that the D3DXMESHDATA member of this structure would normally contain the ID3DXMesh interface for the mesh being stored here. The structure that we will use is derived from this one, with a single extra member added: a CTriMesh pointer:

```
struct D3DXMESHCONTAINER_DERIVED : public D3DXMESHCONTAINER
{
    CTriMesh * pMesh;          // Our wrapper mesh.
};
```

As you can see, our mesh container now contains a pointer to a CTriMesh. The CAllocateHierarchy::CreateMeshContainer function will take the passed ID3DXMesh, create a CTriMesh from it and attach it to the mesh container, instead of storing the ID3DXMesh pointer directly in the mesh container's D3DXMESHDATA member.

> **Note:** In our application we also store the CTriMesh's underlying ID3DXMesh interface pointer in the mesh container's D3DXMESHDATA member. While there might seem little point in doing this, D3DX provides many helper functions that calculate things like the bounding volume of a frame. These functions expect the mesh data to be stored in the D3DXMESHDATA member of the frame as they have no knowledge that we are now using our own CTriMesh object to store and render mesh data. Without us also storing the mesh interface used by the CTriMesh in this member, these functions will not work correctly.

The remaining member variables of the CActor class will now be discussed.

LPDIRECT3DDEVICE m_pD3DDevice

When the application calls the CActor::LoadActorFromX function it must also pass a pointer to the Direct3D device which will be the owner of any meshes in that hierarchy. The CActor passes this device pointer into the D3DXLoadMeshHierarchyFromX function and also stores the address in this member variable. It will need access to this pointer during rendering traversals of the hierarchy so that the absolute world matrices of each frame can be set as the device world matrix prior to rendering any of its attached mesh containers.

CALLBACK_FUNC m_CallBack[CALLBACK_COUNT]

As with the CTriMesh class, the actor has a static array that can be used to store pointers to resource callback functions. The CALLBACK_TYPE enumeration is also familiar to us because the CTriMesh class used this same enumeration within its namespace. The members of this enumeration describe the three possible callback functions which can be registered with the actor. The value of each member in this enumeration also describes the index in the callback array where the matching function pointer will be stored. For example, the CALLBACK_ATTRIBUTEID function pointer will be stored in m_CallBack[2].

TCHAR m_strActorName[MAX_PATH]

This string is where the name of the actor will be stored. This name is the name of the X file passed into the LoadActorFromX method. The actor's name will often be retrieved by the scene object to make sure that the same hierarchical X file does not get loaded more than once. When an external reference is found in an IWF file for example, a search of all the currently loaded actor names will be performed to make sure it does not already exist. If it does, then we will not create a new actor, but will simply point the new CObject's CActor pointer at the one that already exists. When we do this we are referencing the actor with multiple CObjects.

D3DXMATRIX m_mtxWorld

This is where the CActor will cache a copy of the matrix passed into its SetWorldMatrix function. This is the matrix that is used to perform updates to the hierarchy. This matrix will contain the world space position and orientation of the actor.

ULONG m_nOptions

This member will contain a combination of D3DXMESH creation flags used to create the meshes in the hierarchy. The application will pass these flags into the LoadActorFromX function where they will be stored and used later in the CAllocateHierarchy::CreateMeshContainer function to create the ID3DXMesh data in the application requested format. This is another reason why the CAllocateHierarchy object will need access to the CActor which is using it.

CActor::CActor()

The CActor constructor takes no parameters and initializes the object's member variables to zero. Its internal world matrix is set to an identity matrix, essentially describing this actor as currently existing at the origin of world space. The callback array is also initialized to zero.

```
CActor::CActor()
{
    // Reset / Clear all required values
    m_pFrameRoot      = NULL;
    m_pD3DDevice      = NULL;
    m_nOptions        = 0;

    ZeroMemory( m_strActorName, MAX_PATH * sizeof(TCHAR) );
    D3DXMatrixIdentity( &m_mtxWorld );

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

CActor::~CActor()

As with many of our classes, the destructor does not actually perform the cleanup code itself but rather calls the object's Release function to perform this task. This is beneficial, as the Release function can be used to wipe the information from a CActor without destroying it. This would allow us to use the same CActor object to load another X file once we are finished with the data it currently contains. As the Release function is going to have to be written anyway, let us not duplicate this same cleanup code in the destructor. So we simply call it:

```
CActor::~CActor( )
{
    // Release the actor objects
    Release();
}
```

CActor::Release()

A couple of items are quite interesting about this cleanup code. Firstly, notice the use of the D3DXFrameDestroy function to destroy the actor's entire frame and mesh hierarchy. This function was discussed in the textbook. We pass it an ID3DXAllocateHierarchy derived object and the root frame of the hierarchy to be destroyed. D3DX will then step through each frame in the hierarchy and call the DestroyFrame and DestroyMeshContainer callback methods of the passed allocator object. We will see that it is these callback functions that we must implement to perform the actual removal of frame and mesh data from memory. We will cover the CAllocateHierarchy class next.

So, this function instantiates a CAllocateHierarchy object and passes it into the global D3DXFrameDestroy function. The instance of the allocation object used to destroy the hierarchy does not need to be the same one used to create it, which is why we can create a temporary allocation object on the stack in this way. Notice the passing of the **this** pointer into the constructor, so that the allocation object can store a pointer to the actor which is using it.

```
void CActor::Release()
{
    CAllocateHierarchy Allocator( this );

    // Release objects (notice the specific method for releasing the root frame)
    if ( m_pFrameRoot      ) D3DXFrameDestroy( m_pFrameRoot, &Allocator );
    if ( m_pD3DDevice      ) m_pD3DDevice->Release();

    // Reset / Clear all required values
    m_pFrameRoot      = NULL;
    m_pD3DDevice      = NULL;
    m_nOptions        = 0;

    ZeroMemory( m_strActorName, MAX_PATH * sizeof(TCHAR) );
    D3DXMatrixIdentity( &m_mtxWorld );

    // Callbacks NOT cleared here!!!
}
```

When the D3DXFrameDestroy function returns, the entire frame hierarchy and all its meshes will be removed from memory. The Release function then relinquishes its claim on the device and sets its member variables to their initial values.

Notice however that the Release function does not clear the actor's resource function callback array. This is very convenient when you wish to re-use the object. Usually, even if you want to erase the actor's data and reuse it, the same callback functions can be used to load textures and materials. This allows us to release the actor's data without have to re-register the same callback functions. As the callback array is a statically allocated array, it will naturally be destroyed when the object is destroyed.

CActor::RegisterCallback

This function is called by the application to register any of three types of resource callback functions prior to calling the LoadActorFromX method. The passed function pointers and their associated contexts (in our demo, a CScene pointer) are simply stored in the callback array. These callback functions essentially describe whether the actor is in managed or non-managed mode. If the function callbacks CALLBACK_TEXTURE or CALLBACK_EFFECT are being used, then all the meshes in the hierarchy will be created in managed mode. We are currently not using the CALLBACK_EFFECT callback until Module III when we cover effect files. If the CALLBACK_ATRIBUTEID callback is registered, all meshes in the hierarchy will be created in non-managed mode.

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

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

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

These function pointers are not actually used directly by CActor, but are accessed by the CAllocateHierarchy object each time a mesh container is to be created. If the CALLBACK_TEXTURE function pointer exists (m_Callback[0]), then the texture filenames for each material in the mesh being created will be passed to this callback. The callback is responsible for creating the texture and returning a pointer back to the CAllocateHierarchy::CreateMeshContainer function. The texture pointer and the material will be stored inside the CTriMesh, which will then manage its own state changes and have the ability to render itself in a self-contained manner.

If the CALLBACK_ATTRIBUTEID function has been registered (m_CallBack[2]) then the CAllocateHierarchy::CreateMeshContainer function will create all CTriMeshes in the hierarchy in non-managed mode. The texture and material information passed to the CreateMeshContainer function by D3DX will simply be dispatched to this callback. The scene will create and store the textures and materials in its own arrays and return a global attribute ID back to the CreateMeshContainer function for that texture and material combination. The CreateMeshContainer function will then lock the attribute buffer of the CTriMesh and re-map its attribute IDs to this new value. When in this mode, the scene object will be responsible for the state management and subset rendering of each actor's meshes.

CActor::LoadActorFromX

You would be forgiven for thinking that it is in the CActor::LoadActorFromX function that all the hard work happens, and therefore it is expected to be quite large in size. But as you can see, this function is actually very small since it uses the D3DXLoadMeshHierarchyFromX function to handle most of the heavy lifting. Indeed, it is in the CAllocateHierarchy::CreateMeshContainer function that most of the loading code is situated. We will take a look at that class next.

This function should be called by the application only after the resource callback functions have been registered. It will be passed the name of the X file that needs to be loaded along with any mesh creation flags. These mesh creation flags are a combination of D3DXMESH options, which we have used for mesh creation throughout the last lesson. They describe the resource pools we would like the mesh index and vertex buffers created in. The function should also be passed a pointer to the Direct3D device which will own the meshes.

The first thing the function does is instantiate an object of type CAllocateHierarchy. The **this** pointer is passed into the constructor so that its CreateMeshContainer callback function will be able to access the actor's resource callback functions and the mesh creation flags for mesh container creation.

The passed device pointer is stored in the actor's member variable and the reference count incremented. The mesh creation options are also stored by the actor.

Next we call the function that kick starts the whole loading process, D3DLoadMeshHierarchyFromX.

```
HRESULT CActor::LoadActorFromX(LPCTSTR FileName,ULONG Options,LPDIRECT3DDEVICE9 pD3DDevice)
{
    HRESULT hRet;
    CAllocateHierarchy Allocator( this );

    // Validate parameters
    if ( !FileName || !pD3DDevice ) return D3DERR_INVALIDCALL;

    // Store the D3D Device here
    m_pD3DDevice = pD3DDevice;
    m_pD3DDevice->AddRef();

    // Store options
    m_nOptions = Options;

    // Load the mesh heirarchy
    hRet = D3DXLoadMeshHierarchyFromX( FileName, Options, pD3DDevice, &Allocator,
                                       NULL, &m_pFrameRoot, NULL );
    if ( FAILED(hRet) ) return hRet;

    // Copy the filename over
    _tcscpy( m_strActorName, FileName );

    // Success!!
    return D3D_OK;
}
```

We pass the D3DX loader function the name of the file, the mesh creation options, the device, and a pointer to our ID3DXAllocateHierarchy derived object. The four member function of this object will be used as callback functions by D3DXLoadMeshHierarcdhyFromX. NULL is passed in as the fifth parameter, as we have no need to parse custom data objects. If we did, this is where we would pass in a pointer to our ID3DXLoadUserData derived object. For the sixth parameter, we pass in the address of the actor root frame pointer. On function return, this will point to the root frame of the hierarchy. Finally, we pass in NULL as the last parameter as we have not yet covered the ID3DXAnimationController interface. This will be covered in the next chapter.

If the D3DXLoadMeshHierarchyFromX function is successful, this function copies the filename passed in and stores it in the CActor::m_strActorName member variable. This will be used for actor lookups by the scene to avoid loading multiple actors with the same X file data.

CActor::SetWorldMatrix

Once the CActor has had its data loaded via the above function, it is ready to be positioned in the world. We can set the world space orientation and position of the actor (and all its child frames and meshes) using the SetWorldMatrix function.

This function should be passed a matrix describing the new world space transformation matrix for the actor. We can think of this as actually positioning the root frame in the world. Essentially, the passed matrix will become the root frame's parent frame of reference.

Our CScene class will store each CActor pointer in a CObject along with a world matrix. Therefore, when the scene updates the position of the CObject, its world matrix is passed into its CActor's SetWorldMatrix function to generate all the absolute world matrices for each frame. If no valid matrix is passed, then we will set the actor's matrix to identity, for safety.

```
void CActor::SetWorldMatrix( const D3DXMATRIX * mtxWorld, bool UpdateFrames  )
{
    // Store the currently set world matrix
    if ( mtxWorld )
        m_mtxWorld = *mtxWorld;
    else
        D3DXMatrixIdentity( &m_mtxWorld );

    // Update the frame matrices
    if ( UpdateFrames ) UpdateFrameMatrices( m_pFrameRoot, mtxWorld );
}
```

The function also takes a Boolean parameter called UpdateFrames which tells it whether you would like the traversal of the hierarchy and the updating of frame matrices to be performed immediately. If you pass 'true' then the CActor::UpdateFrameMatrices method will be called to traverse the hierarchy and update all the absolute world matrices of each frame. After the call to UpdateFrameMatrices, the absolute matrix in each frame contains the world matrix that should be set on the device in order to render any attached meshes. If 'false' is passed, the input matrix is simply stored in the actor's member variable and the update process can be deferred until a later time.

CActor::UpdateFrameMatrices

This function is responsible for stepping through the hierarchy in a recursive fashion and generating the absolute world matrices for each frame (stored in D3DXFRAME_MATRIX::mtxCombined). It can be called by the application, but is usually called by the SetWorldMatrix function above to recalculate the frame matrices when a new world matrix has been set.

The function is usually passed a pointer to the root frame as its first parameter. Its second parameter is the world space transformation matrix to be applied. The function recursively calls itself, so the changes filter down the hierarchy.

If you examine the SetWorldMarix call above, you can see that it calls this function passing in the root frame and the new world space transformation for the root frame (essentially the position of the actor in the scene). Let us now see what the function does with that information.

```
void CActor::UpdateFrameMatrices( LPD3DXFRAME pFrame, const D3DXMATRIX * pParentMatrix )
{
    D3DXFRAME_MATRIX * pMtxFrame = (D3DXFRAME_MATRIX*)pFrame;

    if ( pParentMatrix != NULL)
        D3DXMatrixMultiply( &pMtxFrame->mtxCombined,
                            &pMtxFrame->TransformationMatrix,
                            pParentMatrix);
    else
        pMtxFrame->mtxCombined = pMtxFrame->TransformationMatrix;

    // Move onto sibling frame
    if ( pMtxFrame->pFrameSibling )
         UpdateFrameMatrices( pMtxFrame->pFrameSibling, pParentMatrix );

    // Move onto first child frame
    if ( pMtxFrame->pFrameFirstChild )
         UpdateFrameMatrices( pMtxFrame->pFrameFirstChild, &pMtxFrame->mtxCombined );
}
```

First, the function casts the passed D3DXFRAME to its correct derived type. This is D3DXFRAME_MATRIX, which you will recall contains an extra matrix member (mtx_Combined) to store the world matrix for each frame during update traversals.

The function recursively calls itself until every frame in the hierarchy has been visited and its absolute world matrix (stored in mtx_Combined) has been calculated. After the function has returned program flow back to the caller, the combined matrix stored in each frame structure will contain the world matrix of that frame that should be set on the device in order to render any meshes attached to that frame.

The passed matrix is combined with the relative matrix stored in that frame to calculate its world matrix. If the frame has a sibling, then the function is called for its sibling too, so that it has a chance to combine its relative matrix with the passed matrix also. As the siblings are stored as a linked list, this causes all siblings of the frame to be visited and have their world matrices generated.

The next and final line of the function is very important to understanding the matrix concatenation process. If the frame has a child frame, then the function is called again for the child frame. Only this time, the matrix passed in is the world matrix that has just been generated for the current frame (i.e., mtx_Combined). It is the world matrix of the parent frame which describes the child frame's frame of reference. In the instance of the function that processes the child, its relative matrix is combined with the parent's **world** matrix, generating the world matrix for the child frame also. As the child position and orientation is described by its relative matrix in the parent's frame of reference, this will correctly position the child frame in world space such that it is correctly offset from its parent. If the parent matrix had more than one child frame, then the children will be connected in a linked list by their sibling pointers. In this case, the instance of the function processing the child will also cause each sibling to be processed using the same parent world matrix.

This function is actually very small but does a lot of work due to its recursive nature. Study this code and make sure you understand it.

CActor::DrawActorSubset

Because the CActors in our application have their AttributeID callback function registered, all CTriMeshes in the hierarchy are created in non-managed mode. This means the CTriMeshes in the scene have their attribute buffers filled with scene global attribute IDs. As the CTriMeshes themselves do not contain an attribute table in this mode, it is the responsibility of the scene object to render any CActors. This is done in the CScene::Render method which we will look at in a moment.

The scene must loop through all attributes it has in its attribute array and call the CActor::DrawSubset method for each actor. The scene must make sure that before it does, it sets the correct texture and material that is mapped to that attribute ID. This is virtually no different from how the scene rendered standalone CTriMeshes in non-managed mode in the previous lesson. The only difference is that the scene calls CActor::DrawActorSubset instead of CTriMesh::DrawSubset.

The DrawActorSubset method filters the rendering request down to each of its meshes. It does so by calling the CActor::DrawFrame method, passing in the subset number for which rendering has been requested. The DrawFrame method is another recursive function that visits every frame in the hierarchy searching for mesh containers. When a mesh container is found attached to a frame, the frame’s world matrix (mtx_Combined) is set on the device, CTriMesh::DrawSubset is called for each mesh container attached to that frame, and traversal continues until every mesh has had a chance to render the requested subset.

As all of this traversal work is recursively performed inside the DrawFrame method, the DrawActorSubset method is a simple wrapper around the DrawFrame call. We will cover the DrawFrame method in a moment.

```
void CActor::DrawActorSubset( ULONG AttributeID )
{
    // Draw the frame heirarchy
    DrawFrame( m_pFrameRoot, (long)AttributeID );
}
```

CActor::DrawActor

If the CActor has not has the CALLBACK_ATTRIBUTEID function registered, but instead has had either CALLBACK_TEXTURE or CALLBACK_EFFECT registered, all meshes in the hierarchy will be created in managed mode. When this is the case, we can essentially describe the actor as being in managed mode.

In managed mode, each CTriMesh in the hierarchy will contain an attribute table containing the texture and material that should be set for each subset. This means each mesh in the hierarchy has the ability to render itself in a self-contained manner and does not require the scene to set the textures and materials on the device prior to subset rendering. Just as CTriMesh has a function called CTriMesh::Draw which can be used render a managed mode CTriMesh in its entirety, so too does the actor.

The CActor::DrawActor function should only be used when the actor is in managed mode. Like the above function, it actually just wraps a call to the DrawFrame function. But unlike the DrawActorSubset call, no subset ID is passed into it. When the DrawFrame function has not been passed an attribute ID, it will render any CTriMeshes that exist in the hierarchy using the CTriMesh::Draw function. We know that this will cause the mesh to render all its subsets, taking care of state changing between each one by itself. Obviously, this can cause many redundant state changes, but managed mode is easy to use if you simply wish to view an asset that you have created or have not yet written the attribute callback function.

```
void CActor::DrawActor( )
{
    // Draw the frame heirarchy
    DrawFrame( m_pFrameRoot );
}
```

CActor::DrawFrame

It is in this function (called by both functions discussed previously) that the main rendering traversal is done. It is recursively called such that it processes every frame in the hierarchy.

For the current frame being processed (which is initially the root when called by the prior functions) its mesh container pointer will be non-NULL if a mesh container is attached. If multiple mesh containers have been assigned to this frame, then each mesh container will be arranged in a linked list via the D3DXMESHCONTAINER::pNextMeshContainer pointer.

If the mesh container pointer is not NULL, then a mesh exists and will need to be rendered. We set the frame world matrix, mtx_Combined (generated by a prior call to SetWorldMatrix or UpdateFrameMatrices), as the world matrix on the device. This matrix describes the world transform for all meshes in this frame.

We next initialize a for loop to step through all mesh containers assigned to this frame. For each one found, we call the CActor::DrawMeshContainer method to render the CTriMesh. How this function operates depends on the attribute ID passed into the function. If an attribute ID has been passed, then DrawMeshContainer will call CTriMesh::DrawSubset using this ID to render the requested subset stored in the mesh. If no attribute ID has been passed, as is the case when this function is called by the DrawActor method, the actor is assumed to be in managed mode and the DrawMeshContainer will render its mesh using CTriMesh::Draw to render all subsets at once.

```
void CActor::DrawFrame( LPD3DXFRAME pFrame, long AttributeID /* = -1 */ )
{
    LPD3DXMESHCONTAINER pMeshContainer;
    D3DXFRAME_MATRIX  * pMtxFrame  = (D3DXFRAME_MATRIX*)pFrame;

    // Set the frames combined matrix
    if (pMeshContainer) m_pD3DDevice->SetTransform( D3DTS_WORLD, &pMtxFrame->mtxCombined );
    // Loop through the frame's mesh container linked list
    for ( pMeshContainer = pFrame->pMeshContainer;
          pMeshContainer;
          pMeshContainer = pMeshContainer->pNextMeshContainer )
    {
        // Draw this container
        DrawMeshContainer( pMeshContainer, pFrame, AttributeID );

    } // Next Container

    // Move onto next sibling frame
    if ( pFrame->pFrameSibling ) DrawFrame( pFrame->pFrameSibling, AttributeID );

    // Move onto first child frame
    if ( pFrame->pFrameFirstChild ) DrawFrame( pFrame->pFrameFirstChild, AttributeID );
}
```

The familiar lines at the bottom of this function allow us to step through any sibling frames attached to the current frame, and the step down a level in the hierarchy to the child frames, if they exist. At the end of this function, all frames will have been visited. If an attribute ID was passed (non-managed mode), then all mesh subsets in the hierarchy with the requested ID will have been rendered. If no attribute ID is passed (-1 is the default) then the actor is in managed mode and all meshes in the hierarchy will have been entirely rendered.

CActor::DrawMeshContainer

The DrawMeshContainer function is called once by the previous function for each mesh container that exists in the tree. A mesh container linked to a frame contains a single CTriMesh, but there may be multiple mesh containers linked to a single frame. The job of this function, as we saw above, is to render the CTriMesh stored within each mesh container appropriately.

```
void CActor::DrawMeshContainer( LPD3DXMESHCONTAINER pMeshContainer,
                                LPD3DXFRAME pFrame, long AttributeID /* = -1 */ )
{
    D3DXFRAME_MATRIX           * pMtxFrame  = (D3DXFRAME_MATRIX*)pFrame;
    D3DXMESHCONTAINER_DERIVED * pContainer = (D3DXMESHCONTAINER_DERIVED*)pMeshContainer;
    CTriMesh                   * pMesh      = pContainer->pMesh;

    // Validate requirements
    if ( !pMesh ) return;

    // Render the mesh
    if ( AttributeID >= 0 )
    {
        // Set the FVF for the mesh
        m_pD3DDevice->SetFVF( pMesh->GetFVF() );
        pMesh->DrawSubset( (ULONG)AttributeID );

    } // End if attribute specified
    else
    {
        pMesh->Draw( );

    } // End if no attribute specified
}
```

This function is very simple thanks to the fact that we are using our CTriMesh class to represent mesh data. First it fetches a pointer to the CTriMesh object stored in the mesh container. Next, a check is done on the AttributeID passed in. If it is less than zero (-1) then this actor is in managed mode and the mesh can render itself (setting its own texture and material states before rendering each of its subsets). When this is the case, the function simply calls CTriMesh::Draw to instruct the mesh to render itself entirely.

If this is not the case, and the passed attribute ID is larger than –1, then the scene is rendering the individual mesh subsets, and all we wish to render is the requested subset of the mesh stored here. So we set the correct FVF flags for the mesh and use the CTriMesh::DrawSubset method to render only the requested subset. Notice that in this implementation of the function, the passed frame pointer is not used. We may very well need this later on though, and it is very useful for the rendering function to have access to its parent.

CActor Accessor Functions

The CActor class exposes several state accessor function so the application can performs tasks such as: retrieve a pointer to one of its registered callback functions, inquire about its mesh creation functions, or retrieve the name of the actor (i.e., the name of the X file).

```
CALLBACK_FUNC CActor::GetCallback( CALLBACK_TYPE Type ) const
{
    return m_CallBack[Type];
}
```

```
ULONG CActor::GetOptions( ) const
{
    return m_nOptions;
}
```

```
LPCTSTR CActor::GetActorName( ) const
{
    return m_strActorName;
}
```

CActor::GetFrameByName

This method is used to search an actor’s hierarchy for a frame with a matching name. The function will return a pointer to this frame if a name match is found. You will recall that in the X file, frames will often be assigned a name, and this is vital if we intend to animate them. We will see in our demo how the scene uses this function, after the automobile has been loaded, to retrieve pointers to the wheel and pivot frames. We then attach the relative matrices of these frames to CAnimation objects that can modify the matrices and allow us to spin the wheels.

The function is passed a string containing the name of the frame we wish to find, followed by a pointer to a frame from which you wish the search to begin. This will usually be the root frame so that the entire hierarchy is searched, but alternatively, we could limit our search to only a given sub-tree if desired.

```
LPD3DXFRAME CActor::GetFrameByName( LPCTSTR strName, LPD3DXFRAME pFrame /* = NULL */ ) const
{
    LPD3DXFRAME pRetFrame = NULL;

    // Any start frame passed in?
    if ( !pFrame ) pFrame = m_pFrameRoot;

    // Does this match ?
    if ( _tcsicmp( strName, pFrame->Name ) == 0 ) return pFrame;

    // Check sibling
    if ( pFrame->pFrameSibling )
    {
        pRetFrame = GetFrameByName( strName, pFrame->pFrameSibling );
        if ( pRetFrame ) return pRetFrame;

    } // End if has sibling

    // Check child
    if ( pFrame->pFrameFirstChild )
    {
        pRetFrame = GetFrameByName( strName, pFrame->pFrameFirstChild );
        if ( pRetFrame ) return pRetFrame;

    } // End if has sibling

    // Nothing found
    return NULL;
}
```

It should be obvious by now that this will be another recursive function because of its need to traverse the hierarchy. Starting at the frame passed in, we compare its name against the passed name. If the frame name matches the string name then we have found our match and return this frame immediately.

If it does not match then a check of the sibling pointer of the current frame is made and the function is recursively called if a sibling exists. We know this will, in turn, cause all siblings in the linked list to be visited and their names checked. When program flow returns back to the current instance, if pRetFrame is not NULL then it means that one of the sibling frames has a name that matched, so we return it. Otherwise, we recursively check the children in the same way.

If we get to the end of the function and there is no frame in the hierarchy with a matching name we return NULL. We will see this function being used to connect the CAnimation objects to the various frames in the hierarchy when we examine the modifications to the CScene class.

CActor::SaveActorToX

It is sometimes useful to be able to save the contents of an actor out to an X file. Perhaps you have altered the data in some way and would like the changes to be preserved. While it may not be an extremely useful option within your actual game, having this ability will certainly come in useful when you are writing your own tools (e.g., a hierarchy editor). In fact, in our next workbook we will build just such a tool (called Animation Splitter) that uses the CActor object to load an X file, split any animation data stored in that X file into multiple animations, and then save the re-organized actor and animation data back out to the X file as a hierarchy. You will understand why we build such a tool in the next chapter when we discuss the D3DX animation system. For the time being, just know that there may be times when you will want your actor's frame hierarchy saved to an X file.

Before we look at the CActor::SaveActorToX method's source code which is a mere few lines in length, let us first take a look at the global D3DX helper function it uses to perform the task.

The D3DXSaveMeshHierarchyToFile Function

The D3DX library exposes a global function that an application can use to save an entire frame/mesh hierarchy out to an X file. It can be thought of as the opposing function to the D3DXLoadMeshHierarchyFromX function in that information flows from the hierarchy in memory out to a new X file, whereas in the case of the D3DXLoadMeshHierarchyFromX function, the information flow is in the other direction. The method is shown below along with a description of each of its parameters.

```
HRESULT WINAPI D3DXSaveMeshHierarchyToFile
(
    LPCSTR pFilename,
    DWORD XFormat,
    const D3DXFRAME *pFrameRoot,
    LPD3DXANIMATIONCONTROLLER pAnimController,
    LPD3DXSAVEUSERDATA pUserDataSaver
);
```

LPCSTR pFilename
This is a string containing the name of the new X file that will be created.

DWORD XFormat
This parameter is used to tell the function whether the application would like the X file to be created as a text or binary X file. We can pass one of the following D3DXF_FILEFORMAT flags:

#define	Value	Description
D3DXF_FILEFORMAT_BINARY	0	Legacy-format binary file.
D3DXF_FILEFORMAT_COMPRESSED	2	Compressed file. May be used in combination with other D3DXF_FILEFORMAT_ flags. This flag is not sufficient to completely specify the saved file format.
D3DXF_FILEFORMAT_TEXT	1	Text file.

const D3DXFRAME **pFrameRoot*
All that is needed to gain access to the frame hierarchy for saving is the root frame. This function can use the root to traverse down through the tree and save off any frames, matrices, meshes, texture filenames and materials it finds. The result will be an X file containing all the information used by the hierarchy.

LPD3DXANIMATIONCONTROLLER ***pAnimController***
We will see in the next chapter that when an X file that contains animation data is loaded, the D3DXLoadMeshHierachyFromX function will return a pointer an ID3DXAnimationController interface. We can think of the animation controller as being a manager object that contains all the animation data used by the hierarchy. It presents the application with a means to play back those animations via its exposed methods. For now, our actor will pass in NULL for this parameter as it does not yet support animation data. In the next workbook, we will upgrade our actor so that it also includes animation support. When this is the case, the save function must also be able to save off any animation data to the X file. Therefore, in the next chapter, when we upgrade our actor, this parameter will be passed a pointer to the actor's animation controller interface.

LPD3DXSAVEUSERDATA ***pUserDataSaver***
When loading an X file using D3DXLoadMeshHierarchyFromX, we have the option of passing in a pointer to an ID3DXLOADUSERDATA derived object, whose methods would be called by D3DX when custom data objects are discovered in the X file. When saving the X file, D3DX will require the application to write out custom data chunks as well. Thus, the abstract ID3DXSAVEUSERDATA base class is the complementary interface for saving. Although we will not use this parameter in our demo, if you open up d3dxanim.h you will see that it is an interface with two methods that must be implemented in the derived class. We will simply pass in NULL as we have no custom chunks in our current application.

Finally, we will look at the code to the CActor::SaveActorToX method:

```
HRESULT CActor::SaveActorToX( LPCTSTR FileName, ULONG Format )
{
    HRESULT hRet;

    // Validate parameters
    if ( !FileName ) return D3DERR_INVALIDCALL;

    // If we are NOT managing our own attributes, fail
    if ( GetCallback( CActor::CALLBACK_ATTRIBUTEID ).pFunction != NULL )
         return D3DERR_INVALIDCALL;

    // Save the hierarchy back out to file
    hRet = D3DXSaveMeshHierarchyToFile( FileName, Format, m_pFrameRoot, NULL, NULL );

    if ( FAILED(hRet) ) return hRet;

    // Success!!
    return D3D_OK;
}
```

The application passes in the name of the X file and the format (binary vs. text) of the file. This request is passed straight into the D3DXSaveMeshHierarchyToFile function, along with the root frame of the actor. Notice that the function returns failure if the actor is in non-managed mode. The reason for this will be discussed in more detail later in this workbook when we cover the CAllocateHierarchy::CreateMeshContainer method. Ultimately, in non-managed mode, the application may have remapped all the subset IDs into a global pool of textures and materials. Because of this, the D3DXSaveMeshHierarchyToFile method will no longer have access to the materials and texture filenames used by each subset in a mesh. Therefore, saving a non-managed actor would result in meshes being written out with incorrect texture and material data. As you are only likely to use the CActor::SaveActorToX in a situation where it is being used as part of a tool you are developing, it is a limitation we can work with. Unlike a real-time game situation, using managed mode actors in a tool is acceptable since performance is not necessarily the top priority.

Source Code Walkthrough – CAllocateHierarchy

The CAllocateHierarchy class is derived from the abstract interface ID3DXAllocateHierarchy. This interface exposes four virtual functions which must be overridden in the derived class. These four functions are used as callbacks during the creation and destruction of the frame hierarchy.

The CActor temporarily instantiates an object of this type in the CActor::LoadActorFromX function and passes it to D3DXLoadMeshHierarchyFromX. This function will use the object's CreateFrame and CreateMeshContainer functions to allow the application to allocate the frames and mesh containers that will be attached to the hierarchy. The CActor instantiates an object of this type a second time in the CActor::Release function and passes it to the D3DXFrameDestroy function to unload the hierarchy from memory. This function uses the interface's DestroyFrame and DestroyMeshContainer methods to allow the application to release the frames and meshes in the hierarchy and perform any other clean up.

The derived class definition (see CActor.h) is shown below. It implements the four base class virtual functions and exposes a constructor that accepts a pointer to a CActor. The CActor pointer is stored in its only member variable so that the CreateMeshContainer method can access the registered callbacks of the actor during material processing.

```
class CAllocateHierarchy: public ID3DXAllocateHierarchy
{
public:
    // Constructors & Destructors for This Class
    CAllocateHierarchy( CActor * pActor ) : m_pActor(pActor) {}


    // Public Functions for This Class
    STDMETHOD(CreateFrame)          ( THIS_ LPCTSTR Name, LPD3DXFRAME *ppNewFrame);
    STDMETHOD(CreateMeshContainer)  ( THIS_ LPCTSTR Name, CONST D3DXMESHDATA *pMeshData,
                                      CONST D3DXMATERIAL *pMaterials,
                                      CONST D3DXEFFECTINSTANCE *pEffectInstances,
                                      DWORD NumMaterials, CONST DWORD *pAdjacency,
                                     LPD3DXSKININFO pSkinInfo,
                                     LPD3DXMESHCONTAINER *ppNewMeshContainer);

    STDMETHOD(DestroyFrame)         (THIS_ LPD3DXFRAME pFrameToFree);
    STDMETHOD(DestroyMeshContainer) (THIS_ LPD3DXMESHCONTAINER pMeshContainerBase);


    // Public Variables for This Class
    CActor  * m_pActor;         // Actor we are allocating for

};
```

CAllocateHierarchy::CreateFrame

The CreateFrame function is called by D3DXLoadMeshHierarchyFromX when a new frame data object is encountered in the X file. The name of the frame will be passed to this function as its first parameter along with the address of a D3DXFRAME pointer.

CreateFrame must allocate the frame structure and perform any initialization of its data members before assigning the passed frame pointer to the newly allocated frame structure. When the function returns, D3DX will attach this frame to the hierarchy in its correct position.

It is very useful that D3DX provides us with a means for creating the frame structure and does not allocate a D3DXFRAME structure on our behalf. This allows us to allocate our D3DXFRAME derived structure type, which contains an extra matrix (mtx_Combined) to store the absolute world matrix of the frame when the hierarchy is updated (via CActor::UpdateFrameMatrices). Because our D3DXFRAME_MATRIX structure is derived from D3DXFRAME, the variables that D3DX cares about (basically just the matrix, sibling, and child pointers) are still in the correct position. So the structure can be safely cast. From the perspective of D3DX, it is passed back a D3DXFRAME structure and it can treat it as such.

```
HRESULT CAllocateHierarchy::CreateFrame( LPCTSTR Name, LPD3DXFRAME *ppNewFrame )
{
    D3DXFRAME_MATRIX * pNewFrame = NULL;

    // Clear out the passed frame (it may not be NULL)
    *ppNewFrame = NULL;

    // Allocate a new frame
    pNewFrame = new D3DXFRAME_MATRIX;
    if ( !pNewFrame ) return E_OUTOFMEMORY;

    // Clear out the frame
    ZeroMemory( pNewFrame, sizeof(D3DXFRAME_MATRIX) );

    // Copy over, and default other values
    if ( Name ) pNewFrame->Name = _tcsdup( Name );
    D3DXMatrixIdentity( &pNewFrame->TransformationMatrix );

    // Pass this new pointer back out
    *ppNewFrame = (D3DXFRAME*)pNewFrame;

    // Success!!
    return D3D_OK;
}
```

The function starts by allocating our derived D3DXFRAME_MATRIX structure, initializing it to zero for safety. It also sets the frame's relative matrix to identity. This is also for safety, since this matrix should be overwritten by D3DX with the actual matrix values for this frame stored in the X file when the function returns.

The next thing we do is store the name of the frame (passed as the first parameter) in the frame structure so that we can traverse the actor's hierarchy and search for frames by name. This is very important because in our demo application we will need to search the actor’s hierarchy for the wheel and pivot frames so that we can attach CAnimation objects to them.

Notice that we cannot simply copy the passed string into the frame structure Name member. This string memory is owned by D3DX and will be released as soon as the function returns. This would leave us with an invalid string pointer in our frame. Therefore we must copy the contents of the string into a new string that we allocate. The _tcsdup function does this job for us. We just pass in the string we wish to copy and it will allocate a new string of the correct size and copy over the contents of the source string. We can then store the returned string in our frame.

Finally, we assign the D3DXFRAME pointer passed by D3DX to point to our newly allocated D3DXFRAME_MATRIX (making sure we cast it). On function return, D3DX will now have access to our frame and can use its sibling and child pointers to attach it to the hierarchy.

CAllocateHierarchy::CreateMeshContainer

The CreateMeshContainer function is called by D3DXLoadMeshHierarchyFromX whenever a mesh data object is encountered in the X file. This method must use the passed information to construct a D3DXMESHCONTAINER (or derived) structure and return it to D3DX.

We are using our own derived mesh container type (called D3DXMESHCONTAINER_DERIVED) which includes an additional CTriMesh pointer. With D3DX providing this mesh creation callback mechanism, we are afforded complete freedom to manage the mesh data and materials for the mesh in whatever way our application sees fit. This will allow us to store the passed mesh data in a CTriMesh and send the material data to the scene callbacks for storage. This function also provides us with a chance to check the creation flags of the mesh and its vertex format and clone it into a new mesh using the format we desire.

Because this function actually has quite a lot of work to do, we will cover it a section at a time. Luckily, the resource management section is almost an exact duplicate of the material handling code we saw in the previous workbook for CTriMesh. This code should be very familiar to you.

```
HRESULT CAllocateHierarchy::CreateMeshContainer(LPCTSTR Name,
                                                CONST D3DXMESHDATA*pMeshData,
                                                CONST D3DXMATERIAL *pMaterials,
                                                CONST D3DXEFFECTINSTANCE *pEffectInstances,
                                                DWORD NumMaterials,
                                                CONST DWORD *pAdjacency,
                                                LPD3DXSKININFO pSkinInfo,
                                                LPD3DXMESHCONTAINER *ppNewMeshContainer)
{

    ULONG                       AttribID, i;
    HRESULT                     hRet;
    LPD3DXMESH                  pMesh           = NULL;
    D3DXMESHCONTAINER_DERIVED *pMeshContainer = NULL;
    LPDIRECT3DDEVICE9           pDevice         = NULL;
    CTriMesh                   *pNewMesh        = NULL;
    MESH_ATTRIB_DATA           *pAttribData     = NULL;
    ULONG                      *pAttribRemap    = NULL;
    bool                       ManageAttribs  = false;
    bool                       RemapAttribs   = false;
    CALLBACK_FUNC              Callback;

    // We only support standard meshes (i.e. no patch or progressive meshes in this demo)
    if ( pMeshData->Type != D3DXMESHTYPE_MESH ) return E_FAIL;
```

We can see immediately that the function accepts quite a lengthy parameter list. As the first parameter we are passed the name of the mesh. This is the name assigned to the mesh data object in the X file, if one exists. As the second parameter we are passed a D3DXMESHDATA structure. You will recall from the textbook that this structure contains either an ID3DXMesh interface pointer, an ID3DXPMesh interface pointer, or an ID3DXPatchMesh interface pointer. It also contains a Type member variable describing which mesh type it is. The D3DXMESHDATA member is shown below:

```
typedef struct D3DXMESHDATA
{
    D3DXMESHDATATYPE Type;
    union
    {
      LPD3DXMESH           pMesh;
      LPD3DXPMESH         pPMesh;
      LPD3DXPATCHMESH pPatchMesh;
    }
} D3DXMESHDATA, *LPD3DXMESHDATA
```

D3DXMESHDATATYPE is an enumeration which can contain one of three values: D3DXMESHTYPE_MESH, D3DXMESHTYPE_PMESH or D3DXMESHTYPE_PATCHMESH.

In our demo code, we will only support the common case where the structure contains an ID3DXMesh pointer. If you wish to add support for loading progressive meshes from an X file, then you should be able to modify this function to handle both cases without any problem given our discussions in the last chapter.

Returning to the above code, we can see that the third parameter to this function is an array of D3DXMATERIAL structures containing the texture filename and material information for each subset in the passed mesh. The fourth parameter contains an array of effect instances. We will ignore any reference to this at the moment until we cover effect files in Module III. The fifth parameter describes the number of elements in the previous two arrays (the number of subsets in the mesh). This is followed by a pointer to the mesh adjacency information. The penultimate parameter is a pointer to a D3DXSKININFO structure. We will see this being used in Chapter 11 when we cover skinned meshes and skeletal animation. For now we can ignore this member. The final parameter is the address of a pointer to a D3DXMESHCONTAINER structure which our function will assign to the mesh container structure that we allocate and populate within this function. When the function returns, D3DX will use this pointer to connect the mesh container to its correct frame in the hierarchy.

In the next code section we see that the first thing we do is assign the ID3DXMesh interface stored in the passed D3DXMESHDATA member to a local member variable (pMesh) for easier access. Notice that although we are copying a pointer to an interface here, we are not incrementing the reference count. This is because we may not actually want to keep this mesh. We have to first check its creation parameters and its vertex format.

Currently we only support FVF style mesh creation, so if this mesh's FVF flags are zero, then we return from the function.

```
    // Extract the standard mesh from the structure
    pMesh = pMeshData->pMesh;

    // We require FVF compatible meshes only
    if ( pMesh->GetFVF() == 0 ) return E_FAIL;

    // Allocate a mesh container structure
    pMeshContainer = new D3DXMESHCONTAINER_DERIVED;
    if ( !pMeshContainer ) return E_OUTOFMEMORY;

    // Clear out the structure to begin with
    ZeroMemory( pMeshContainer, sizeof(D3DXMESHCONTAINER_DERIVED) );

    // Copy over the name
    if ( Name ) pMeshContainer->Name = _tcsdup( Name );
```

You can see above that we allocate our new mesh container. It will be a D3DXMESHCONTAINER_DERIVED structure which has a CTriMesh pointer that we will need to use. We initialize the structure to zero and then copy over the name of the mesh into the mesh container's Name member. As described in the CAllocateHierarchy::CreateFrame method, the memory containing the string is owned by D3DX, so we must make a copy because we cannot rely on its persistence after the function returns.

At this point we have an empty mesh container, so it is time to start filling it up with the passed data. Before we do that however, we know that we want our data to be stored in a CTriMesh object, so we will allocate one. To avoid confusion, the point should be made that the local variable pNewMesh is a pointer to a CTriMesh object. The local variable pMesh is a pointer to an ID3DXMesh interface.

```
    // Allocate a new CTriMesh
    pNewMesh = new CTriMesh;
    if ( !pNewMesh ) { hRet = E_OUTOFMEMORY; goto ErrorOut; }
```

We now have an empty CTriMesh in hand. In the previous workbook, it is at this point that we would register our callback functions, but that would be pointless in this code because the CTriMesh::LoadMeshFromX function will never be used. That is why the same callback code will need to be implemented by hand in this function.

The ID3DXMesh data has already been loaded and passed to us by D3DX along with the materials that it uses. We will see in a moment that we will attach this ID3DXMesh to the CTriMesh using the CTriMesh::Attach function. But before we do that, we want to make sure that the ID3DXMesh that we have been passed is using a data format our application is happy with. If not, we will need to clone it into the appropriate format.

The next section of code tests to see whether the FVF flags of the ID3DXMesh that D3DX has passed us contains vertex normals. If it does not, then this mesh is no good to us and we will clone a new mesh and add the normals ourselves. We also clone the mesh if the mesh creation options are not what we requested. This may seem a strange test to do; after all, the application will pass the desired mesh creation options into the CActor::LoadActorFromX function, which in turn passes the options on to the D3DXLoadMeshHierarchyFromX function. Therefore, the meshes should surely be created using the options we initially registered with the actor. Well, you would certainly think so. However, in our laboratory tests on various machines we found that the meshes created by D3DX did not always match the mesh creation options passed into D3DXLoadMeshHierarchyFromX. Sometimes it even placed the vertex or index buffer in system memory. So to be safe, we will retrieve the options from the passed mesh and compare them against the options registered with the actor (by the application), and if they are not a match, we will clone the mesh using the correct mesh creation flags.

```
    // If there are no normals, this demo requires them, so add them to the mesh's FVF
    if (!(pMesh->GetFVF()&D3DFVF_NORMAL) || (pMesh->GetOptions()!=m_pActor->GetOptions()) )
    {
        LPD3DXMESH pCloneMesh = NULL;

        // Retrieve the mesh's device (this adds a reference)
        pMesh->GetDevice( &pDevice );

        // Clone the mesh
        hRet = pMesh->CloneMeshFVF( m_pActor->GetOptions(),
                                    pMesh->GetFVF() | D3DFVF_NORMAL,
                                    pDevice,
                                    &pCloneMesh );

        if ( FAILED( hRet ) ) goto ErrorOut;

        // Note: we don't release the old mesh here, because we don't own it
        pMesh = pCloneMesh;

        // Compute the normals for the new mesh if there was no normal to begin with
        if ( !(pMesh->GetFVF() & D3DFVF_NORMAL) ) D3DXComputeNormals( pMesh, pAdjacency );

        // Release the device, we're done with it
        pDevice->Release();
        pDevice = NULL;

        // Attach our specified mesh to the new mesh (this addref's the chosen mesh)
        pNewMesh->Attach( pMesh );


    } // End if no vertex normal or wrong options
    else
    {
        // Simply attach our specified mesh to the new mesh (this addref's the chosen mesh)
        pNewMesh->Attach( pMesh );

    } // End if options ok
```

In the above code you can see that if the mesh creation options are incorrect or normals do not exist in the vertices then we clone a new mesh using the correct format. We never release the original mesh interface because we never incremented its reference count earlier. At this point, the original mesh (which we no longer need) has a reference count of 1 (i.e., D3DX’s claim on the interface). When the function returns and D3DX decrements the reference count of the original mesh, it will be unloaded from memory. Notice that after the clone operation, we reassign the local pMesh pointer to point at the cloned mesh interface instead.

If normals did not previously exist then we just added room for them. Since the normal data is not yet initialized, we use the D3DXComputeNormals function, passing in the adjacency information that we were passed by D3DX, to calculate the normals.

Next, we release the device interface (retrieved from the mesh) which we no longer need, and we call the CTriMesh::Attach method to attach the tri-mesh to our ID3DXMesh. We looked at the CTriMesh::Attach function in the previous workbook -- it assigns the CTriMesh internal ID3DXMesh pointer to point to the passed interface. The final section of the above code demonstrates that when the passed mesh is already in the correct format, we can simply attach it to the CTriMesh. The Attach method will automatically increment the reference count of the passed interface.

At this point we have a CTriMesh with a valid ID3DXMesh containing the mesh data from the X file. We now have to decide what to do with the material data passed into the function. If the actor is in managed mode, then we will wish to register the texture filenames for each subset with the texture callback (pre-registered with the actor) and store the texture pointers and materials in the mesh itself. If this is a non-managed mode actor, then we will need to perform attribute buffer remapping.

The first line of the following code tests to see if the actor has had its CALLBACK_ATTRIBUTEID function registered. If so, then this actor is in non-managed mode and the meshes will contain their own attribute lists. If the function pointer is NULL, then ManageAttribs will be set to true and the actor is in managed mode. When this is the case, our first task is to allocate the CTriMesh's attribute data array to the correct size -- we need to it to be large enough to contain an entry for each subset. We allocate this array using the CTriMesh::AddAttributeData function and pass in the number of subsets in the mesh. Once it is allocated we use CTriMesh::GetAttributeData to retrieve a pointer to the first element in the array. This pointer will be used to populate the array with texture pointers and material information.

```
    // Are we managing our own attributes ?
    ManageAttribs = (m_pActor->GetCallback
                    ( CActor::CALLBACK_ATTRIBUTEID ).pFunction == NULL);

    // Allocate the attribute data if this is a manager mesh
    if ( ManageAttribs == true && NumMaterials > 0 )
    {
        if ( pNewMesh->AddAttributeData( NumMaterials ) < -1 )
                                              { hRet = E_OUTOFMEMORY; goto ErrorOut; }
        pAttribData = pNewMesh->GetAttributeData();

    } // End if managing attributes
    else
    {
        // Allocate attribute remap array
        pAttribRemap = new ULONG[ NumMaterials ];
        if ( !pAttribRemap ) { hRet = E_OUTOFMEMORY; goto ErrorOut; }

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

    } // End if not managing attributes
```

The above code also shows the case for a non-managed mode mesh. In this case we know we will need to re-map the attribute buffer, so we allocate an array large enough to be used as a temporary re-mapping buffer. This was all covered in the CTriMesh class in the previous workbook, so this should not be new to you. We will use this temporary buffer to store the global attribute IDs for each subset returned by the scene's resource callback function.

We now loop through each material in the passed D3DXMATERIAL array and process them in accordance with the mode of the actor. The following code was described in its entirety in the previous chapter (only it was inside CTriMesh instead), so you should refer back if you need to refresh your memory. At the end of the loop, the attribute data array inside the CTriMesh will contain an element for each subset in the mesh. Each element will contain the texture, material and possible effect file used by the subset.

```
    // Loop through and process the attribute data
    for ( i = 0; i < NumMaterials; ++i )
    {
        if ( ManageAttribs == true )
        {
            // Store material
            pAttribData[i].Material = pMaterials[i].MatD3D;

            // Note : The X File specification contains no ambient material property
            pAttribData[i].Material.Ambient = D3DXCOLOR( 1.0f, 1.0f, 1.0f, 1.0f );

            // Request texture pointer via callback
            Callback = m_pActor->GetCallback( CActor::CALLBACK_TEXTURE );
            if ( Callback.pFunction )
            {
                COLLECTTEXTURE CollectTexture = (COLLECTTEXTURE)Callback.pFunction;
                pAttribData[i].Texture = CollectTexture( Callback.pContext,
                                                         pMaterials[i].pTextureFilename );

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

            } // End if callback available

            // Request effect pointer via callback
            Callback = m_pActor->GetCallback( CActor::CALLBACK_EFFECT );
            if ( Callback.pFunction )
            {
                COLLECTEFFECT CollectEffect = (COLLECTEFFECT)Callback.pFunction;
                pAttribData[i].Effect = CollectEffect( Callback.pContext,
                                                       pEffectInstances[i] );

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

            } // End if callback available

        } // End if attributes are managed
```

The next code block shows what happens when the actor is non-managed. Again, you can refer to the prior workbook for details about the operations encountered.

```
        else
        {
            // Request attribute ID via callback
            Callback = m_pActor->GetCallback( CActor::CALLBACK_ATTRIBUTEID );
            if ( Callback.pFunction )
            {
             COLLECTATTRIBUTEID CollectAttributeID =(COLLECTATTRIBUTEID)Callback.pFunction;
             AttribID = CollectAttributeID( Callback.pContext,
                                            pMaterials[i].pTextureFilename,
                                            &pMaterials[i].MatD3D, &pEffectInstances[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
```

At this point, a managed mesh will be completely set up, with its internal attribute information stored for self-contained rendering.

If the actor is in non-managed mode, then the current mesh still needs to have its attribute buffer re-mapped. So the next snippet of code will lock the attribute buffer of a non-managed mesh and use the temporary re-map array we just populated to change the attribute of each triangle to the new global attribute ID. This code is also not new to us -- the CTriMesh performs a very similar step when its CTriMesh::LoadMeshFromX function is used.

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

        // Lock the attribute buffer
        hRet = pMesh->LockAttributeBuffer( 0, &pAttributes );
        if ( FAILED(hRet) ) goto ErrorOut;

        // Loop through all faces
        for ( i = 0; i < 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
        pMesh->UnlockAttributeBuffer( );

    } // End if remap attributes

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

Our CTriMesh is now fully created, and the scene (or whatever object has registered the resource callbacks) has loaded and stored any textures and materials that the mesh uses.

We can now perform optimizations and cleaning using our CTriMesh methods. We will first perform a weld of the vertices to merge any duplicated vertices that may exist into a single vertex. This will allow us to reduce the vertex count in some cases. We then issue a call to CTriMesh::OptimizeInPlace to perform a compaction of the data (removing degenerate triangles for example), an attribute sort of the attribute buffer (for better subset rendering performance), and a vertex cache optimization to rearrange the mesh data for better vertex cache coherency.

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

With our mesh in top condition, we can now assign it to the mesh container that we allocated earlier in the function. Recall that we actually allocated an instance of our derived mesh container class, which contains a CTriMesh pointer in addition to the usual members of the base class. We store the CTriMesh pointer to our new mesh in the mesh container and store the CTriMesh's underlying ID3DXMesh interface in the mesh container's D3DXMESHDATA member. We do this final step so that D3DX helper functions (e.g., D3DXComputeBoundingSphere) will still be able to work with the mesh data stored in our hierarchy. Remember that any D3DX helper function that deals with meshes in a hierarchy will not be aware that the hierarchy now contains mesh data in the proprietary CTriMesh type added to the derived class. These functions expect the mesh data to be stored as an ID3DXMesh interface inside the D3DXMESHDATA member of the mesh container.

By fetching the CTriMesh's underlying ID3DXMesh interface and storing it in the D3DXMESHDATA member of the mesh container, we keep everybody happy. Our actor can access the methods of CTriMesh using the mesh container's CTriMesh pointer, and D3DX can traverse the hierarchy and access the raw mesh data via the ID3DXMesh interface stored in the expected place.

One such D3DX helper function that will need to access the mesh data is the D3DXSaveMeshHierarchyToFile method which we discussed earlier (see CActor::SaveActorToX). This function will traverse the hierarchy and write the frame hierarchy information and mesh container information out to the X file. But this function needs more than just access to the mesh data; it also needs to know about the materials and texture filenames used by the mesh so that they can be written to the file as well. This would seem to present a bit of a problem since the application manages the loading an storing of textures in data areas the actor might not have access to. The solution is actually quite simple -- it is in this function that we are first passed the D3DXMATERIAL array that contains the texture filenames and materials for each subset. We have seen how the scene callback functions are invoked by the actor to turn information in this array into real textures (instead of just filenames). Therefore, at this point, we already know the materials and texture filenames used by the mesh because D3DX passed them into this function to help us load the correct resources. So when this X file is saved, we simply need D3DX to get access to this array of D3DXMATERIAL's that it originally passed us during the loading process.

As we saw earlier, the D3DXMESHCONTAINER structure has members to store both a pointer to a D3DXMATERIAL array and the number of materials used by this array. This is exactly where the D3DXSaveMeshHierarchyToFile function expects the material and texture data for each mesh container to be stored. So all we have to do to make the save function work, is make a copy of the D3DXMATERIAL array that was passed into this function and store that in the mesh container too. We must make a copy of the material array, and not just assign the mesh container members to point directly at it, because the material array is owned by D3DX and will be destroyed as soon as this method returns.

The next section of code shows how to do everything just discussed. It stores the CTriMesh's underlying ID3DXMesh interface in the mesh container's D3DXMESHDATA member, and copies the passed D3DXMATERIAL array into a new array. A pointer to this array and the number of elements in it are also stored in the mesh container's member variables, so that they can be accessed by D3DXSaveMeshHierarchyToFile.

```
    // Store our mesh in the container
    pMeshContainer->pMesh = pNewMesh;
    pMeshContainer->D3DXMESHDATA.pMesh=pNewMesh->GetMesh();
    pMeshContainer->NumMaterials   = NumMaterials;

    // Copy over material data only if in managed mode (i.e. we can save)
    if ( NumMaterials > 0 && ManageAttribs == true )
    {
        // Allocate material array
        pMeshContainer->pMaterials = new D3DXMATERIAL[ NumMaterials ];

        // Loop through and copy
        for ( i = 0; i < NumMaterials; ++i )
        {
            pMeshContainer->pMaterials[i].MatD3D = pMaterials[i].MatD3D;
            pMeshContainer->pMaterials[i].pTextureFilename =
                      _tcsdup( pMaterials[i].pTextureFilename );

        } // Next Material

    } // End if any materials to copy
```

Now the new mesh container contains all of the information we need it to store and it is ready to be attached to the hierarchy. As our final step, we assign the D3DXMESHCONTAINER pointer passed into this function by D3DX to point to the newly allocated container. On function return, D3DX will now have access to our mesh container and will attach it to the correct frame in the hierarchy.

```
    // Store this new mesh container pointer
    *ppNewMeshContainer = (D3DXMESHCONTAINER*)pMeshContainer;

    // Success!!
    return D3D_OK;

ErrorOut:
    // If we drop here, something failed
    DestroyMeshContainer( pMeshContainer );

    if ( pDevice      ) pDevice->Release();
    if ( pAttribRemap ) delete []pAttribRemap;
    if ( pNewMesh     ) delete pNewMesh;

    // Failed....
    return hRet;
}
```

The bottom section of the code performs cleanup if any errors happen. If anything goes wrong throughout this function, program flow is diverted to this section of code and everything is correctly cleaned up.

> **NOTE**: Notice that we only make a copy of the material array and store it in the mesh container if the mesh is in managed mode. In non-managed mode, the attribute IDs of each subset may have been remapped to different values. If we try to save an actor containing a non-managed mesh, the attribute IDs would be wrong and incorrect texture and material data would be output to the file for each subset. Because of this, the CActor::SaveActorToX function only works when the actor is in managed mode. This is not a major limitation, because you will generally want to save actor data when you are using the actor as a tool (not in your actual game). In this case, using the actor in managed mode is probably perfectly acceptable from a performance perspective.

CAllocateHierarchy::DestroyFrame

The CAllocateHierarchy::DestroyFrame function is called by the D3DXFrameDestroy function when it removes the hierarchy from memory. For each frame visited by that function, this function is passed the frame that needs to be deleted. D3DX cannot de-allocate the frame for us since it has no idea how we allocated the frame memory in the CAllocateHierarchy::CreateFrame method. It also has no means of knowing what other cleanup might have to be performed.

```
HRESULT CAllocateHierarchy::DestroyFrame( LPD3DXFRAME pFrameToFree )
{
    D3DXFRAME_MATRIX * pMtxFrame = (D3DXFRAME_MATRIX*)pFrameToFree;

    // Validate Parameters
    if ( !pMtxFrame ) return D3D_OK;

    // Release data
    if ( pMtxFrame->Name ) free( pMtxFrame->Name ); // '_tcsdup' allocated.
    delete pMtxFrame;                               // 'new' allocated.

    // Success!!
    return D3D_OK;
}
```

If the passed frame pointer is valid, the memory allocated for the frame name string is deleted. Notice that we use the C free function to delete this memory because _tcsdup was used to copy the string (_tcsdup uses the C memory allocation function alloc to allocate the memory). We must always make sure we match the memory allocation functions with their proper de-allocation functions if we are to avoid memory leaks.

Once the string has been deleted, we delete the frame structure itself. Notice how we use the delete operator for this because we allocated the frame memory in CreateFrame using the C++ new operator.

CAllocateHierarchy::DestroyMeshContainer

This is the final interface function that needs to be implemented, and it is responsible for releasing the memory for a mesh container. When our actor de-allocates its frame hierarchy (using D3DXFrameDestroy), this method will be called for every mesh container. A pointer to the mesh container to be deleted is the only parameter to this function.

We begin by releasing the D3DXMATERIAL array. Recall that this array was allocated in the CreateMeshContainer method to make a copy of the passed D3DXMATERIAL array so that the D3DXSaveMeshHierarchyToFile method will be able to access the materials and textures used by the mesh during the hierarchy saving procedure. If the mesh is in non-managed mode then this array should not exist, and the mesh container's pMaterials pointer should be NULL.

```
HRESULT CAllocateHierarchy::DestroyMeshContainer( LPD3DXMESHCONTAINER
                                                       pContainerToFree )
{
    D3DXMESHCONTAINER_DERIVED * pContainer = (D3DXMESHCONTAINER_DERIVED*)pContainerToFree;

    // Validate Parameters
    if ( !pContainer ) return D3D_OK;

    // Release material data ( used for saving X file )
    if ( pContainer->pMaterials )
    {
        for ( i = 0; i < pContainer->NumMaterials; ++i )
        {
            // Release the string data
            if ( pContainer->pMaterials[i].pTextureFilename )
                free( pContainer->pMaterials[i].pTextureFilename );

        } // Next Material

        // Destroy the array
        delete []pContainer->pMaterials;
        pContainer->pMaterials = NULL;

    } // End if any material data
```

Now we release the rest of the data allocated by the mesh container. First, you will recall that we stored a copy of the CTriMesh's underlying ID3DXMesh interface in the container's D3DXMESHDATA member. We must now release this interface. Second, we must also free the memory that was allocated to store the name of the mesh. Since this memory was allocated using _tcsdup in CreateMeshContainer, we must release it using the C free function. Finally, we delete the CTriMesh object stored in the mesh container's pMesh member.

```

    // Release data
    if ( pContainer->MeshData.pMesh ) pContainer->MeshData.pMesh->Release();
    if ( pContainer->Name ) free( pContainer->Name );   // '_tcsdup' allocated.
    if ( pContainer->pMesh ) delete pContainer->pMesh;
    delete pContainer;

    // Success!!
    return D3D_OK;
}
```

We have now discussed the CAllocateHierarchy class and the CActor class in their entirety. You should be fairly comfortable with your understanding of how they work together to construct the entire frame hierarchy from an X file, taking care of frame and mesh asset allocation in the process.

Source Code Walkthrough – CAnimation

The CAnimation class is implemented in this demo to demonstrate a simple means for animating frame hierarchies. This class is not one that we will find ourselves using moving forward, but understanding its operations and the way it interacts with an actor’s hierarchy should prove helpful in preparing us for the following chapter on animation.

The CAnimation class is not used by the CActor or CAllocateHierarchy classes. Objects of this type are instantiated by the scene and attached to frames in an actor’s hierarchy. The scene will then use an object of this type to apply rotations via the relative matrix of the frame.

The scene will use the CActor::FindFrameByName function to search for a frame that it wishes to animate (e.g., Wheel_FL) and that function will return a pointer to the matched frame. The scene will then store a pointer to that matrix in a CAnimation object (using CAnimation::Attach). During each iteration of the game loop, the scene can then call the CAnimation::Rotate family of functions to apply rotations to the attached frame.

In our application, the scene instantiates six CAnimation objects: four will be attached to the wheels of the cars and two will be attached to the pivot frames to perform the steering animation.

The class definition (see CAnimation.h) is shown below.

```
class CAnimation
{
public:
    // Constructors & Destructors for This Class
             CAnimation();
    virtual ~CAnimation();

    // Public Functions for This Class
    void            Attach      ( LPD3DXMATRIX pMatrix, LPCTSTR strName = NULL );
    LPCTSTR         GetName     ( ) const { return m_strName; }
    LPD3DXMATRIX    GetMatrix   ( ) const { return m_pMatrix; }

    void            RotationX   ( float fRadAngle, bool bLocalAxis = true );
    void            RotationY   ( float fRadAngle, bool bLocalAxis = true );
    void            RotationZ   ( float fRadAngle, bool bLocalAxis = true );

private:
    // Private Variables for This Class
    LPD3DXMATRIX    m_pMatrix;      // Attached interpolator matrix
    LPTSTR          m_strName;      // Name (if provided) of the frame
};
```

The class only has two member variables:

LPD3DXMATRIX m_pMatrix
This member is set via the CAnimation::Attach method. It will point to the relative matrix of a frame in the hierarchy. We could use this object type without frame hierarchies, since it can serve as a generic rotation matrix generator/updater. In our demo application however, any calls to RotationX, RotationY, or RotationZ functions will apply a rotation to a frame in our automobile hierarchy.

LPSTR m_strName
This member is also set via the CAnimation::Attach method. It is the name of the animation as well as the name of the frame in the hierarchy that will have its relative matrix updated by this object.

CAnimation::CAnimation()

The constructor simply initializes the member variables to NULL.

```
CAnimation::CAnimation()
{
    // Clear any required variables
    m_pMatrix   = NULL;
    m_strName   = NULL;
}
```

CAnimation::~CAnimation()

The destructor has to take care of releasing the string memory. This memory is allocated in the Attach method using _tcsdup, so a free call is required.

```
CAnimation::~CAnimation()
{
    // Release any memory
    if ( m_strName ) free( m_strName ); // '_tcsdup' allocated.

    // Clear Variables
    m_pMatrix   = NULL;
    m_strName   = NULL;
}
```

CAnimation::Attach

This function is called by the scene to attach a matrix and frame name to the object. The passed matrix pointer is copied into the member variable first. If the string containing the name is not NULL, then it already contains a name (perhaps the object was used previously to animate another frame), so we release its memory prior to copying the passed string.

```
void CAnimation::Attach( LPD3DXMATRIX pMatrix, LPCTSTR strName /* = NULL */ )
{
    // Attach us to the matrix specified
    m_pMatrix = pMatrix;

    // Release the old name
    if ( m_strName ) free( m_strName );
    m_strName = NULL;

    // Duplicate new name if provided
    if ( strName ) m_strName = _tcsdup( strName );
}
```

CAnimation::RotationX

The application animates the attached frame via its matrix using one of three rotation functions: RotationX, RotationY or RotationZ. As all three functions are identical (with the exception that they build a rotation around a different axis), we will only show the code to the RotationX function here. Check the sourse code if you would like to see implementations for the other two.

```
void CAnimation::RotationX( float fRadAngle, bool bLocalAxis /* = true */ )
{
    D3DXMATRIX mtxRotate;

    // Validate Prerequisites
    if ( !m_pMatrix ) return;

    // Generate rotation matrix
    D3DXMatrixRotationX( &mtxRotate, fRadAngle );

    // Concatenate the two matrices
    if ( bLocalAxis )
        D3DXMatrixMultiply( m_pMatrix, &mtxRotate, m_pMatrix );
    else
        D3DXMatrixMultiply( m_pMatrix, m_pMatrix, &mtxRotate );
}
```

This function is passed an angle (in radians) describing the angle we wish to rotate the frame by. The function also accepts a Boolean parameter called bLocalAxis which allows us to toggle whether we would like the frame to be rotated about its own coordinate system axis (usually the case) or whether we would like the rotation to apply around the parent frame’s axis. As the frame is relative to its parent frame in the hierarchy, this is a simple case of switching the matrix multiplication order. We will apply local rotations to our wheel frames so that they rotate around their own origins and axes.

The function creates an X axis rotation matrix using the D3DXMatrixRotationX function and then multiplies the returned matrix with the attached frame matrix. The Boolean is checked to see if local or parent relative axis rotation should be applied and multiplication order is adjusted accordingly.

We have now covered all the new objects that have been introduced in this lab project. In the next section, we will look at how they are used collectively by the scene to load a multi-mesh X file and animate the desired frames.

Source Code Walkthrough – CScene Modifications

The changes to the scene class are quite small, but they are significant. For starters, any X files loaded via CScene::LoadSceneFromX or CScene::LoadSceneFromIWF will now be loaded into CActor objects instead of directly into CTriMesh objects. In the case of an IWF file, the external references in the IWF file will be created as actors. Even if an X file contains only a single mesh and no frame hierarchy, the CActor class can still be used in the same way. The D3DXLoadMeshHierarchyFromX function will always create a root frame in such instances and the mesh will be attached as an immediate child.

The scene object will also have its Render function updated to handle actors instead of meshes. Of course, the scene will also need to create the CAnimation objects at load time and attach them to the relevant frames in the hierarchy. The CScene::AnimateObjects function will loop through its four wheel CAnimation objects and use their member functions to apply rotations to the wheels of the car.

The updated CScene class will include a pointer to an array of CActors. This array will contain all CActors that have been loaded. While this is the only new member we have to add, we will also list the member variables that were added in the previous workbook to get a better understanding of how everything fits together. The significant member variables of CScene are shown below:

```
    // Private Variables for This Class
    TEXTURE_ITEM       *m_pTextureList [MAX_TEXTURES];  // Array of texture pointers
    D3DMATERIAL9        m_pMaterialList[MAX_MATERIALS]; // Array of material structures
    CAnimation          m_pAnimList    [MAX_ANIMATIONS];// Array of animations

    ULONG               m_nTextureCount;                // Number of textures stored
    ULONG               m_nMaterialCount;               // Number of materials stored

    CObject           **m_pObject;                      // Array of objects storing meshes
    ULONG               m_nObjectCount;                 // Number of objects

    CTriMesh          **m_pMesh;                        // Array of loaded scene meshes
    ULONG               m_nMeshCount;                   // Number of meshes
    CActor            **m_pActor;                       // Array of loaded scene actors
    ULONG               m_nActorCount;                  // Number of actors
    ATTRIBUTE_ITEM      m_pAttribCombo[MAX_ATTRIBUTES]; // Table of attribute combinations
    ULONG               m_nAttribCount;                 // Number of attributes
};
```

Let us just briefly remind ourselves of the purpose of these member variables and the data they will contain when the scene is loaded:

TEXTURE_ITEM *m_pTextureList [MAX_TEXTURES]
ULONG m_nTextureCount
This array (added in the previous workbook) will contain all textures used by the scene. Each TEXTURE_ITEM structure contains a texture pointer and a texture filename. When X files are being loaded (either via CTriMesh::LoadMeshFromX or CActor::LoadActorFromX), all meshes will have a scene texture callback function registered. The texture data (filename) from the X file will be passed to the callback function and, if the texture does not already exist, it will be loaded and added to this array. If an IWF scene is being loaded, then this array will contain the textures used by the external references (X files) and the textures stored in the IWF file that are applied to the internal meshes (non X file meshes). So this array will contain all textures used by all CActors and CTriMeshes in the scene.

D3DMATERIAL9 m_pMaterialList [MAX_MATERIALS]
ULONG m_nMaterialCount
This array is analogous to the texture array and was also added in the previous workbook. It contains all the materials used by all meshes and actors in the scene. This includes all materials loaded from X files

as well as materials used by IWF internal meshes. This array will only contain the materials used by meshes and actors created in non-managed mode. As we know, meshes in managed mode will store the material information internally in their attribute data array. This is true whether they are standalone (in their own CObject) or part of an actor's hierarchy.

ATTRIBUTE_ITEM m_pAttribCombo[MAX_ATTRIBUTES]
ULONG   m_nAttribCount
This array was also introduced in the previous chapter and describes an array of texture/material combinations used by all non-managed mode meshes. It is an index into this array that is returned from the scene attribute callback, used to re-map the attribute buffer of non-managed mode meshes. Each element in this array contains the texture and material for a single scene level subset.

CObject   **m_pObject
ULONG   m_nObjectCount
As with all previous implementations of the CScene class that we have implemented, this array contains all the objects that need to be rendered by the scene. Now a CObject may contain a pointer to a CActor instead of a CTriMesh. If the data is loaded from an IWF file for example, this array can contain a mixture of CObject types -- some may contain standalone CTriMeshes (i.e., the meshes stored internally in the IWF file) while others contain CActors that were specified in the IWF file as external references. Each object contains a world matrix that represents the position and orientation of the object in the scene. It is this object array that the scene will loop through and render during each cycle of the game loop.

CTriMesh   **m_pMesh
ULONG   m_nMeshCount
This array will contain any standalone CTriMeshes that have been loaded and stored in CObjects. If the scene is loaded from an X file then this array will be empty. If the scene has been loaded from an IWF file then it will contain CTriMeshes for the meshes in the file.

CActor   **m_pActor
ULONG   m_nActorCount
This array will contain pointers to all CActor objects that have been loaded by the scene and stored in CObjects. These members are the only new ones introduced in this demo application. A CActor will be created by each call to the CScene::LoadSceneFromX file. Multiple actors can be created and added to this array if the CScene::LoadSceneFromIWF file is used. An actor is created for each external mesh reference stored in the file.

Now that we know how the various member variables are connected, let us cover the CScene methods that are now or are significantly modified by the introduction of CActor. The discussion of each function below will have the label 'Modified' or 'New' indicating whether the method previously existed and has just been updated or whether it is a brand new addition to the CScene class. Some methods such as the constructors and destructors have also been modified to now initialize and destroy the new CActor array. We will not show the code to these functions as they contain routine steps that we have seen dozens of times before. Check the source code for more details.

CScene::LoadSceneFromX - Modified

As we have come to expect, this method is called by the CGameApp::BuildObjects function to load X files. This function is only called if we wish a single X file to be loaded. Sometimes we may want to load IWF scenes containing multiple meshes and external X file references, and on these occasions we will use the CScene::LoadSceneFromIWF call instead. In this lab project, we are loading a single X file containing an automobile, and as such, this function will be called to start the scene construction process.

This function is not very large and is mostly unchanged from the previous version. We will look at it in two sections.

```
bool CScene::LoadSceneFromX( TCHAR * strFileName )
{
    HRESULT hRet;

    // Validate Parameters
    if (!strFileName) return false;

    // 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');

    CActor * pNewActor = new CActor;
    if (!pNewActor) return false;

    // Load in the specified X file
    pNewActor->RegisterCallback( CActor::CALLBACK_ATTRIBUTEID, CollectAttributeID, this );
    hRet = pNewActor->LoadActorFromX( strFileName, D3DXMESH_MANAGED, m_pD3DDevice );
    if ( FAILED(hRet) ) { delete pNewActor; return false; }
```

This function is passed the filename (complete with path) describing the name of the X file that is to be loaded. This is also the filename that will be passed and stored by the CActor that is ultimately created. The first thing the function does is make a copy of the complete filename and strip off the name of the file so that we are just left with the file path. It can often be useful for the scene object to know which folder houses its assets, so that it can be used to load any textures that the X file references.

After the path of the X file has been stored in the scene, we allocate a new CActor object. We know from our earlier discussion of CActor that it will initially contain no data and its root pointer will be set to NULL. The code then registers the CScene::CollectAttributeID static method as the CALLBACK_ATTRIBUTEID callback for the actor. This places the actor in non-managed mode, so all textures and materials contained in the X file we are about to load will be stored at the scene level. We know that this function will return global attribute IDs which are used to re-map the attribute buffers of all non-managed mode meshes.

With our non-managed mode callback registered with the actor, it is time to instruct the actor to load the X file. We call the CActor::LoadActorFromX function with the filename passed into the function and the D3DXMESH_MANAGED mesh option flag. This option flag will be stored in the actor and used to make sure that any CTriMeshes we create have their vertex and index buffers allocated in the managed resource pool. We also pass in a pointer to the Direct3D device that will own the mesh resources.

When this function returns, our CActor will have been populated with the frame/mesh hierarchy contained in the X file. Any textures and materials used by the meshes in the hierarchy will have been loaded and stored in the scene's texture, material and attribute arrays (by the callback).

Our CActor object is still a standalone object at this point, so we must add its pointer to the CScene's actor array. To do this we call a new CScene utility function called AddObject to resize the current CActor array and make room at the end for another pointer. (This function works identically to the AddMesh and AddObject utility functions from previous lessons). We then copy our new CActor pointer into the CActor array, at the end of the list. This is the list that the scene will traverse to destroy all currently loaded actors when the scene is cleaning itself up during its own destruction.

```
    // Store this new actor
    if ( AddActor( ) < 0 ) { delete pNewActor; return false; }
    m_pActor[ m_nActorCount - 1 ] = pNewActor;

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

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

As shown in the above code, we then allocate a new CObject. Our scene will work with CObjects so that both meshes and actors can be managed in a consistent manner. Most importantly, the CObject allows us to pair each CTriMesh or CActor with a world matrix. So when we create the CObject, we pass the CActor pointer into its constructor for storage. We then use the CScene::AddObject method to make room in the CScene's CObject array for another CObject pointer. Finally, the new CObject is added to the end of the array.

In this demo we do not use the light group mechanism from Chapter Five so that we can keep the code as focused as possible on the goals at hand. The X file will also contain no lighting information, so we will need to set up some default lights for viewing purposes. The next section of code handles this by creating four direction lights and storing them in the first four elements of the scene's D3DLIGHT9 array. These lights will be set on the device later and used by the pipeline to light our polygons. The remainder of the code is shown below:

```
    // 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( 0.2f, 0.2f, 0.2f, 0.0f );
    m_pLightList[1].Specular = D3DXCOLOR( 0.3f, 0.3f, 0.3f, 0.0f );
    m_pLightList[1].Direction = D3DXVECTOR3( -0.819f, -0.573f, -0.0f );

    m_pLightList[2].Type = D3DLIGHT_DIRECTIONAL;
    m_pLightList[2].Diffuse = D3DXCOLOR( 0.2f, 0.2f, 0.2f, 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( 0.1f, 0.05f, 0.05f, 0.0f );
    m_pLightList[3].Specular = D3DXCOLOR( 0.6f, 0.6f, 0.6f, 0.0f );
    m_pLightList[3].Direction = D3DXVECTOR3( 0.0f, -0.707107f, 0.707107f );

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

    // Build Animation 'Interpolators'
    BuildAnimations();

    // Success!
    return true;
}
```

Notice at the very bottom of the above function, we make a call to the new BuildAnimations method. This function is responsible for allocating six CAnimation objects and attaching them to the wheel and pivot frames in the hierarchy. We will look at that function in just a short while.

CScene::AddActor - New

This function is included here for completeness, but you have seen code like this many times before in our studies together. AddActor resizes the actor array when one or more CActors have to be added to the scene. The function takes a Count parameter (which defaults to 1) which can allow the function to add space for multiple CActor pointers at the end of the array.

Recall that the process of resizing is simply a case of allocating a large enough array and copying over the data from the old array to the new array. The old array can then be deleted and the original pointer can then be assigned to point at the new array. This function returns the index of the first newly allocated element in the array so that the caller can start adding the new information into the array from that point.

```
long CScene::AddActor( ULONG Count /* = 1 */ )
{
    CActor ** pActorBuffer = NULL;

    // Allocate new resized array
    if (!( pActorBuffer = new CActor*[ m_nActorCount + Count ] )) return -1;

    // Existing Data?
    if ( m_pActor )
    {
        // Copy old data into new buffer
        memcpy( pActorBuffer, m_pActor, m_nActorCount * sizeof(CActor*) );

        // Release old buffer
        delete []m_pActor;
    } // End if

    // Store pointer for new buffer
    m_pActor = pActorBuffer;

    // Clear the new items
    ZeroMemory( &m_pActor[m_nActorCount], Count * sizeof(CActor*) );

    // Increase Actor Count
    m_nActorCount += Count;

    // Return first Actor
    return m_nActorCount - Count;
}
```

CScene::BuildAnimations

The CScene::BuildAnimations function is called at the bottom of both the LoadSceneFromX and the LoadSceneFromIWF functions. Its purpose is to provide the scene class with an opportunity to attach CAnimation objects to various frames stored in the actors. In our demo, this function creates six CAnimations objects and attaches them to the four wheel frames and the two pivot frames.

At the start of the function, we declare an array with six string elements. Each string in this array contains the name of the frame we wish to animate. These are the names of the frame data objects in the X file and the thus, the names of the D3DXFRAME's in the hierarchy of actors we will be searching for.

> **NOTE:** This function will not be a permanent member of our CScene class. It has been added to this lab project only to introduce you to the concept of frame animation, before we cover it in proper detail in the next chapter. Therefore, this function is almost completely hard-coded for the model we are going to use and as such, will not be very useful to you outside of this single project.

In the outer loop, we loop six times. Each iteration of this loop is searching through all frames to find a frame with the given name. For example, in the first iteration of this loop, we are searching all objects in the scene which contain actors with a frame named 'Wheel_FL'. If we find one, we attach the frame's matrix to the CAnimation object (using the CAnimation::Attach function) and we are done with this frame name. Then we continue on to the next iteration of the outer loop.

```
void CScene::BuildAnimations( )
{
    ULONG   i, j;
    LPCTSTR FrameNames[] = { _T("Wheel_FL"), _T("Wheel_FR"),
                             _T("Wheel_BL"), _T("Wheel_BR"),
                             _T("Wheel_Pivot_FL"), _T("Wheel_Pivot_FR") };

    // Search for each of the four wheels
    for ( i = 0; i < 6; ++i )
    {
        // Loop through each object and search
        for ( j = 0; j < m_nObjectCount; ++j )
        {
            CObject * pObject = m_pObject[j];
            if ( !pObject ) continue;
            // Any actor ?
            CActor * pActor = pObject->m_pActor;
            if ( !pActor ) continue;

            // Any frame by this name ?
            LPD3DXFRAME pFrame = pActor->GetFrameByName( FrameNames[i] );
            if ( pFrame )
            {
                // Attach the animation to this object, and skip to the next wheel
                m_pAnimList[i].Attach( &pFrame->TransformationMatrix, FrameNames[i] );
                break;

            } // End if found frame
            else
            {
                // Clear this interpolator out
                m_pAnimList[i].Attach( NULL );

            } // End if no frame found

        } // Next Object

    } // Next Wheel

    // If wheel yaw has already been applied, set initial rotation to this
    m_pAnimList[ Wheel_Pivot_FL ].RotationY( D3DXToRadian( fWheelYaw ) );
    m_pAnimList[ Wheel_Pivot_FR ].RotationY( D3DXToRadian( fWheelYaw ) );
}
```

If a frame cannot be found, we set the corresponding CAnimation object in the scene’s array such that it is not attached to any frame. We do this by passing NULL into the Attach method.

Finally, at the bottom of this function we set the initial steering angle that has been assigned to the wheels. Feel free to set the fWheelYaw variable to an initial angle other than zero if you want the automobile wheels rotated by some amount. Notice that, as discussed earlier, we apply a Y axis rotation to the pivot frames so that both of the car’s front wheels are rotated around the car’s up vector. Because the CAnimation objects are already attached to their frames at this point, we can simply use their RotationY methods (for the CAnimation objects attached to the pivots) to perform this task.

One thing of interest in the above function is that we are accessing to the two pivot animation objects using Wheel_Pivot_FL and Wheel_Pivot_FR as array indices. These values, as well as the indices for the other frames, are defined in the scene module’s namespace, as shown below:

```
namespace
{
    const UCHAR AuthorID[5]        = { 'G', 'I', 'L', 'E', 'S' };
    const UCHAR Wheel_FL           = 0;
    const UCHAR Wheel_FR           = 1;
    const UCHAR Wheel_BL           = 2;
    const UCHAR Wheel_BR           = 3;
    const UCHAR Wheel_Pivot_FL     = 4;
    const UCHAR Wheel_Pivot_FR     = 5;
    float       fWheelYaw          = 0.0f;
    float       fWheelYawVelocity  = 1000.0f;
};
```

Until now, only the AuthorID has existed in the namespace -- it was used by the IWF loading function to identify GILES™ custom chunks. Now we have added six Wheel_ members which describe the indices of the CAnimation objects in the scene that use the specified frames. We also have three new members at the bottom. The variable fWheelYaw will contain the current angle at which the pivot frames are rotated (i.e., the angle at which the front wheels are rotated). We saw this variable used in the previous code segment to set the rotation of the wheels. The variable fWheelYawVelocity contains the speed at which the front wheels will rotate left and right. We will see this variable used in a moment, when we examine the CScene::ProcessInput function.

CScene::AnimateObjects - Modified

This function is called by CGameApp::FrameAdvance for each cycle of the main game loop. This function has been a part of this class from the very start and provides the scene a chance to update its object during each frame.

In this version of the function, we rotate the four wheel frames via their matrices so that the car's wheels are continuously revolving around their local X axes. This function rotates the wheels at two revolutions per second (i.e., 720 degrees per second). We calculate the rotation angle for a given call by scaling 360 degrees by the number of revolutions (2) and then multiplying this by the elapsed time since the last frame. This will usually be fractions of a second, scaling the rotation angle for any particular update to only a few degrees. We pass the result of this calculation into the D3DXToRadian function so that we have the rotation angle in radians (required by the CAnimation::Rotation functions).

```
void CScene::AnimateObjects( CTimer & Timer )
{
    float fRevsPerSecond = 2.0f, fRadAngle;

    // Generate wheel frame rotation angle.
    fRadAngle = D3DXToRadian( (360.0f * fRevsPerSecond) * Timer.GetTimeElapsed() );

    // Rotate the wheels
    m_pAnimList[ Wheel_FL ].RotationX( fRadAngle );
    m_pAnimList[ Wheel_FR ].RotationX( fRadAngle );
    m_pAnimList[ Wheel_BL ].RotationX( fRadAngle );
    m_pAnimList[ Wheel_BR ].RotationX( fRadAngle );

}
```

Notice that because we have set up array index variables in the module namespace, we can quickly index into the CScene::m_pAnimList array and call the RotationX function for the correct CAnimation objects.

CScene::ProcessInput – Modified

CScene::ProcessInput is called by CGameApp::ProcessInput, which itself is called by CGameApp::FrameAdvance. The actual trapping of key data and subsequent camera updates are found in the CGameApp:ProcessInput function, as we saw in all previous lessons. When it calls this function, it passes in an array of all the keys with their up/down state. In this function we are testing for the comma key and the period key, which allow the user to rotate the wheels left or right respectively.

The first thing this function does is check the VK_COMMA and VK_DOT indices of the key array. If either has its top four bits set, then they key is depressed and rotations need to be applied. To provide a smooth deceleration of the wheel when the user lets go of the key, we use the fWheelYawVelocity variable to store the turn velocity. We can think of this value as the number of degrees per second that we wish the wheels to turn left or right. Every time the key is down during this function call, another 400 degrees per second is added to the turn velocity. The velocity can be either positive or negative for left and right steering. If no keys are pressed, then 400 degrees per second is subtracted from the velocity each time the function is called. This allows the wheel rotations a more realistic ramping up/down when the keys are pressed and released.

```
void CScene::ProcessInput( UCHAR pKeyBuffer[], CTimer & Timer )
{
    float fAngle = 0.0f;

    // Apply rotations ?
    if ( pKeyBuffer[ VK_COMMA ] & 0xF0 )
        fWheelYawVelocity -= 400.0f * Timer.GetTimeElapsed();
    else if ( pKeyBuffer[ VK_DOT ] & 0xF0 )
        fWheelYawVelocity += 400.0f * Timer.GetTimeElapsed();
    else
    {
        // Decelerate the yaw
        if ( fWheelYawVelocity > 0 )
            fWheelYawVelocity -= min(fWheelYawVelocity, 400.0f * Timer.GetTimeElapsed() );
        else
            fWheelYawVelocity -= max(fWheelYawVelocity, -400.0f * Timer.GetTimeElapsed());
    } // End if declerate
```

At this point the yaw velocity is scaled by the elapsed time to give us the actual number of degrees we need to rotate the wheels in this frame. Remember, the rotations are being added incrementally. The fWheelYaw member contains the current turn angle of the wheels -- it has the incremental turn angles added to it each time the function is called. We clamp the turn angle to 45 degrees, so if the yaw angle exceeds this amount with the newly calculated incremental angle added to it, we set the yaw angle to –45 or +45 and recalculate the incremental angle to stay within these bounds.

Once the incremental angle has been calculated and clamped, we pass it into the CAnimation::RotationY method of the animation objects attached to both pivots. Since the pivots are defined in the root's frame of reference, a rotation around the pivot Y axis rotates the wheel about the automobile's up vector.

```
    // Get the angle increase for this frame
    fAngle = fWheelYawVelocity * Timer.GetTimeElapsed();

    // Anything applicable
    if ( fAngle != 0.0f )
    {
        // Apply to total yaw and clamp where appropriate
        fWheelYaw += fAngle;
        if ( fWheelYaw >  45.0f )
           { fAngle -= (fWheelYaw - 45.0f); fWheelYaw = 45.0f; fWheelYawVelocity = 0.0f; }
        if ( fWheelYaw < -45.0f )
           { fAngle -= (fWheelYaw + 45.0f); fWheelYaw = -45.0f; fWheelYawVelocity = 0.0f; }

        m_pAnimList[ Wheel_Pivot_FL ].RotationY( D3DXToRadian( fAngle ) );
        m_pAnimList[ Wheel_Pivot_FR ].RotationY( D3DXToRadian( fAngle ) );

    } // End if any rotation
}
```

If you are a little rusty with this kind of velocity scaling, refer back to Chapter Four where we used a similar technique for our camera movement system.

CScene::Render

The render function is responsible for setting the states and rendering the subsets of all CObjects in the scene. The CObjects can contain both actors and/or meshes, so we must make sure that we take this into account. The reason the scene is responsible for state setting in this application is that we created our actors/meshes in non-managed mode. Most of the following code should be familiar to:

```
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 );

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

    } // Next Light
```

The function first checks that the device is valid and sets the D3DRS_NORMALIZENORMALS render state to true. This will make sure that even if a mesh or actor world matrix includes a scaling characteristic that would ordinarily scale the normals, no problems with the lighting pipeline will be encountered.

We then render call the RenderSkyBox function to render the skybox mesh (if one exists). This will only be the case if the scene has been loaded from an IWF file and the file contains a skybox entity.

The final section of the above code sets up the lights on the device and enables them. Unlike earlier demos where the light group system was employed, in this demo we are simply setting up the first N lights found in the IWF file. Alternatively, if we are loading the scene straight from an X file, four direction lights are hard-coded and set up as we saw earlier.

Why are we setting the same lights each time the render loop executes if they never change? The reasons are twofold. First, if you decide to integrate the light group system into this demo yourself, you will absolutely need to set and unset the lights belonging to each light group before rendering its assigned polygons. Second, this approach allows you to place code in the scene class that might dynamically change the light properties. By setting the light properties each time, we make sure that any dynamic changes to a light slot's properties will be reflected in the next render.

The next section of code contains the significant changes from previous implementation of CScene. We render our scene by looping through each CObject in the scene object array. For each object we check its CTriMesh pointer and its CActor pointer to see which are valid. This will tell us whether the object contains a single mesh or an actor encapsulating a complete mesh hierarchy. If the mesh pointer is valid then we set the object's world matrix and FVF flags on the device. If the actor pointer is valid then we pass the object's world matrix into the CActor::SetWorldMatrix function. Earlier we saw that by passing true for the second parameter (as in this case) the hierarchy will be traversed and the absolute world matrices for each frame in the hierarchy will be generated. At this point, either the mesh or the actor is ready to be rendered.

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

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

        } // End if mesh
        else
        {
            // Set up actor / object details (Make sure we update the frame matices!)
            pActor->SetWorldMatrix( &m_pObject[i]->m_mtxWorld, true );

        } // End if actor
```

Because the scene is using its meshes and actors in non-managed mode, we must now loop through every global subset in the scene's ATTRIBUTE_ITEM array, fetch the texture and material for that subset, and set them on the device. If a subset contains no material, then a default material of bright white is used. If a subset has no texture, then the texture stage is set to NULL and the subset will be rendered in shaded mode only.

With the states set, we call either CTriMesh::DrawSubset or CActor::DrawSubset to render the given subset depending on which type of object we are processing.

```
        // 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
            if ( pMesh )
                pMesh->DrawSubset( j );
            else
                pActor->DrawActorSubset( j );

        } // Next Attribute

    } // Next Object

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

}
```

As we know, CTriMesh::DrawSubset simply wraps a call to the ID3DXMesh::DrawSubset function, instructing the underlying mesh to render its subsets. If the object contains an actor then CActor::DrawSubset is called and this function does a lot more work. We examined this function earlier and saw how it traverses every frame in the hierarchy and searches for frames that have mesh containers attached. For each one found, the frame's world matrix is set as the device world matrix and then CTriMesh::DrawSubset is called for each mesh attached to that frame. This will cause all matching subsets in all meshes contained in the hierarchy to be rendered.

Finally, we disable all of the lights at the end of the function. While this is not strictly necessary in our current demo, it does allow us to safely animate or even delete lights from the scene's light array if we wanted to. This step makes sure that if a scene's light is deleted, that its properties will not continue to be used by the light slot is was previously assigned to.

CScene::ProcessReference

Before wrapping up this workbook, we will examine the modified CScene::ProcessReference function, which now handles creating actors from IWF file external X file references.

As usual, the scene uses the IWF SDK classes to load data from IWF files and then calls various functions (ProcessMeshes, ProcessMaterials, ProcessVertices, etc.) to extract the data from the IWF SDK objects and copy it into its own arrays. Recall that the ProcessEntities function is called to process any entities stored in the IWF file and this function in turn called the ProcessReference function if an entity being processed was a GILES™ reference entity. The ProcessReference function was implemented in the previous workbook to extract the filename of an external reference and use it to load a CTriMesh and add it to the scene. Now, this function (while mostly unchanged) creates a CActor from the reference filename instead.

We will not dwell long on this function since much of the actor creation and setup is the same as the CScene::LoadSceneFromX function which we have already covered.

The first section of the function checks that this is an external reference before continuing. It then builds the filename string used to create the absolute filename (with path) for the X file being loaded. The IWF ReferenceEntity object only contains the name of the X file and does not include path information (for obvious reasons). We do this by adding the entity name to the path name stored in the scene's m_strDataPath string. This data path was stored in this string in the LoadSceneFromIWF function by truncating the absolute filename passed in.

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

    // Skip if this is anything other than an external reference.
    // Internal references are not supported in this demo.
    if (Reference.ReferenceType != 1) return true;

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

We will not immediately load the actor (like we do in the LoadSceneFromX function) because the IWF file might contain multiple instances of the same actor in the scene. So, first a search is done through the scene's CActor array to see if an actor exists with a matching name. If one is found then we will use this pointer, otherwise, we will need to create a new CActor, register the scene callbacks and load the file:

```
    // Search to see if this X file has already been loaded
    for ( ULONG i = 0; i < m_nActorCount; ++i )
    {
        if (!m_pActor[i]) continue;
        if ( _tcsicmp( Buffer, m_pActor[i]->GetActorName() ) == 0 ) break;

    } // Next Actor

    // If we didn't reach then end, this Actor already exists
    if ( i != m_nActorCount )
    {
        // Store reference Actor.
        pReferenceActor = m_pActor[i];
    }   // End if Actor already exists
    else
    {
        // Allocate a new Actor for this reference
        CActor * pNewActor = new CActor;
        if (!pNewActor) return false;

        // Load in the externally referenced X File
        pNewActor->RegisterCallback(CActor::CALLBACK_ATTRIBUTEID,CollectAttributeID,this );
        hRet = pNewActor->LoadActorFromX( Buffer, D3DXMESH_MANAGED, m_pD3DDevice );
        if ( FAILED(hRet) ) { delete pNewActor; return false; }

        // Store this new Actor
        if ( AddActor( ) < 0 ) { delete pNewActor; return false; }
        m_pActor[ m_nActorCount - 1 ] = pNewActor;

        // Store as object reference Actor
        pReferenceActor = pNewActor;

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

At this point, the local variable pReferenceActor is pointing to the actor we need. This will either be a new actor we have just loaded or a pointer to an actor that already existed in the scene's CActor array.

Our final task is to add a new CObject to the scene's object array and store this actor pointer in it. As it happens, IWF references also contain a world matrix describing where the reference is situated in the scene, so we can copy this matrix from the reference object into the CObject as well.

```
    // Now build an object for this Actor (standard identity)
    CObject * pNewObject = new CObject( pReferenceActor );
    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;
}
```

Conclusion

In this lab project we put in place the foundations of a very useful class called CActor, which we will see used repeatedly throughout the rest of this course. This will be true in next chapter when we cover hierarchical animation and in the following chapters when we cover skinning and skeletal animation. The CActor class is a very welcome addition to our toolkit and it will make our job a lot simpler from here on in.

While we lightly touched on hierarchical animation in this chapter, the next chapter will be devoted entirely to this subject. We will learn how to use the D3DX animation controller to load complex keyframed animations that can be blended with other animations and applied to our hierarchy. These techniques will then be carried into later lessons where the animation controller will be used to animate the skeletal system of game characters and other dynamic entity types. Be sure that you understand how our hierarchy (and CActor) system works before moving on, since it is the core of just about everything we will do in the next few lessons.