Chapter Nine

Frame Hierarchies

Introduction

In Chapter Eight, while examining the process of loading and rendering mesh data stored in X files, we learned that the D3DXLoadMeshFromX function provided a useful and yet restrictive service: regardless of how many meshes were contained in the X file, a single mesh was always output. Multiple meshes, were they to exist in the file, were simply transformed and collapsed into a single output mesh. While this outcome may be acceptable under certain conditions, a more flexible X file loading system that allows us to optionally maintain the individual mesh entities would certainly be ideal. This is because there are circumstances that frequently occur in game development where the preservation of the distinct meshes is preferred. We will examine a number of these cases as we move forward in this chapter.

For example, consider the case where an X file is used to store the representation of an automobile. It is not uncommon for an automobile model to be constructed using several meshes (ex. chassis, doors, trunk, hood, and tires), each requiring independent animation. The game designer may want the wheels to rotate as the car accelerates or the doors to swing open and closed when the player enters or exits the vehicle. It would be difficult indeed to apply such localized animations to the various sections of the automobile if a single mesh representation was used. Earlier lessons have taught us that when we apply a transform to a mesh, it is applied to all of its vertices. Therefore, if we wish to represent a model that has independently moving parts, it should be represented as a collection of unique meshes that, taken together, represent the overall model. With a multiple mesh representation, each mesh is rendered separately. This affords us the freedom to alter the transformation matrix for a given mesh before it is rendered, applying animation to position, orientation, or even both simultaneously.

The X file loading techniques we have discussed thus far would not be able to be used in circumstances where such a representation is stored in the X file. Using the D3DXLoadMeshFromX function, we would effectively abandon the multi-mesh layout of the file, getting back a single mesh that is the sum of all specified submeshes. While the process of collapsing submeshes into a single mesh is handled intelligently by D3DX (the original submeshes are correctly positioned within the overall model representation) we will have lost the ability to animate/transform those original sub-meshes separately from the rest of the model.

When constructing or loading a multi-mesh representation, we do not simply want the various sub-meshes stored such that they are transformed and rendered in total isolation from one another. Rather, it would be expected that the underlying spatial relationship between each of these individual meshes will still be preserved, even when these submeshes experience their own unique transformations. If the vehicle moves, then all of its submeshes must move together, to maintain model integrity. We would not want to move the automobile chassis mesh only to find that the wheels have not moved along with it. However, we do want the freedom to rotate the wheels without rotating the chassis.

Designing such an automobile representation in any of the popular 3D modeling packages is quite common, and we are generally going to want to use a multi-mesh representation in our application as well. One important question is, when this automobile is exported to the X file format, how are these meshes arranged in the file such that they can be imported and assembled into a multi-mesh vehicle model that behaves as expected? The quick answer is that each mesh in the file will have a

transformation matrix assigned to it that describes how it is positioned relative to something else in the scene. This 'something else' might be another mesh used in the model, or some other point in space (perhaps even the scene origin itself). The result is a spatial hierarchy of meshes, arranged in a tree-like fashion. There is a root node, and then a series of child nodes positioned relative to the root. These children may have child nodes positioned relative to themselves, and so on.

Using our automobile example, the chassis mesh might be stored in the root node of the tree and the four wheel meshes might be stored as child nodes of the root. In each child node, a wheel submesh would also be accompanied by a 4x4 transformation matrix which describes the position and orientation of the wheel mesh. This description is *not* an absolute (i.e. world space) transformation as we might expect, but is in fact a transformation *relative* to the parent node; in this case, the chassis. In this simple example, the transformation matrix for each wheel mesh would describe the position and orientation of the wheel relative to the location and orientation of the chassis mesh (the parent node). Therefore, regardless of how we animate the chassis (by applying transformations to the matrix in the root node), the world matrix used to render each submesh can always be calculated such that object integrity is maintained.

In this example, the root node matrix is used to set the world space transformation for the chassis mesh. This is essentially the world matrix for the entire automobile representation. It is this matrix that would be used to transform the chassis mesh into world space prior to rendering it. The world space matrices used to render each submesh are *not* directly contained in the representation (or in the X file for that matter) and must be calculated each time the root node matrix is updated by the application. We accomplish this by combining a parent node matrix (the chassis/root matrix in this example) with the child transformation matrix (a wheel matrix) prior to rendering. As each child wheel matrix describes the position of the wheel relative to the chassis, and the root node matrix describes the world space position of the chassis itself, multiplying these two matrices produces the world space transformation matrix for the wheel. By generating the world matrices for each submesh at runtime, just prior to rendering, we are assured that whenever we apply a transformation to the root node matrix, the world matrix generated for each child mesh will correctly transform the child along with the parent. This ensures that the spatial relationship between the children (the wheel meshes) and the parent (the chassis mesh) is maintained and that changes to the root node matrix are propagated through the entire tree. The result is that we can move the entire model, with all of its submeshes, by applying transformations to the root node matrix and then calculating the new world space matrix for each child mesh that results from these changes to the root. Note as well that we also have the ability to apply transformations to the relative matrix of a child object. In this manner, local rotations could be applied to our wheel meshes without being applied to the chassis.

Graphics programmers commonly refer to the data arrangement that describes these relative relationships as a *spatial hierarchy* or a *frame hierarchy*. Understanding how to load, animate, and render a multi-mesh frame hierarchy is vital if we are to progress past the simple mesh representations we have covered thus far. To understand the relative arrangement of a frame hierarchy, we must first examine the concept of *frame of reference*. This will be the subject of the next section.

9.1 Frame Hierarchies

Physicists and mathematicians use the concept of a *frame of reference* (a.k.a. *reference frame*) to describe an arbitrary point in space with an arbitrary orientation. We use frames of reference all the time in our everyday lives to achieve all manner of tasks and as such, we are often not aware that we are using them at all. Without a frame of reference it would be impossible to evaluate a grid coordinate on a map, give somebody directions to your house, or even specify a position in 2D or 3D space.

A simple example of why we need to use a frame of reference is apparent when we consider giving somebody travel directions. Imagine that Person A lives in San Francisco and that Person B resides in Miami. If A asks B the general direction he should travel to reach Mexico City, B might incorrectly assume that they both live in Miami and offer information describing Mexico City as being positioned due west. If A also assumes that B is living in San Francisco and is offering his directions based from that location, A would head off in a westerly direction and find himself in the middle of the Pacific Ocean. Of course, Person A could have stayed dry if B had revealed that his directions were using Miami as a frame of reference. The correct directions should have been given as "From Miami, head west". In this case, Person A could have traveled to Miami first and then followed the directions provided, or he could have extrapolated his own directions from the Miami-based directions. This would have informed him that he needed to head due south and then east.

Another common example of frame of reference is one which demonstrates how the perceived velocity of a moving object is changed when classified using different frames of reference. This example will also have some relevance to 3D animation, as we will learn later. To demonstrate, imagine Person A (who we shall refer to as Speedy Steve) riding a motorcycle up the street in a northerly direction at a speed of 150 kph. Just as he passes by Person B (who we shall call Dastardly Dennis), Dennis walks out into the center of the street and positions himself directly behind Speedy Steve. He then pulls back on his super-slingshot as hard as he can, and releases a stone in the direction Speedy Steve is traveling. The stone leaves the slingshot and travels at a constant 155 kph. Dennis is a rather nasty fellow whose intention is to strike Speedy Steve in the back of the head so hard that he is knocked from his motorcycle. However, what Dennis had cruelly planned does not come to fruition. Instead, the stone gently taps Speedy Steve on the back of the helmet, so softly that he hardly notices it. He carries on northbound and vanishes out of sight. Dennis is left scratching his head and wondering why his dastardly plan failed.

The reason Dennis' plot did not work becomes clear when we examine the nature of the relative velocities. Although the stone that Dennis fired was traveling at a frightening 155 kph in Dennis' frame of reference, Speedy Steve is also traveling at 150 kph himself in the same direction. So from Steve's perspective, the stone is only gaining on him at a relative velocity of (155 – 150) 5 kph. That it, using Speedy Steve as a frame of reference, the velocity of the stone is only 5 kph. Therefore it strikes him with very little force. Of course, if Speedy Steve was stationary like Dennis, then the stone would have hit him at 155 km per hour and probably done some serious damage. For maximum dastardly effect, Dennis should have preempted Steve's arrival and been laying in wait. If he would have released the stone at the same speed from in front of Steve, and in a southerly direction, it would have hit Steve head-on with devastating force. Using Steve as a frame of reference in this example, what would be the velocity of the stone at the point of impact? The answer is (155 + 150) 305 kph. This is because both

Steve and the stone are traveling in exact opposite directions and therefore their combined velocity is used to calculate their impact velocity. As Steve hurtles towards the stone, it in turn is hurtling towards him. So from Steve's point of view, that stone is getting closer at a very high rate of speed indeed. The main point is that the position, orientation and velocity of something can actually be very different when observed using different frames of reference.

As it turns out, we have consistently used frames of reference throughout this course series when dealing with mesh data and transformations, although their application is so obvious it might be overlooked. The 3D vector <10, 10, 10> would mean nothing to us, and could not even be evaluated, if we did not know that these numbers describe a position in space relative to the origin of some coordinate system. Indeed, a coordinate system provides a means for describing locations within a frame of reference. When plotting points in any coordinate system (world space, view space, projection space, screen space, etc.) we are using a frame of reference with the origin of the coordinate system serving as the origin of the frame. We know that this is usually classified as position (0, 0, 0).

Throughout Graphics Programming Module I we discussed mathematical 'spaces' and coordinate systems in some detail. So we see now that in effect, we have been discussing frames of reference all along. Each of the 'spaces' we discussed (model space, world space, view space, etc.) had an origin and orientation which described a frame of reference. All positions in that space are defined relative to this origin and axis orientation. In model space, all vertex positions are defined relative to the fixed position (0,0,0) and the coordinate axes are the standard orthogonal XYZ axes. In world space, those same vertices are then defined relative to another fixed point in space (also assigned the coordinates (0,0,0)) with its own orthogonal set of XYZ axes. This point is considered the origin of our world space system and is therefore the frame of reference for all vectors plotted in world space. View space is yet another frame of reference that exists at the camera position (in which (0,0,0) is also used as the frame of reference origin for all points described within that space). It too has its own set of XYZ orthogonal axes which, with respect to the world's standard axes, describe the orientation of the camera.

It should now be clear then that transformations provide us with a means for modifying values within a frame of reference. If we consider the three transformations we have studied in most detail (scaling, translation, and rotation) we understand that they can be used to modify the position of a point in order to generate a new position within the same coordinate space (frame of reference). Transformations also provide us the ability to transform (or perhaps more appropriately, "reclassify") values from one frame of reference to another. We have seen how the world space transformation transitions vectors relative to the model space frame of reference into vectors relative to the world space frame of reference. Many people actually find this a more intuitive way of visualizing transformations from one space to another. We are simply taking a point in one frame of reference and reclassifying it in another.

So then a frame of reference is essentially a *local space*: it has an origin from which emerges a set of axes that define how the space is oriented and thus how things in that local space are measured with respect to that origin. Through the use of transformations we can create a situation in which we redefine points in one frame of reference to be classified with respect to another frame of reference. That is how our models maintain a presence in our game world. Each model usually has its own frame of reference (model space, in which its vertices are initially defined), but through our transformation pipeline, we can also describe how those same vertices are situated in the world frame of reference, the camera frame of reference, and so on.

The reason why an understanding of frames of reference is so important can be demonstrated when we examine our automobile example. We will use a simple example of five separate meshes (four wheels and a car body) for our automobile. These meshes will be arranged in the X file in a hierarchical format such that the positions of the four wheels are defined relative to the position of the car body. In other words, the car body is stored at the root of the automobile hierarchy and the four wheels will be children of this node.

In Fig 9.1 we see the automobile hierarchy. In the diagram, we see a **frame** structure which will represent a node in the tree. A single frame contains a 4x4 transformation matrix defining a location and orientation as just described and stores pointers to *N* child frames. Each frame (short for 'frame of reference') can also store one or more pointers to data items -- car part meshes in this case. In this example, each frame contains only a single child mesh, but this need not always be the case.

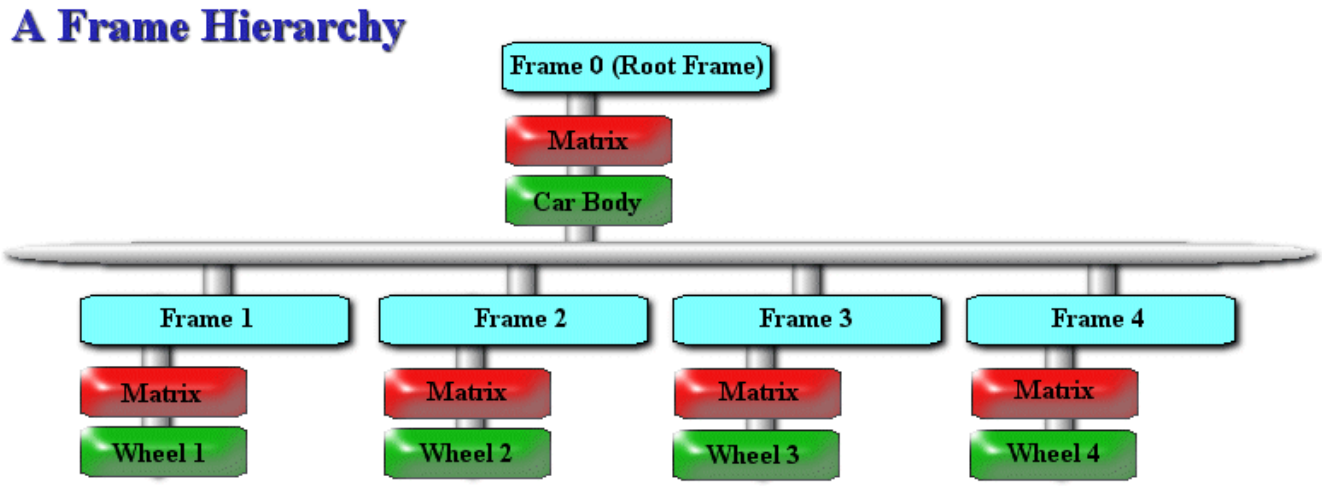


Figure 9.1

Each 4x4 frame matrix consists of two components. It has a translation component in its bottom row which describes the origin of the frame of reference. This origin is defined *relative to* its parent (i.e. it tells us the location of the origin of the child frame within the parent's frame of reference). The rest of the matrix describes the parent-relative orientation of the frame itself (i.e. how much rotation off the parent axes the child axes happen to be). The combination of parent-child relationships defined by the transformations stored in each node matrix creates a spatial hierarchy.
While the root node has no direct parent in the hierarchy, we can still think of its matrix as a relative matrix. In this particular case, the root node 'parent' is the world system origin, and the root frame matrix stores a position/orientation relative to that system. As such, the root frame transformation matrix will serve as a world matrix for the entire hierarchy, enabling us to position and orient the entire set of hierarchy data in our world (assuming the root node does not get attached to some other parent of course – for example, an automobile carrier that is transporting cars to the showroom). This is possible because the children of the root frame are defined in its local space (i.e. reference frame), so by moving the root frame in the world, we are moving its frame of reference and thus any items that are defined within it.

So in our simple case, applying a translation to the root frame would position that frame somewhere in the game world. Certainly we would expect that the children must also experience the same transformation so that they follow their parent to the new location (the spatial dispositions of the child

objects in the hierarchy must be preserved in the process of moving their parent). Keep in mind that while the root node world transformation must be propagated down to each child so that they move as well, the relative relationships between parents and children must remain intact since they exist within that given reference frame. Thus the world transformation and the parent-relative transformations at each node must be combined in some fashion. We will examine the appropriate matrix combination strategy shortly.

The root frame in this particular example stores a single mesh (the car body) and its transformation matrix describes the world orientation and position of the frame. Thus to transform the car body mesh vertices into world space, we would just multiply the vertices by the frame 0 matrix directly. In that sense, we can think of the mesh stored in the root frame as having a local space defined by that root frame (i.e. the mesh vertices are defined as if the root frame origin were the model space origin for the mesh itself). We will see shortly that in fact, this holds true for all of our frames and that we are essentially building a tree of local space coordinate systems all nested within each other.

We will want to store each wheel in a separate frame so that rotation can be applied to the individual wheels when the car accelerates. This would be a local rotation for the wheel, likely around its local X or Z axis depending on how the wheel model was oriented in the modeling program. Thus matrix multiplication order is very important here since the frame matrix stores a translation that already positions the wheel away from the center of the car body. Of course, we would not want that translation to happen first, since that would just rotate the wheel around the parent. Thus the local wheel rotation would have to happen before the wheel was translated into its world space position. In other words, we want to start with the wheel at the origin, rotate it around its local X or Z axis so that it spins, and then push that wheel out into its final position relative to its parent. So the local rotation matrix would have to come first in the multiplication order, followed by the parent-relative frame matrix, and not the other way around.

For simplicity, let us forget about the mesh vertices and work only with mesh center points. In a sense we can even think of each mesh as a single vertex in a model called ‘scene’. Let us further assume that we would like the car to be moved to world location (100,100,100) and we define a 4x4 translation matrix **W** to accommodate this.

If we assume that our car body is centered at the origin of the model and thus requires no additional transformations relative to the center of the model, its parent-relative (i.e. frame) transformation matrix would be an identity matrix (which we can call **F0** for Frame 0). As mentioned above, this will actually be our root frame matrix in the case of the car body since this mesh stored in the root frame.

Let us call the final world space transformation matrix for the car body **B**. This will be the matrix that we use to transform the car body mesh into world space.

B = F0 * W
B = W (since F0 is an identity matrix)

As you can see, all we have done above is multiply the matrix in the root frame with translation matrix **W**. This results in matrix **B**, which is a matrix describing a world space translation of (100,100,100) to move the car body vertices into world space.

Now let us examine one of our wheel frames and say that in this example, each wheel is offset 10 units from the center of the car body. Thus each wheel frame will have a 4x4 translation matrix defining its relationship to its parent. This is the frame matrix for the wheel and we will call it **F1** (for the first wheel).

The question is, how do we find the final *world space* transformation for the wheel (we will call it **T1**)? F1 does not describe the position of the wheel using the world space frame of reference, but that is exactly what we need in order to transform its vertices into world space prior to rendering. Therefore, the wheel frame matrix must be multiplied with something else to generate the final world space transformation matrix for the wheel.

T1 = **F1** * ?

We know that matrix **F1** contains a transformation relative to its parent **F0**, so these will have to be multiplied together so that both meshes are in the same space and share a common frame of reference. However, as we have potentially repositioned **F0** in the world by multiplying it with translation matrix **W**, the final world space position of the wheel must take this into account also so that the wheel and the car body frames are translated by the same amount.

T1 = **F1** * (**F0** * **W**)
T1 = **F1** * **B**

As you can see, **F1** describes its position relative to **F0**, so multiplying these matrices together will move the wheel from its own local space into the local space of the entire automobile representation. We can imagine at this point that the wheel has been correctly positioned relative to the car body but that the car itself is still at the origin of world space. Finally, **W** contains the transform matrix that positions the entire model in the world (we used this to position the car body mesh) so this too must be applied to the wheel mesh. Therefore, the car body world matrix is **B=F0*W** and the world matrix of the wheel is **F1*B**.

So if we wanted to rotate the wheel around its own local Z axis, it should be apparent to you where that 4x4 rotation matrix ($\mathbf{Z_{rot}}$) would need to go in the chain:

T1 = $\mathbf{Z_{rot}}$ * **F1** * **F0** * **W**

Now in this particular case, the parent of the wheel is the root node of the hierarchy. Thus the equation simplifies to:

T1 = $\mathbf{Z_{rot}}$ * **F1** * **B**

Hopefully you see the pattern here. If you wanted to attach something to the wheel (a hubcap for example), you would add a new frame for it and define it relative to its parent (the wheel) and continue on down the chain.

It is important to keep in mind that any transformation at a node affects all of its children when we combine the matrices in this way to generate world matrices for each frame (and the meshes they may

contain). In that sense, it can be helpful to think of frames as a system of joints. Any part of the scene that needs to be moved or rotated independently will be assigned its own joint. The connections between those joints are essentially offsets of some distance from the parent, represented by the translation row in the frame matrix. How those joints are oriented would be represented in the upper 3x3 portion of the matrix (an orthogonal rotation matrix). Thus at any given node in the tree, its world transformation matrix is defined as the concatenation of any local transforms we wish to apply to that frames matrix, the parent-relative transformation matrix stored in the frame itself, and the parent's current **world** space transformation matrix (which is itself a by-product of the same logic). Therefore, in order to calculate the world space transformation matrix of a given frame in the hierarchy, we must have first calculated the world space transformation matrix for its parent frame, as it is used in the concatenation.

In our automobile example we saw that each wheel has its own frame of reference (Fig 9.1). We now also know that a frame matrix stores a mathematical relationship describing some translational offset from a parent frame and a local orientation for that node which is relative to the parent node's orientation. Practically speaking, we can consider the mesh vertices that are stored at a given frame as children of that frame as well. That is, they utilize the same matrix concatenation scheme to describe how far away they need to be from the frame root node. The only real difference is that the vertices will have their final matrix concatenation done in the pipeline during rendering (just as we saw in Graphics Programming Module I) while child frames have to calculate their new matrix information manually as we descend the hierarchy. In reality though, both are experiencing the same thing -- offsetting and orienting with respect to their parent, which itself was offset with respect to its parent and so on up the chain. Also remember that while the root frame's matrix will often serve as the world matrix for any immediate child meshes of the root frame (again, the parent of the root frame is often going to be the world origin itself) we now know that this is not the case for meshes stored at subsequent child frames -- their world matrices must be generated by stepping down the tree to those frames, concatenating any direct or indirect parent matrices as we go.

Before wrapping up, let us step through our automobile example one more time, using some real values to try to clarify the important points.

If we reconsider Figure 9.1, we can imagine the root frame (we will assume the root frame has no specified parent in the scene and thus defaults to having the world origin as its parent frame) might have a matrix that stores a translation vector of (100,100,100). This then describes the center of the entire automobile as being at location (100,100,100) in world space. We will assume for now that the matrix contains no rotational information and that the car body has been designed such that its front faces down the world Z axis when no rotations are being applied. Now, let us assume that the wheels themselves are placed at offsets of 10 units along both the automobile's X and Z axes, such that the front driver's side wheel for example (wheel 1) is positioned (assuming a steering wheel on the left hand side) at offsets <–10,0,10> from the automobile center point, the front passenger wheel (wheel 2) is positioned at offset <10,0,10> from the automobile center point, the driver side rear wheel (wheel 3) is positioned at offset <–10,0,-10> and finally the passenger side rear wheel is located at offset <10,0,-10> from the automobile center point. Remember that the world space center point of the automobile is the root frame position described by its matrix.

Prior to our previous discussion, we might well have assumed that the information would be stored in the X file such that the matrices of each wheel frame would describe the absolute world space position

of that wheel. That is, it would have been easy to assume that the five matrices in this example had their translation vectors (the 4th row of each matrix) represented as:

Matrix Translation Vectors (This is not correct)

Root Matrix	= **100 , 100 , 100**	(The World Matrix of the Hierarchy)	
Wheel 1 Matrix	= **90 , 100 , 110**	(World Matrix offset by	–10, 0, 10)
Wheel 2 Matrix	= **110 , 100 , 110**	(World Matrix offset by	10, 0, 10)
Wheel 3 Matrix	= **90 , 100 , 90**	(World Matrix offset by	-10, 0, -10)
Wheel 4 Matrix	= **110 , 100 , 90**	(World Matrix offset by	10, 0, -10)

If the wheel frames contained absolute matrices such as these, which described the complete world space transformation for any child objects of the frame (the wheels in this example), then rendering the hierarchy would be very straightforward:

1. set root frame matrix as device World Matrix
2. render any child meshes of the root frame (the car body)
3. for each child frame of the root frame
4. set child frame matrix as device World Matrix (a wheel matrix)
5. render the wheel mesh for that child

While this is certainly easy enough to code (and in fact, we have done this throughout the last course module and in the last chapter), this is no longer the hierarchy we envisioned. In this case, all frames share the same parent since they are all defined relative to the same point in space (the world origin). What was a hierarchy has now devolved into a set of standard world space meshes. The relationship between the car body and the wheels exists in name only, for the children are not affected by changes to their assigned parent. In terms of their behavior, the world is their parent since that is what they are defined relative to.

However, if we were to examine the translation vector of each of the five matrices in a proper automobile X file, we would correctly find that the wheel matrices would actually describe transformations relative to their parent (the root frame in this case). Again, we would notice that the root frame has no parent frame so can be thought of as being a transformation relative to the origin of world space and serves as the base world matrix for the entire hierarchy:

Matrix Translation Vectors (Correctly stored relative matrices for child frames)

Root Matrix	= **100 , 100 , 100**	(The World Matrix of the Hierarchy)
Wheel 1 Matrix	= **-10 , 0 , 10**	(RootMatrix * Wheel1Matrix = 90 , 100 , 110)
Wheel 2 Matrix	= **10 , 0 , 10**	(RootMatrix * Wheel2Matrix = 110 , 100 , 110)
Wheel 3 Matrix	= **-10 , 0 , -10**	(RootMatrix * Wheel3Matrix = 90 , 100 , 90)
Wheel 4 Matrix	= **10 , 0 , -10**	(RootMatrix * Wheel4Matrix = 110 , 100 , 90)

Using the render approach described previously would obviously not work any longer. If we were to simply set wheel matrix 1 as the world matrix before rendering wheel 1 then the wheel would be rendered at world space position (–10,0,10) while the car body sits at position (100,100,100). Also, when we wish to move the automobile about in the world, ideally we only want to translate or rotate the

root frame and let the children experience these updates appropriately. As discussed previously, for this to work we must move through the hierarchy one level at a time, concatenating matrices as we go. So we would start at the root level, set the root frame's matrix and render any immediate child meshes -- in this case the car body. At this point the car body would be rendered at world space position (100,100,100). Next we would iterate through each child frame of the root. For each child frame we create a temporary matrix by multiplying its frame (i.e. parent-relative) matrix by the root frame matrix. In the case of wheel 1 for example, the translation vector of this combined matrix in the above example would now be (90,100,110). This clearly describes a world space position, but it also shows us that this wheel remains correctly offset (-10,0,10) from the current root frame position. We then set this temporary combined matrix as the device world matrix and render that wheel. We repeat this process for each additional wheel.

Since a given hierarchy could be many levels deep, we see right away that this concatenation/rendering approach would be nicely served by a simple recursive function call. When the matrix of any frame in the hierarchy is altered, these changes will automatically be propagated down through all child frames as we traverse the hierarchy and combine the matrices at each frame prior to rendering their immediate child meshes.

If we were to change the root frame matrix in our above example to some new position, then the next time we render the hierarchy and step through the levels of the hierarchy combining matrices, the world space positions of all the wheels would be updated too -- even though we do not physically alter the matrices of these child frames. This last point is important to remember. If our application were to alter the root frame matrix so that it now had a translation vector of <50,0,50>, then the next time the automobile is rendered using the concatenation technique just described we know certain things will be true. For starters, the car body would be rendered with its center point at (50,0,50). This is obvious because for the root frame meshes we simply set the root frame matrix as the world matrix. When we render the wheel 1 mesh, we create a temporary world matrix by combining the root frame matrix with the wheel 1 frame matrix (which still has a translation vector of <–10,0,10> and has not changed) and use this to render the wheel 1 mesh as before. This time, the combined matrix would have a translation vector which places the wheel at position (40,0,60) in the world, still an offset of (-10,0,10) from the root matrix position (50,0,50). So while the wheel matrices have not changed, the temporary matrices we generate to render the frame's meshes have changed. The result is that the wheels will automatically move along with the car body as expected and maintain their proper spatial positioning at all times.

Matrix Translation Vectors (Root Matrix now contains position 50,0,50)

Root Matrix	= **50, 0, 50**	(The World Matrix of the Hierarchy)
Wheel 1 Matrix	= **-10, 0, 10**	(RootMatrix * Wheel1Matrix = 40 , 0 , 60)
Wheel 2 Matrix	= **10, 0, 10**	(RootMatrix * Wheel2Matrix = 60 , 0 , 60)
Wheel 3 Matrix	= **-10, 0, -10**	(RootMatrix * Wheel3Matrix = 40 , 0 , 40)
Wheel 4 Matrix	= **10, 0, -10**	(RootMatrix * Wheel4Matrix = 60 , 0 , 40)

The same logic holds true for any rotations or scaling we may apply to any matrix in the hierarchy. If we were to rotate the root frame 45 degrees clockwise about its local Y axis (by creating a Y axis rotation matrix and multiplying it with the root matrix), then when we traverse the hierarchy combining matrices as we go from level 1 to level 2, the temporary matrix created for each wheel will also rotate the wheel by the same amount. The offset of <–10,0,10> for wheel 1 for example is still preserved of course. The

result is that as the car body rotates, the wheels rotate too -- not about their own center points but about the center point of the car body. This is because the frame of reference they are defined in is experiencing its own local rotation, thus dragging its children along with it. This allows the children to remain in their correct position at all times.

Seeing some example code on how to traverse and render a hierarchy should cement all of this into place. This is not our final code of course because we have not yet discussed how to load the hierarchy from an X file or what support structures we may need, but we can imagine that each frame in the hierarchy could be represented using a structure a little like the following:

```
struct CFrame
{
      LPSTR             pFrameName;
      D3DXMATRIX        Matrix;
      CSomeMeshClass *  pMesh;
      CFrame *          pChildFrame;
      CFrame *          pSiblingFrame;
};
```

Keep in mind how the frames are connected together and that each frame contains a pointer to a mesh which is assigned to that frame. Everything stored in this frame, be they vertices or other frames are all children of the frame. That is, they are live in its local space and are defined relative to its origin and are affected by its orientation. In our example, if this was the root frame, then this mesh pointer would point to the model for the car body.

The pChildFrame pointer points to a linked list of child frames, but it is important to note that the linked list is not organized in the way you might expect. Rather, you should think of the pSiblingFrame member as being the pNext member of a traditional linked list and then visualize each tier in the hierarchy as a separate linked list. So the linked list of one level will be attached to its parent level using the pChildFrame pointer of the parent frame. Using our hierarchy as an example, the root frame would have its pSiblingFrame pointer set to NULL because the root frame is not actually part of any linked list of frames at that level in the hierarchy (there is only one frame at the root level – Level 0). The root frame's pChildFrame pointer would point to Frame1 (the first wheel), linking the root level to Level 1. Frame 1 would have its pChildFrame point set to NULL (as will frames 2, 3, and 4) because it has no additional child frames. However, Frame1 will connect to Frame2 via its pSiblingFrame pointer. Frame2 will have its pSiblingFrame pointer pointing to Frame3. Finally, Frame3 will have its pSiblingFrame pointer pointing to Frame4. Frame4 will have a NULL pSiblingFrame pointer since it is the end of the list at that level in the hierarchy.

In summary, the pChildFrame pointer for a given frame structure points to the first child frame in a linked list of child frames and represents another level in the tree. Each child frame in the linked list would be attached by their pSiblingFrame pointer. From this we can understand that all frames linked by their pSiblingFrame pointers belong to the same level in the hierarchy and share the same parent. Their transformation matrices will define their positions and orientations relative to the same frame of reference, which would be the world space position described by the parent frame. In our automobile example, the hierarchy has two levels: the first level contains only the root frame, the second level contains the four wheel frames. The wheels are all considered siblings because they share the same parent.

In order to render this hierarchy, we could use a recursive function like the following:

```
void DrawHierarchy(CFrame *pFrame, D3DXMATRIX *pParentMatrix,
                   IDirect3DDevice9 *pDevice )
{
    D3DXMATRIX  mtxCombined;

    if ( pParentMatrix != NULL)
      D3DXMatrixMultiply( &mtxCombined, &pFrame->Matrix, pParentMatrix);
    else
      mtxCombined = pFrame->Matrix;

    pDevice->SetTransform ( D3DTS_WORLD , &mtxCombined );
    pFrame->pMesh->Render( );

    // Move on to sibling frame
    if(pFrame->pSiblingFrame)
        DrawHierarchy(pFrame->pSiblingFrame, pParentMatrix, pDevice);

    // Move on to first child frame
    if(pFrame->pChildFrame)
        DrawHierarchy(pFrame->pChildFrame, &mtxCombined, pDevice);
}
```

There are a few key points to note about this code. First, we only need to call this recursive function once from our application, passing in the hierarchy root frame as the first parameter and NULL as the second parameter (because the root frame has no parent). Actually, this latter point is only true if we do not wish the root frame to move in our world in some way. If we wanted to translate the root frame, and thus the entire hierarchy, to some new location in the world, we can pass in a translation matrix as the second parameter. This represents a translation relative to the world frame of reference, which is the root frame's parent under normal circumstances.

Of course, we could simply modify the root frame matrix directly to generate the same result, but it is preferable not to do this. If we change the contents of the root frame matrix (which is virtually always an identity matrix), we make it more difficult to attach this hierarchy to a node in another hierarchy should we want to do so at a later time. For example, consider a machine gun hierarchy that contains moving parts. If we wanted to attach the root frame of the machine gun (assume the handle/grip is the root frame location in the gun model hierarchy) to a character's hand frame, then all we would have to do is make the machine gun root frame a child of the hand frame. If we kept the gun root frame matrix an identity matrix, then as the character moves his hand and passes those transforms down the tree, the matrix multiplication step would just give the machine gun root frame the same position/orientation as the character's hand (since the multiplication by identity just results in the same matrix). Thus attaching and detaching hierarchies to/from other hierarchies to build more complex hierarchy trees becomes a simple matter of pointer reassignment in a linked list.

However, to keep things simple, in the root node case for this example, pParentMatrix will be NULL. Therefore, the matrix of the root frame would not be combined and would instead be used directly as the world matrix for rendering the car body.

You should also note that it is possible for a single frame to contain more than one mesh, so we might imagine that the pMesh pointer actually points to a linked list of meshes attached to this frame. Thus calling the pMesh->Render function would actually step through all sibling meshes in the list and render them. Generally, artists will not design a model where several meshes belong to a single frame because we would have no individual control over these meshes since they would all share the same frame matrix. Applying an animation to the frame matrix would animate all the meshes that are stored in that frame, which is not something we usually want to do. In such an instance, it would probably make more sense for the artist to combine the multiple meshes into a single mesh and assign it to the frame as a single mesh. Either way, for now we can forget about the semantics of multiple meshes per frame because our example deals with only one mesh per frame.

As the code shows, after the child meshes of the current frame have been rendered, it is time to process any sibling frames this level in the hierarchy might have. As it happens, the root frame is the only frame in this level of the hierarchy and does not have any sibling frames. The frame's pSiblingFrame pointer would be set to NULL in this instance and so we move on.

Child frame processing is next. The root frame's pChildFrame pointer will point to Frame1 (in our example). So we call DrawHierarchy again (recursively) passing in as the first parameter the child frame (Frame1) and the root frame matrix as the second parameter (the parent matrix). Keep in mind that at this point the car body mesh has been rendered and the root frame has been totally processed.

As we enter DrawHierarchy for the second time, we are now dealing with Frame1 as the current frame and the root frame matrix is the parent matrix passed in as parameter two. We multiply the frame matrix with the parent matrix so that we now have a temporary combined matrix that contains the complete world space transformation for any meshes in this frame. This combined matrix is sent to the device as the current world matrix and the mesh attached to this frame is drawn (the Wheel 1 mesh in our example).

Next we test to see if this frame has a sibling frame, and indeed it has: Frame1's pSiblingFrame pointer will be pointing to Frame2. So we call the DrawHierarchy function again, passing in Frame2 as the first parameter and the parent matrix passed into this function as the second parameter.

This is a very important point to remember when processing siblings -- we do not pass in the combined matrix that we have just calculated because it is only relevant to the current frame and its children. We can see for example, that while Frame2 should be combined with the root frame matrix (its parent), it should not be combined with Frame1's matrix. So for each sibling that we process, we pass in and use the same parent matrix that was passed into that level of the hierarchy by the level above it. The root frame passed its matrix down to level 2 when it processed its child pointer; now this same matrix (the root frame matrix) is passed along to every sibling at that level and combined with their frame matrices to create their world space transforms.

The story obviously changes when processing child frames. When we call DrawHierarchy for a child frame, we are actually stepping down to the next level of the hierarchy for that particular frame. Therefore, any frames in the lower levels of that branch of the tree must have their matrices combined with the combination of matrices that went before them. If our Wheel 1 frame had a single child frame introducing a new level (a 3rd level) into the hierarchy, this new frame would be traversed down into

when we call DrawHierarchy for the child pointer of Frame1. In that case, we would pass the combined matrix (Frame1 * Parent), and not the initial parent matrix that is used to process siblings. The parent matrix passed into this child of Frame 1, would be (Frame1 Matrix * Root Matrix) which describes the world space position of its parent (Frame 1). It is this world space position that is the frame of reference for the matrix that would be stored in Frame 1's child. In summary, the process is as follows:

1. **For each frame**
2. **Combine frame matrix with parent matrix (if exists)**
3. **Render current frame's mesh(es) using combined matrix**
4. **Process all sibling frames (frames on the same level of the hierarchy). Pass them the same parent frame that the current frame was passed, so that they too can combine their matrices with the parent matrix.**
5. **Process the child frame of this frame. Pass in the new combined matrix that was used to render the mesh(es) at this frame. This effectively steps us down into the next level of the hierarchy for children of the current frame where the world space position generated for the current frame, becomes the frame of reference that any child frames have been defined relative too.**

Hopefully the concept of a spatial hierarchy now makes more sense to you. It is vital that you understand how these relationships work because we will encounter them again and again in graphics programming projects. Not only will we be dealing with spatial hierarchies here in this chapter, but we will see them again in the next two chapters and indeed throughout this entire series of courses. Make sure that you are comfortable with how a hierarchy works in theory as well as how to traverse and render one. In summary, we can think of a spatial hierarchy as a nested set of local space coordinate systems, where each local system is defined relative to the parent system it is attached to one level up in the tree. Also, do not forget that changes to a frame's matrix will automatically influence all of the world matrices generated for its children, since they are defined in its local space.

9.2 Frame Animation

To apply animation to a given frame, the most common approach is to assign it a unique **animation controller**. The animation controller's job is to ensure that the frame matrix to which it is attached is properly updated for the frame (i.e. it *controls* how the frame animation proceeds). Note that animations are generally going to be locally applied to a given frame (which in turns affects any children of course). In this section we will compose some simple code to help examine this process. To be clear, this will all be for illustration only -- we will not use any of this code in our applications. We will examine animation in much more detail in the next chapter, but for now, let us just get a high level understanding of the process since it illustrates the need for frame hierarchies and lets us see how they work.

Let us imagine that we wanted to rotate the four wheels of our example car by some fixed amount over time. We can do this by periodically applying transformations to the matrices of the wheel frames. In our example, each wheel frame requires local rotation updates at regular intervals and thus four animation controllers will be needed; one to manage each frame matrix. For the sake of this discussion, let us call this animation controller type a CAnimation. We will see later how this fits in with the D3DX

framework we will be using and thus we are going to get used to this naming convention early on. Our animation controller class (CAnimation) in this case could be very simple since it need only rotate the wheels about their center points.

```
class CAnimation
{
public:
    CFrame *pAttachedFrame;
    void Rotate ( float Degrees ) ;
};
```

Our CAnimation object is tasked with applying rotations to a single frame matrix. It contains a pointer to a frame in the hierarchy (i.e, the frame which it should animate) and a method that will build a rotation matrix to concatenate with the frame matrix whenever it is called. Calling this function multiple times will allow us to accumulate rotations for the frame by a specified number of degrees.

After the frame hierarchy is loaded at application startup and stored in memory as a tree of D3DXFRAME structures (we will see how to do this later in the chapter), we can call a function that searches through the hierarchy for the wheel frames and creates a CAnimation instance for each one. Again, our CAnimation object is a controller, attached to a single frame and used to manipulate its matrix to animate the frame (and thus its children). In a moment we will see how a global function might be constructed to traverse the frame hierarchy searching for the wheel frames and create new CAnimation objects which will be attached to each of them. For each animation, we will set its pAttachedFrame pointer to point at the wheel frame it is intended to animate.

> **Note:** When we use the D3DX library later to load a frame hierarchy from an X file, we will see that each frame will include a string storing its name. This is typically assigned to the frame by the artist or modeler who created the X file. This allows us to locate a frame in the hierarchy by searching for its name. In our hierarchy we can imagine that frames 1, 2, 3 and 4 would have the names 'Wheel 1', 'Wheel 2' , 'Wheel 3' and 'Wheel 4'. In the example code that follows we use the _tcsstr function to search for all frames that have the words 'Wheel' in their frame name.

As our four CAnimation objects will, in effect, combine to create a set of animation data for the automobile we will also write a simple container class that will handle the task of maintaining the list of animations. We will appropriately call this container a CAnimationSet. In our example, we will need to add four CAnimation objects to our CAnimationSet because we wish to animate the four wheels of our automobile (thus we wish four animations to be played).

```
class CAnimationSet
{
 public:
   CAnimation   **m_ppAnimations;
   DWORD          m_nNumberOfAnimations;

   void AddAnimation ( CAnimation pAnimation );
};
```

Once the X file data is loaded into the hierarchy we will call our CreateAnimations function (see below) to search for the four wheel frames and create, attach, and store our CAnimation objects. CreateAnimations would be recursive of course and need only be called once by our application. The

application will pass in a pointer to the root frame of the hierarchy to start the process, along with a pointer to an empty CAnimationSet object. Each CAnimation created by this function will be added to the CAnimationSet object.

```
void CreateAnimations ( CFrame *pFrame , CAnimationSet *pAnimationSet )
{
    CAnimation *pAnimation = NULL;
    if ( !pFrame )  return;
    if (  tcsstr ( pFrame->pFrameName , "Wheel" )
    {
       pAnimation = new CAnimation;
       pAnimation->pAttachedFrame = pFrame;
       pAnimationSet->AddAnimation ( pAnimation );
    }

    if ( pFrame->pSiblingFrame ) CreateAnimations (pFrame->pSiblingFrame);
    if ( pFrame->pChildFrame   ) CreateAnimations (pFrame->pChildFrame);
}
```

When the above function returns, the application's CAnimationSet object will have an array of four pointers to CAnimation objects (one for each wheel frame) and its m_nNumberOfAnimations member will be set to 4. Notice that the CreateAnimations function adds a newly created animation to the animation set using the CAnimationSet::AddAnimation member function. We will not show any code for this function but we can imagine that it might be a simple housekeeping function that resizes the m_ppAnimations array and adds the passed CAnimation pointer to the end of it.

Once our main game loop is running, we will need to animate and render the hierarchy. The animation update might take place in a separate application-level animation function (ex. AnimateObjects) or it could take place as we traverse the hierarchy for rendering. If we were going to use the latter approach, then each frame would have to know about its assigned controller so that it could run its update on the spot, prior to concatenating with its parent matrix. If we are going to do the animation updates before the hierarchy traversal, then this is not necessary. For example we could write a function that loops through all of the animations in the animation set and calls their Rotate functions like so:

```
void AnimateObjects ( float AngleInDegrees )
{
      for ( DWORD i = 0 ; i < pAnimationSet->m_nNumberOfAnimations; i++ )
      {
            CAnimation *pAnimation = pAnimationSet->m_ppAnimations[i];
            pAnimation->Rotate( AngleInDegrees );
      }
}
```

This function would ensure that all of the frame matrices were updated by their attached animation controllers prior to the hierarchy traversal. The CAnimation::Rotate method would be responsible for applying a local rotation for the frame to which it is attached. Again, this will ultimately wind up rotating all child frames and child meshes as well, once we do our hierarchy traversal.

In our example it is assumed that each wheel mesh has been defined in its own model space such that its center is at the origin and it is facing down the positive Z axis. We can think of the model space X axis

as being the axle on which the wheel rotates. Thus, we wish the animation object to rotate its attached wheel around the wheel frame's local X axis. In this case, we might imagine our CAnimation::Rotate method to look something like this.

```
void CAnimation::Rotate ( float Degrees )
{
   D3DXMATRIX  mtxRotX;
   D3DXMatrixRotationX( &mtxRotX , D3DXToRadian (Degrees) );
   D3DXMatrixMultiply(&pAttachedFrame->Matrix, &mtxRotX, &pAttachedFrame->Matrix);
}
```

As you can see, the function will first build an X axis rotation matrix. This will then be multiplied with the relative matrix of the attached frame, the result of which will replace this frame matrix. The order of the matrix multiplication is significant. We wish to rotate the wheel about its own X axis and then combine it with the current matrix of the frame so that the rotated wheel is then translated out to its position relative to the parent frame *after* being rotated. Therefore, we do (RotMatrix * FrameMatrix) so that the final matrix first rotates any children around their local X axis and then positions them relative to the parent frame of reference.

In our current example, each time the Rotate function is called, the matrices belonging to the wheel frames will store cumulative rotations. That is, the rotation is permanent and the frame matrix contents are directly modified (not done in a temporary matrix) by the attached CAnimation object. This may or may not be desirable depending on the circumstances, but it illustrates the point and works fine.

Rendering the newly animated hierarchy is no different than the code we saw previously. All our CAnimation objects have done is modified the relative matrix stored in the frame for which each is attached. We still start at the root and work our way down through the levels, combining the matrices of each frame with their parent. When we reach the frame for wheel 1, the parent matrix will be combined with the newly rotated frame matrix to create the frame world matrix and the wheel will be rendered. It will still be in its correct position relative to the car (wherever we may have moved the root frame to in the world) but will have been successfully rotated about its own center point.

Again you are reminded that these localized rotations affect all children for a frame – meshes and other child frames alike. For example, imagine that our car representation has another frame containing the mesh of a wheel hubcap attached to wheel 1 as a child (Fig 9.2).

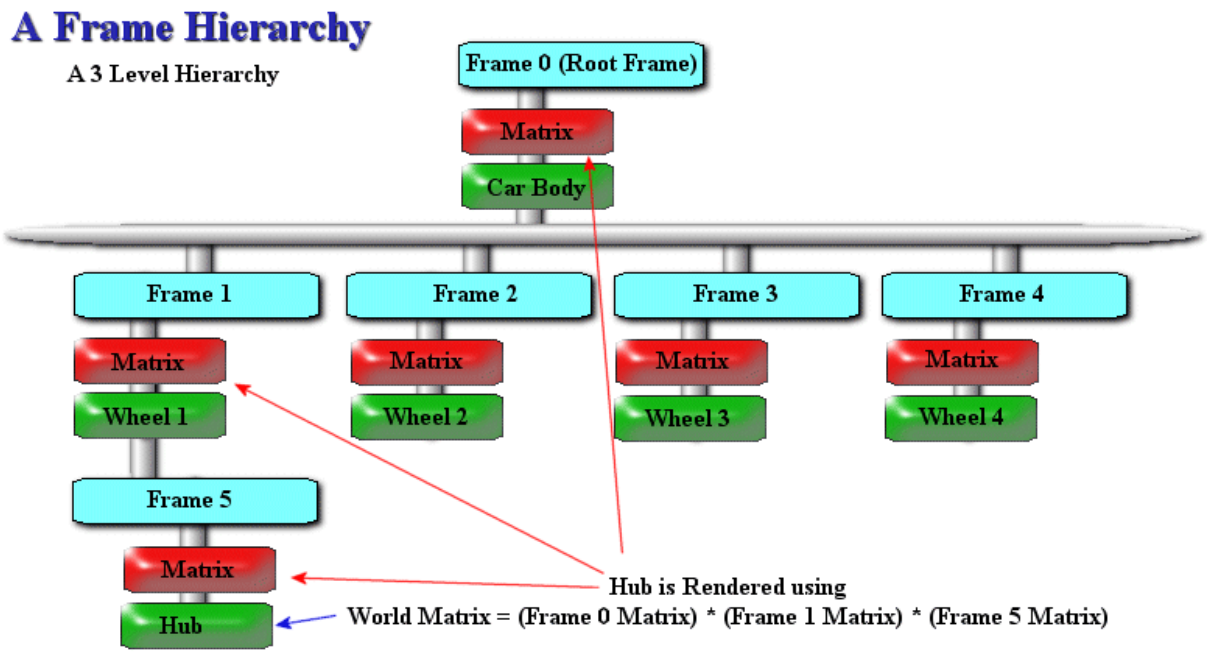


Figure 9.2

When we arrive at Frame 5 in our tree, the DrawHierarchy function will have been passed a parent matrix from the previous recursion (when processing Frame 1). That matrix will actually be the root frame matrix combined with the Frame 1 matrix. If the animation attached to Frame 1 rotated its assigned frame matrix, then this rotation will be propagated down to the hubcap frame as well and the hubcap will rotate along with its parent. This is as it should be of course since the hubcap is defined in the (now modified) local space of its parent.

As we will see later in this course, it is precisely this characteristic of a hierarchy that makes them perfect candidates for modeling articulated structures like game characters. It is easy to see the hierarchical relationship between the parts of the human body. The upper arm (origin = shoulder) would have the forearm (origin = elbow) as a child frame. The forearm would have the hand (origin = wrist) as its child frame and the hand would have five finger frame children (which might themselves be jointed and linked in a similar manner). If we wanted the model to raise its arm in the air by rotating about the shoulder, we could simply have a CAnimation object attached to the upper arm frame generate a local rotation matrix and use it to transform the frame (by multiplying it with the upper arm frame matrix). As a result, all of the children will move as expected since the rotation is propagated down through the hierarchy during our tree traversal.

So in summary, we now understand what a hierarchy is and we should be to see how an artist might design a scene or an object, not as a single mesh, but as a number of different meshes which are all spatially positioned and oriented relative to one another. These meshes are stored in frames, where each frame will have a transformation matrix that will not represent an absolute world transformation but a relative transformation describing the relationship of its own local system with its parent frame. The exception to the rule is the root frame which initially has no parent. The root frame matrix can instead be thought of as storing the core model space coordinate system (i.e. the base reference frame) from which

the rest of the hierarchy proceeds. This means that it can serve as a world space transformation for the entire hierarchy if it takes on values other than identity.

We also know that when we render the hierarchy we must process each frame, combining its matrix with the current combined matrix containing the concatenation of all the higher level frames in the hierarchy linked via its parent. We examined how to render the siblings and child frame objects for each frame using the parent matrix and the combined frame matrix respectively. Finally, we learned that because the matrices for our frames are not absolute, we can very easily animate individual frames in the hierarchy in their own local space and watch those animations propagate down through the subsequent levels of the tree. We used the concept of animation objects (CAnimation) to handle the animation (matrix updating) of the frame to which it is attached.

Finally, we discussed that a set of frame animations could be intuitively packaged together to represent an animation set. While the discussion of CAnimation and CAnimationSet classes may have seemed trivial since the animation data could be contained in any arbitrary structures, getting used to the idea of how these objects work with the frame hierarchy will greatly help us in understanding the loading of multi-mesh X files. It will also help us understand the animation system that D3DX provides to assist with the animation of frame hierarchies.

Unfortunately, when we load an X file using D3DXLoadMeshFromX or its sibling functions, any hierarchy information stored in the file is discarded. Instead, the function traverses the hierarchy in the file, applying the relative transformations so that all of the vertices in all of the meshes in the file are now in their model space positions. At this point, the mesh information has now been correctly transformed and collapsed into a single mesh object which is returned to the caller. So in the case of our automobile hierarchy, D3DXLoadMeshFromX would correctly offset each of the wheels from the car body mesh, but then collapse it all into a single mesh object, discarding the entire hierarchy. While the single mesh will look correct when not in motion, because the wheels are not separate meshes, we have lost the ability to animate them independently from the rest of the car.

Not to worry though, we do have some solutions at our disposal that allow for preservation of the hierarchy. As this chapter progresses, we will begin to examine some of our options. To get things started, the best place to begin is by looking more closely at the way our data is going to be stored. So in the next few sections we will examine X files in some detail.

9.3 X File Templates

Before DirectX 9.0, hierarchical scenes stored in X files had to be loaded and parsed manually using a set of packaged COM objects. These COM objects still exist today and they can continue to be used to parse X files, but as of DirectX 9.0, the D3DX library contains a function called D3DXLoadMeshHierarchyFromX that automates the entire process. The complete hierarchy is loaded (all child frames, meshes, textures, materials, and so on) in such a way that it still affords our application complete control over how the hierarchy data is allocated in memory. This level of insulation is certainly welcome and it will save us a good deal of coding and debugging, but it is still helpful to be at least somewhat familiar with how the data is stored in the X file and how it can be extracted. That is the purpose of this section.

The X file format is essentially a collection of templates. A **template** is a data type definition, very similar to a C structure, describing how the data for an object of a given type is stored in the file. Just as a C structure describes how variables of that type will have their member variables arranged in memory, an X file template describes how data objects with the same name (and GUID) will have their data laid out in the file. Once we know how to inform DirectX about the components of a template, the X file loading functions will recognize objects of that type in the file, know exactly how much memory is required, and understand how to load the data. The X file format is actually similar to the IWF format in many ways. We can think of a data object in the X file as the equivalent of the IWF chunk. Just as there are many different chunk types in the IWF format for representing different types of information, there are different data objects in an X file represented as templates. IWF chunks can also have child chunks, just as X file templates can have child templates for data objects.

For example, the X file Frame template can have many different types of child templates: the TransformationMatrix data type (which is the matrix for a frame in the hierarchy) and the Mesh data type (of which there can be many) are common examples. The Frame can also have other Frames as children, which we now know is the means by which a hierarchy is constructed.

As in the IWF format, you can also create custom data templates that are understood by your own application, but are not a part of the X file standard. For example, you could have a PersonalDetails template which contains a name, address, and telephone number, turning the X file into a standard address book (not that we would likely do such a thing). When the D3DXLoadMeshHierarchyFromX function encounters a custom data object, it will pass the custom data to your application using something similar to a callback function. This gives the application a chance to process that data.

Fortunately, we will not have to create our own templates to load a hierarchical X file. When using the D3DX library supplied X file loading functions, the standard templates are registered automatically. We will be studying some of the standard templates in this lesson to get a feel for how a hierarchy is laid out in the X file, but for details on all standard templates (Table 9.1), look at the SDK documentation in the section *DirectXGraphics / Reference / X File Reference / X File Format Reference / X File Templates*.

Table 9.1: X File Standard Templates

• Animation • AnimationKey • AnimationOptions • AnimationSet • AnimTicksPerSecond • Boolean • Boolean2d • ColorRGB • ColorRGBA • Coords2d • DeclData • EffectDWord • EffectFloats • EffectInstance • EffectParamDWord • EffectParamFloats • EffectString • FaceAdjacency • FloatKeys • Frame • FrameTransformMatrix • FVFData • IndexedColor	• Material • Matrix4x4 • Mesh • MeshFace • MeshFaceWraps • MeshMaterialList • MeshNormals • MeshTextureCoords • MeshVertexColors • Patch • PatchMesh • PatchMesh9 • PMAttributeRange • PMInfo • PMVSplitRecord • SkinWeights • TextureFilename • TimedFloatKeys • Vector • VertexDuplicationIndices • VertexElement • XSkinMeshHeader

Note: We will not need to register these standard templates with DirectX. In fact, even if we use custom objects in our X file which are not part of the standard, we do not have to register those either. We can simply store the template definitions at the head of the X file itself (just underneath the standard X file header) and the Direct X loading functions will automatically register them for us before it parses the file. More on this momentarily…

9.3.1 X File Components

At the top of every X file is a small line of text which looks something like this:

Xof 0303txt 0032

This value is written to the file by the modeling application and serves as a signature to identify the file to the loading application as a valid X file. The ‘Xof’ portion identifies it as an X file. The ‘0303’ text tells us that this X file was built using the X file version 3.3 templates. Following the version number is ‘txt’, which tells us that this file is a text file. X files can also be stored in binary format, and if this is the case then we would see ‘bin’ instead. Finally, the file signature line ends with ‘0032’ which tells us the file uses 32-bit floating point values instead of 64-bit floats.

We never have to read or parse this line manually as it is handled by the D3DX loading functions and the appropriate action will be taken. The following example shows the signature of a binary X file created using the version 3.2 templates with 64-bit floating point precision:

Xof 0302bin 0064

Following the signature there may be a list of template definitions that describe the data format for objects found in the file. This will typically be the case only if the X file uses custom data objects, but it is possible that some X files may include template definitions that are part of the standard.

X files contain three types of data objects: **top level**, **child**, and **child data reference**. A data object is a packet of information stored in the format of a registered template. Let us have a look at a very simple X file to see what a data object is and how it is stored. This example will use a common top level data object: a Mesh. The Mesh will contain a child data object for defining a Material. First we will look at the DirectX standard templates for both a Mesh and a Material.

```
template Mesh
{
        <3D82AB44-62DA-11cf-AB39-0020AF71E433>
        DWORD   nVertices;
        array   Vector vertices[nVertices];
        DWORD   nFaces;
        array   MeshFace faces[nFaces];
        [ ... ]                         // An open template
}
```

The template above is of type ‘Mesh’ and this is the name we would use to define instances of this object in the X file. The first member of any template is its GUID. Every template must include a GUID so that it can be uniquely identified. This prevents name mangling when a custom template shares the same name as another custom template or one of the standard templates. Keep this requirement in mind if you develop your own custom templates.

You can generate a GUID using the ‘Guidgen.exe’ tool that ships with the Microsoft visual C/C++ compiler. You can find it in the ‘\Common\Tools’ directory of your MSVC installation. Fig 9.3 shows the tool in action.

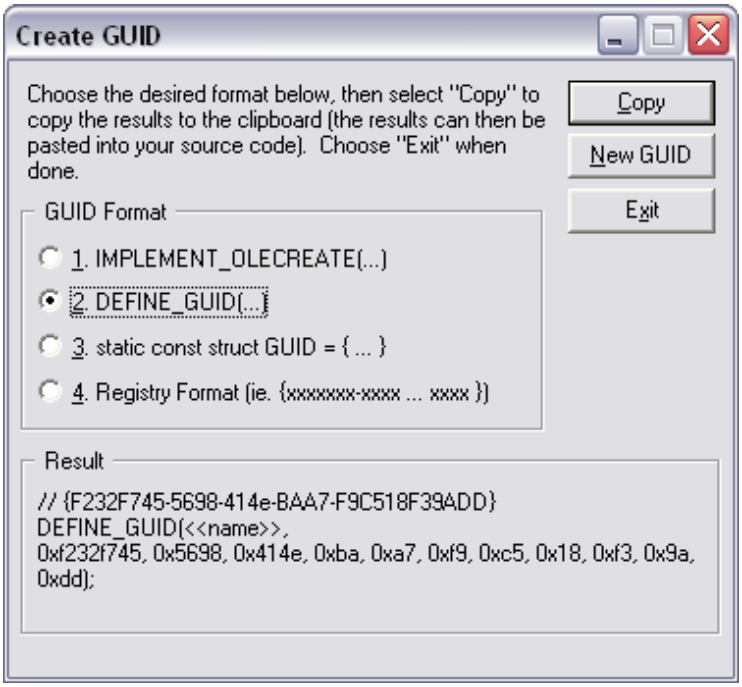


Figure 9.3

Guidgen uses a combination of parameters (hardware serial numbers, current date/time, etc.) to generate an ID that is guaranteed to be unique throughout the world. If for some reason you do not like the GUID it generates, you can simply hit the 'New GUID' button to build another one. In Fig 9.3 the second radio button selected helps format the GUID in the result window so that you can cut and paste it into an application. If we were to generate a GUID for a custom template called Contact, we would include it in the definition like so:

```
template Contact
{
     < F232F745-5698-414e-BAA7-F9C518F39AD >
     STRING Name;
     STRING Address1;
     STRING Address2;
     STRING PhoneNumber;
     DWORD Age;
}
```

Here we have simply copied over the top line of the Guidgen result pane and removed the two forward slashes and replaced the curly braces with chevrons.

When we use the D3DXLoadMeshHierarchyFromX function to load an X file, all standard templates will be recognized by their GUIDs and loaded on our behalf. In the case of a custom data object, something more akin to a callback function will be used instead. The application defined callback function (which will be triggered by D3DX when a custom template is encountered) will check the GUID that is passed and determine whether or not it matches a custom template the application would like to support. The application can skip this data if it does not match.

The DEFINE_GUID macro stores GUID information in a structure with a name that you can assign so that you can do fast GUID comparison tests. For our Contact template example above:

```
// {F232F745-5698-414e-BAA7-F9C518F39ADD}
DEFINE_GUID(ID_Contact, 0xf232f745, 0x5698, 0x414e, 0xba, 0xa7, 0xf9, 0xc5, 0x18,
0xf3, 0x9a, 0xdd);
```

This might look like a complex way to assign a name to a GUID at first, but if you examine the Guidgen result window, you will see that the code is the same. We have simply substituted ID_Contact in the <<name>> section.

With this alias defined, our application callback mechanism can do a quick test to determine whether to process a data object passed from a D3DX loading function:

```
if( *pDATAGUID == ID_Contact ){ // Process the objects data here }
```

Again, for standard templates, these steps are unnecessary. All standard templates such as the Mesh template we are discussing will be identified automatically by D3DXLoadMeshHierarchyFromX and loaded on our behalf.

Let us take another look at the standard Mesh template and continue our discussion of its members.

```
template Mesh
{
        <3D82AB44-62DA-11cf-AB39-0020AF71E433>
        DWORD   nVertices;
        array Vector    vertices[nVertices];
        DWORD   nFaces;
        array MeshFace  faces[nFaces];
        [ ... ]                         // An open template
}
```

The basic building blocks for a template are the data types that it stores. For example, we see that the first data member after the GUID in this template is a DWORD which describes how many vertices are in the mesh. Table 9.2 presents a list of all of the basic data types you can use in a template.

Table 9.2: Template Data Types

Type	Size
WORD	16 bit integer
DWORD	32 bit integer
FLOAT	32 bit floating point value
DOUBLE	64 bit floating point value
CHAR	8 bit char
UCHAR	8 bit unsigned char
BYTE	8 bit unsigned char
STRING	NULL terminated string

We can also specify arrays of the supported types or even other templates using the **array** keyword. A good example is the array of Vector objects in the mesh which contains one Vector for each mesh vertex. The Vector template is another standard template:

```
template Vector
{
   < 3D82AB5E-62DA-11cf-AB39-0020AF71E433 >
    float x;
    float y;
    float z;
}
```

The preceding nVertices DWORD (in the Mesh template) describes the number of items in the vector array that follows. This is the pattern that will be followed when arrays are used in a template. In this example it describes the number of vertices the Mesh object contains followed by an array of that many Vector data objects.

Following the Mesh vertex array is another DWORD that describes how many faces are in the Mesh. That value is followed by an array of MeshFace objects (another standard template).

```
template MeshFace
{
   < 3D82AB5F-62DA-11cf-AB39-0020AF71E433 >
    DWORD nFaceVertexIndices;
    array DWORD faceVertexIndices[nFaceVertexIndices];
}
```

This object type describes a single face in a mesh. The first member is a DWORD describing how many vertices the face uses (3 for a triangle, 4 for a quad, etc.) and it is followed by an array of that many DWORDs. Each DWORD in the array is the **index** of a vertex in the Vector array just discussed.

All of these components describe the Mesh object in its most basic form and every Mesh object must have the previously discussed components (a vertex and face list). At the bottom of the Mesh template we have a set of square brackets with ellipsis in between. This indicates that the Mesh template is an *open template* format that can include any type and any number of additional data objects.

9.3.2 Open, Closed, and Restricted Templates

Some templates will include child objects, and others will not. Templates that cannot include children are considered **closed**. This is the default template type. Conversely, templates that allow for any child type and number are called **open** templates. The Mesh template is an example of an open template. The third type of template can have child objects, but only of specific types specified by the template. These are called **restricted** templates and they must specifically list the templates that are legal child objects. A restricted template can have any number of the data objects specified by the template as embedded child data objects.

When a template is closed it cannot have any children. If children are somehow included in a closed template, they will be ignored by the loading code. The following example will demonstrate what this means. We start with two example templates:

```
template Matrix4x4 {
  <DED0E885-ED92-4224-A090-E81FC8AB2B83>
  array FLOAT Matrix[4][4];
}

template TestTemplate {
  <A96F6E5D-04C4-49C8-8113-B5485E04A866>
  DWORD Flags;
}
```

At the moment, these are closed templates. But we can certainly imagine a situation where an object like the following could be useful:

```
TestTemplate ChildTest {
  0;                    // Flags
  Matrix4x4 MyMatrix    // Matrix
  {
      1.0; 0.0; 0.0; 0.0;
      0.0; 1.0; 0.0; 0.0;
      0.0; 0.0; 1.0; 0.0;
      0.0; 0.0; 0.0; 1.0;;
  }
}
```

Given the previous template definitions, this is not legal. Technically we can do it, but the D3DX loading functions would completely ignore the matrix data because the TestTemplate definition described a closed data object which cannot have child objects (the Matrix4x4 object above) embedded. However, if we were to use ellipsis in the template definition to open the template, our usage above becomes completely legal:

```
template TestTemplate {
  <A96F6E5D-04C4-49C8-8113-B5485E04A866>
  DWORD Flags;
  [ ... ]
}
```

You may have noticed the additional semi-colon at the end of the child matrix data object in the above example. This was not a typo -- it signifies the end of a child object within a parent data object. Therefore, we have a semi-colon separating each member of the matrix plus an additional semi-colon denoting the end of the matrix data object within the outer data object. There is some helpful discussion of the usage of commas and semi-colons in the SDK documentation.

The use of the ellipsis is much like their use in C/C++: they indicate that some variable set of data can be included. To restrict the legal child types for a template (therefore creating a restricted template type), we simply include the list of types within the square brackets:

```
template TestTemplate {
  <A96F6E5D-04C4-49C8-8113-B5485E04A866>
  DWORD Flags;
  [ Matrix4x4, Vector3 ]
}
```

The comma separated list for a restricted template can include type names with optional GUIDs:

```
[ Matrix4x4 <DED0E885-ED92-4224-A090-E81FC8AB2B83>, Vector3 ]
```

Using a GUID is not required, but it can be a good idea to avoid ambiguity if new templates are introduced that cause a name clash. It also provides the ability to overload templates with the same name and restrict children to a specific type.

Describing the X file format layout is not easy and often is best accomplished with a simple example of an actual X file.

X File Sample 1: Basic Mesh

Here we see the contents of an actual X file that contains a single top level data object (a Mesh). Notice how the data is laid out in the exact format described by the Mesh template discussed earlier.

```
Xof 0303txt 0032

mesh{
  6;               // Number of vertices
 -5.0;-5.0;0.0;,   // Vertex list ( Vector array )
 -5.0;5.0;0.0;,
  5.0;5.0;0.0;,
  5.0;-5.0;0.0;,
 -5.0;5.0;-5.0;,
 -5.0;5.0;5.0;;

  2;               // Number of faces
  3;0,1,2;,        // MeshFace Array
  3;3,4,5;;
}
```

This example contains no color information or vertex normals, but it does define a basic mesh and provide the core requirements discussed previously.

Since the Mesh template is open, we can embed any child data objects we wish, although only a certain subset of child data objects will be understood by the D3DX loading functions. So let us take a look at another standard X file template that defines a material just like the D3DMATERIAL9 structure we looked at in Chapter Five of Module I.

```
template Material
{
  < 3D82AB4D-62DA-11cf-AB39-0020AF71E433 >
    ColorRGBA  faceColor;
    float      power;
    ColorRGB   specularColor;
    ColorRGB   emissiveColor;
}
```

Notice that the standard Material template does not include an Ambient member. We actually mentioned this in passing in Chapter Eight, and now we see this to be true at the low level. The D3DXLoadMeshFromX function could not return ambient information for each material used by the mesh being loaded because the material data objects used to describe mesh materials in the X file have no ambient member.

The faceColor member of the Material template is equivalent to the Diffuse color member of the D3DMATERIAL9 structure. This is represented using the standard template ColorRGBA seen next. The specular and emissive members use the standard template ColorRGB, also shown next. We are already familiar with how the DirectX lighting pipeline deals with materials and, with the exception of the missing ambient component, the X file material data object describes material properties in the same way as its D3DMATERIAL9 counterpart. It is the data stored within these data objects that the D3DXLoadMeshFromX function returns in its D3DXMATERIAL array.

The ColorRGBA Template	The ColorRGB Template

`template ColorRGBA`
`{`
`<35FF44E0-6C7C-11cf-8F52-0040333594A3>`
`  float red;`
`  float green;`
`  float blue;`
`  float alpha;`
`}`
	`template ColorRGB`
	`{`
	`  <D3E16E81-7835-11cf-8F52-0040333594A3>`
	`  float red;`
	`  float green;`
	`  float blue;`
	`}`

We will now expand our X file example by adding a material data object as a child of the mesh.

X File Sample 2: Basic Mesh with Material

```
Xof 0303txt 0032

mesh{
    6;              // number of vertices
   -5.0;-5.0;0.0;,  // vertex list
   -5.0;5.0;0.0;,
    5.0;5.0;0.0;,
    5.0;-5.0;0.0;,
   -5.0;5.0;-5.0;,
   -5.0;5.0;5.0;;

    2;             // number of faces
    3;0,1,2;,      // face list
```

```
    3;3,4,5;;

    Material {      // child data object of mesh
      1.000000;1.000000;1.000000;1.000000;;
      0.000000;
      0.000000;0.000000;0.000000;;
      0.000000;0.000000;0.000000;;
    }
}
```

Here we see a top level object (the Mesh) with one child (the Material). The mesh is considered a top level object because it has no parent. That is to say, it is not embedded in any other data object.

Defining materials as direct children of objects like this means that this material object belongs to, and can only be referenced by, this mesh. While this is sometimes the case, most of the time X files will represent materials (and other children) as **data reference objects**. A data reference object is used to save space. If the X file contained five meshes that all required the same material, we can save space by creating the material once at the top of the X file as a top level data object and assigning it a name (a reference). All meshes that use the material can use the name as a reference, much like the concept of instances we looked at in previous lessons. This is obviously much more memory space efficient than defining the same material data object in each of the five separate meshes.

This next example uses the same basic X file, but this time the material is defined as a top level object and assigned a name so that any number of meshes in the file would be able to reference it.

X File Sample 2: Basic Mesh with Material Reference

```
Xof 0303txt 0032

Material WhiteMat {     // This is a Data Object
   1.000000;1.000000;1.000000;1.000000;;
   0.000000;
   0.000000;0.000000;0.000000;;
   0.000000;0.000000;0.000000;;
}

mesh{                   // This is a Data Object
    6;                  // number of vertices
   -5.0;-5.0;0.0;,      // vertex list
   -5.0;5.0;0.0;,
    5.0;5.0;0.0;,
    5.0;-5.0;0.0;,
   -5.0;5.0;-5.0;,
   -5.0;5.0;5.0;;
    2;             // number of faces
    3;0,1,2;,      // face list
    3;3,4,5;;

   {WhiteMat}      // Data Reference Object as a child of the mesh Data Object
}
```

We can use any top level data object as a child of another object (assuming we have given the data object a name) by placing the name in between two curly braces. In the previous example, the line '{WhiteMat}' signifies that WhiteMat is the first material used by this mesh. D3DX supplied loaders automatically handle cases for both child objects and child reference objects. However, the same rules still apply for closed and restricted templates. References will only be valid in open templates or if explicitly included in the type list for a restricted template.

In the previous examples, we saw both a material data object and a material data reference object being embedded as a direct child of the mesh object. While this was a nice illustration of the concept, most meshes will not assume either form. Meshes will often use multiple materials (recall the subset discussion from Chapter Eight), and they will need to be mapped to faces in the mesh. So instead, there will be a single material list data object stored inside each mesh and the material objects (or material references) will be stored inside this object. The MeshMaterialList standard template contains a list of materials used by the mesh as well as the face mapping information:

```
template MeshMaterialList
{
    < F6F23F42-7686-11cf-8F52-0040333594A3 >
        DWORD nMaterials;
        DWORD nFaceIndexes;
        Array DWORD FaceIndexes;
       [Material]
}
```

The first DWORD describes how many Material data objects are embedded as child data objects or child data reference objects of this material list object. This is followed by the number of faces the material list affects. This will usually be equal to the number of faces in the entire face list of the mesh unless there is only one material and then nFaceIndexes=1 (indicating that this single material should be applied to all faces in the mesh). Following this is an array of DWORDS describing which material is applied to which face. Each element in this array is a zero-based index describing a material from a list of child materials that will be embedded in this object. This is a restricted template that can only have Material objects as children.

X File Sample 3: Basic Mesh with Reference Based MeshMaterialList

Our next example will embed a MeshMaterialList in a Mesh object and the material list will have two Material children. We can see the way that X files work using a hierarchy of sorts, even when it is not a spatial one. Also note that the Mesh and the Materials are top level data objects and that we are **referencing** the materials from within the MeshMaterialList.

```
Xof 0303txt 0032

Material WhiteMat{  // Top Level Data Object
   1.000000;1.000000;1.000000;1.000000;;
   0.000000;
   0.000000;0.000000;0.000000;;
   1.000000;0.000000;0.000000;;
}
Material RedMat{    // Top Level Data Object
   0.000000;0.000000;1.000000;1.000000;;
```

```
   0.000000;
   0.000000;0.000000;0.000000;;
   0.000000;0.000000;1.000000;;
}
mesh{                // Top Level Data Object
   6;                // Number of vertices
  -5.0;-5.0;0.0;,    // Vertex list
  -5.0;5.0;0.0;,
   5.0;5.0;0.0;,
   5.0;-5.0;0.0;,
  -5.0;5.0;-5.0;,
  -5.0;5.0;5.0;;
   2;                // Number of faces
   3;0,1,2;,       // Face list
   3;3,4,5;;       // Second Face

   MeshMaterialList {    // child data object of Mesh
     2;                  // Two materials in the list
     2;                  // Two faces in the following material face list

     0,                  // the first face uses Material 0
     1;;                 // the second face uses Material 1

     {WhiteMat}
     {RedMat}

   }// End Material list

}// end of mesh object
```

As we can see here, the mesh has as a child data object: the MeshMaterialList data object. As specified by the template, its first value describes the number of materials that will be assigned to this list (2). This is followed by the number of faces in the mesh which will have materials assigned to them (two in this example). Following this information is the index data that describes for each face, which material it uses in the list that follows. The first face index is zero which describes the first face in the mesh as having 'WhiteMat' (Material 0) assigned to it. The second face index is a value of 1 meaning it has been assigned the second material (Material 1) in the list 'RedMat'. Following the index list is the list of materials belonging to the material list and for which the indices above it are relevant. In this example, the materials are referenced and are actually defined as top level objects at the head of the file. This will allow any other meshes which may also be represented in the file to reference these same materials in their material list data objects, thus saving storage space.

X File Sample 4: Basic Mesh with Reference Based MeshMaterialList

For the sake of completeness, let us see the same example again, but with the materials embedded in the MeshMaterialList data object rather than treated as top level data objects.

```
Xof 0303txt 0032

mesh{
 6;                // number of vertices
-5.0;-5.0;0.0;,    // vertex list
```

```
-5.0;5.0;0.0;,
 5.0;5.0;0.0;,
 5.0;-5.0;0.0;,
-5.0;5.0;-5.0;,
-5.0;5.0;5.0;;
 2;          // number of faces
 3;0,1,2;,  // face list
 3;3,4,5;;


 MeshMaterialList
 {    // child data object of Mesh
   2; // Two materials in the list
   2; // Two faces in the following material face list

       // material face list
   0,  // the first face uses Material 0
   1;; // the second face uses Material 1

   Material { // Material 0,child data object of MaterialList
   1.000000;1.000000;1.000000;1.000000;;
   0.000000;
   0.000000;0.000000;0.000000;;
   0.000000;0.000000;0.000000;;
   }
   Material { // Material 1, child data object of MaterialList
   1.000000;0.000000;0.000000;1.000000;;
   0.000000;
   0.000000;0.000000;0.000000;;
   0.000000;0.000000;0.000000;;
   }
 } // end material list

}// end of mesh object
```

The current example is a fairly complete X file. Indeed, if your meshes do not contain any textures, texture coordinates, or vertex normals, then your meshes might look very similar to this. You should try opening up some text based X files and examining them to get a better understanding of the format and the way that custom data objects have their templates listed at the top of the file.

To continue, let us look at how texture information would be stored for a mesh in an X file. We know from our last chapter that texture pixel data is not stored inside the X file (although you could create a custom template to store image data directly), only the file name is stored. Recall that D3DXLoadMeshFromX passed back a buffer of D3DXMATERIAL structures, where each element contained a D3DMATERIAL9 and a texture file name. We can assume then that in X files, texture and material data can somehow be paired together, and this is indeed the case -- instead of each face having a separate material and texture index, the Material object we just studied will now contain a TextureFilename child object. Material objects can still exist on their own without texture filename objects inside them (as seen previously), but often, a Material will describe a material/texture combination used by faces in the mesh.

The TextureFilename data object is a simple one that contains a string describing the name of the texture image file.

```
template TextureFilename
{
  < A42790E1-7810-11cf-8F52-0040333594A3 >
    string filename;
}
```

Let us now update our example to see what our mesh would look like in X file format if we now used materials that also referenced textures.

X File Sample 5: Materials storing Texture Filenames

```
Xof 0303txt 0032

Material BrickMat{            // Top Level Data Object
1.000000;1.000000;1.000000;1.000000;;
0.000000;
0.000000;0.000000;0.000000;;
1.000000;0.000000;0.000000;;
Texture Filename {"BrickWall.bmp";}
}

Material WaterMat{            // Top Level Data Object
0.000000;0.000000;1.000000;1.000000;;
0.000000;
0.000000;0.000000;0.000000;;
0.000000;0.000000;1.000000;;
TextureFilename {"WaterTexture.bmp";}
}

mesh{             // Top Level Data Object
  6;              // Number of vertices
 -5.0;-5.0;0.0;,  // Vertex list
 -5.0;5.0;0.0;,
  5.0;5.0;0.0;,
  5.0;-5.0;0.0;,
 -5.0;5.0;-5.0;,
 -5.0;5.0;5.0;;
  2;        // Number of faces
  3;0,1,2;, // Face list
  3;3,4,5;;

  MeshMaterialList {// Child data object of Mesh
   2;               // Two materials in the list
   2;               // Two faces in the following material face list

   0,               // the first face uses Material 0
   1;;              // the second face uses Material 1

   {WallMat}
   {WaterMat}

  }// End Material list
```

```
}// end of mesh object
```

When we look at the Material data objects which contain both color and texture information, we understand that it is exactly this information which the D3DXLoadMeshFromX function returns to our application in the D3DXMATERIAL array. Each element in this array describes the texture and material combination for a subset of faces in the mesh.

As we learned in Chapter Five, we will need vertex normals if we intend to use the DirectX lighting pipeline. While we know how to generate vertex normals manually with the D3DXComputeNormals function (after we have cloned the mesh to include space for normals), it is certainly nice when the X file can provide those normals directly. If a level editor or 3D modeling application exports vertex normal information, then this is often preferred because it gives the artist the ability to edit those normals to achieve custom lighting effects. Storing vertex normals in the X file is easy enough with the standard X File template called MeshNormals:

```
template MeshNormals
{
   < F6F23F43-7686-11cf-8F52-0040333594A3 >
     DWORD nNormals;
     array Vector normals[nNormals];
     DWORD nFaceNormals;
     array MeshFace meshFaces[nFaceNormals];
}
```

The first DWORD (nNormals) describes the number of normals in the normals array (an array of Vectors) that will follow. There are typically as many vertex normals in this list as there are vertices in the mesh, but sometimes a mesh may choose to save space by assigning the same normal to multiple vertices. If this is not the case, we can just read the normals in and assume that there is a one-to-one mapping of normals to vertices.

If the normal count is different from the number of vertices in the parent mesh vertex list, then this means that the normals are mapped to the vertices in the mesh using indices, similar to the face list. When this is the case, the bottom two members of this template are used. The DWORD nFaceNormals will equal the number of faces in the mesh, and thus the MeshFace array. Each MeshFace object in the array holds indices into the normal list for each vertex in the face. For example, if the first MeshFace contains the indices 10,20,30, then the first vertex of the face uses normal [10] in the normal list, the second vertex uses normal [20] and the third vertex uses normal [30]. More often than not, meshes contain per-vertex normals rather than the MeshFace index scheme and the last two members of this template are not used.

The next X file example adds vertex normals as a new child object for the Mesh.

X File Sample 6: Mesh with Vertex Normals (List Version)

```
Xof 0303txt 0032

mesh{
 6;               // number of vertices
```

```
-5.0;-5.0;0.0;,   // vertex list
-5.0;5.0;0.0;,
 5.0;5.0;0.0;,
 5.0;-5.0;0.0;,
-5.0;5.0;-5.0;,
-5.0;5.0;5.0;;
 2;          // number of faces
 3;0,1,2;,   // face list
 3;3,4,5;;

 MeshMaterialList {// child data object of Mesh
  2; //Two materials in the list
  2; //Two faces in the following material face list

      // material face list
  0,  // the first face uses Material 0
  1;; // the second face uses Material 1

  Material {            // Material 0,child data object of MaterialList
  1.000000;1.000000;1.000000;1.000000;;
  0.000000;
  0.000000;0.000000;0.000000;;
  0.000000;0.000000;0.000000;;
  TextureFilename {"example.bmp";} // child data object of Material
  }

  Material {           // Material 1, child data object of MaterialList
  1.000000;0.000000;0.000000;1.000000;;
  0.000000;
  0.000000;0.000000;0.000000;;
  0.000000;0.000000;0.000000;;
  }
 }

 MeshNormals {                // child data object of Mesh object
 6;                           // one for each vertex
 0.000000;1.000000;0.000000;, // Normal 1
 0.000000;1.000000;0.000000;, // Normal 2
 0.000000;1.000000;0.000000;, // Normal 3....
 0.000000;1.000000;0.000000;,
 0.287348;0.957826;0.000000;,
 0.287348;0.957826;0.000000;;
 2;               // number of faces
 3; 0,2,1;,       // indices of how the normals are mapped to each face
 3; 3,4,5;;
 }

}// end of mesh object
```

Notice that the MeshFace type contains the number of indices and then the indices themselves. In this particular example, the MeshFace list is not necessary since the number of vertices equals the number of normals, but we added the mapping list just to demonstrate the concept. Notice also that the MeshNormals data object is embedded as a child of the mesh object. This makes sense as this object contains the vertex normal information for that mesh.

If you take a look at the 'Mesh Normals' object above, you will see that the first value is 6, which indicates that there are six normals in the list that follows. This simple mesh only has two faces comprised of six vertices in total. After this normal count there follows a list of the six normals. Typically, this would be all the information stored as we have six vertices and six vertex normals and that is all we need to know. For completeness however, in this particular example we demonstrate how there may not be a one-to-one mapping between normals in the normal list and vertices in the vertex list. Following the normal list you can see a value of 2, which means that following this value will be an array of 'Mesh Face' objects used to describe how the normals in the normal list map to vertices in the mesh vertex list. We can see that the first face in this list has 3 indices (which happen to map to the first face in the mesh) and has indices 0, 2 and 1. This means the first vertex of the first face in the mesh should use the first normal in the normal list (normal[0]), the second vertex of the first face of the mesh should use normal[2] from the normal list and the third vertex in the first face of the mesh should use normal[1]. We can also see that the second 'Mesh Face' object has three indices and that the first, second and third vertices of this face use the fourth, fifth and sixth normals in the normal list respectively.

While we have seen how to pair a texture and a material together and assign the combination to faces in the mesh, there is currently no information in our X File example describing how the texture of a given material data object is mapped to a face. Thus, we need a way to store per vertex texture coordinates for the mesh. We do this by embedding another child object, called MeshTextureCoords, inside the mesh object. This standard template is shown below:

```
template MeshTextureCoords
{
  < F6F23F40-7686-11cf-8F52-0040333594A3 >
    DWORD nTextureCoords;
    array coords2d textureCoords[nTextureCoords] ;
}
```

The DWORD describes the number of texture coordinates in the following array of coord2d objects. Typically there will be a set of texture coordinates for each vertex in the mesh vertex list (provided the mesh is textured) and this is always a 1:1 mapping. For example, the first texture coordinate in the array is the texture coordinate set for the first vertex in the mesh vertex list, and so on. While there may be fewer texture coordinates in this list than there are vertices in the mesh, this will still be a 1:1 mapping. If a mesh has 50 vertices, but only the first 30 belong to textured faces and needed texture coordinates, only 30 texture coordinate sets might exist in this array, but they will map to the first 30 vertices in the parent mesh vertex list.

Each pair of texture coordinates in the MeshTextureCoords array are represented as the standard template Coords2d, shown below:

```
template Coords2d
{
   < F6F23F44-7686-11cf-8F52-0040333594A3 >
     float u;
     float v;
}
```

Finally, let's take a look at our X file example with texture coordinates added.

X File Sample 7: Mesh with Texture Coordinates

```
Xof 0303txt 0032

mesh{
 6;               // number of vertices
-5.0;-5.0;0.0;,   // vertex list
-5.0;5.0;0.0;,
 5.0;5.0;0.0;,
 5.0;-5.0;0.0;,
-5.0;5.0;-5.0;,
-5.0;5.0;5.0;;
 2;          // number of faces
 3;0,1,2;,   // face list
 3;3,4,5;;

 MeshMaterialList {// child data object of Mesh
  2; //Two materials in the list
  2; //Two faces in the following material face list

      // material face list
  0,  // the first face uses Material 0
  1;; // the second face uses Material 1

  Material {            // Material 0,child data object of MaterialList
  1.000000;1.000000;1.000000;1.000000;;
  0.000000;
  0.000000;0.000000;0.000000;;
  0.000000;0.000000;0.000000;;
  TextureFilename {"example.bmp";} // child data object of Material
  }

  Material {            // Material 1, child data object of MaterialList
  1.000000;0.000000;0.000000;1.000000;;
  0.000000;
  0.000000;0.000000;0.000000;;
  0.000000;0.000000;0.000000;;
  }
 }

 MeshNormals {                // child data object of Mesh object
 6;                           // one for each vertex
 0.000000;1.000000;0.000000;, // Normal 1
 0.000000;1.000000;0.000000;, // Normal 2
 0.000000;1.000000;0.000000;, // Normal 3....
 0.000000;1.000000;0.000000;,
 0.287348;0.957826;0.000000;,
 0.287348;0.957826;0.000000;;
 2;               // number of faces
 3;0,2,1;,        // indices of how the normals are mapped to each face
 3;3,4,5;;
 }
```

```
 MeshTextureCoords {// child data object of mesh object
 6;                 // number of Texture coordinates one for each vertex

 0.308594;0.152344;,    // texture coordinate 1
 0.007813;0.152344;,    // texture coordinate 2 ....
 0.007813;0.007813;,
 0.308594;0.007813;,
 0.328125;0.156250;,
 0.415681;0.156250;;
 }

}// end of mesh object
```

> **NOTE:** You may be wondering how to assign multiple textures (or materials) to a single face. Unfortunately, this is not possible using the standard templates. That is, multi-texturing is not supported as part of the X file standard, so custom templates would be required to remedy this situation.

There are still many standard templates we have yet to discuss. Some will be discussed later in the chapter, others in the next chapter. But for now, we have a good overall idea of how the X file format works. For a complete list of templates, please be sure to check the X file reference in the SDK documentation. The information we have just discussed will be very useful in the future, especially if you wish to read or write your own X file custom data objects. We do not, however, need to be fully versed in the X file format or its various templates to be able load a scene hierarchy. We demonstrated in Chapter 8 that we did not need to be aware of the way a mesh was laid out in an X file in order to load the mesh using the D3DXLoadMeshFromX function. Nevertheless, this X file discussion has set the stage for our return to frame hierarchies, which we will revisit in the next section in terms of their X file representation.

9.3.3 Hierarchical X Files and Frames

Our focus in this chapter will continue to be on scenes or objects that consist of multiple meshes. When this is the case, the X file will be stored as a frame hierarchy. As already discussed, the hierarchy is constructed using Frames, where a frame takes the form of a tree node in terms of implementation. From the perspective of an X file, these frames will include a transformation matrix stored as a child data object as well as any number of child mesh objects and/or child frame objects. When an X file contains many meshes, they will not generally be defined as top level objects. They are instead stored as frame child objects. It is typical for the X file to contain a single top level Frame (the root frame) and have all other meshes and frames defined either directly or indirectly as children of this root frame. The standard templates for defining hierarchies are Frame and FrameTransformMatrix.

```
template Frame
{
  < 3D82AB46-62DA-11cf-AB39-0020AF71E433 >
  […]
}
```

Frame is an open template that will usually have a FrameTransformMatrix, one or more Mesh objects, and/or other Frames as children. The D3DX loading functions D3DXLoadMeshHierarchyFromX and

the simpler D3DXLoadMeshFromX understand Mesh, FrameTransformMatrix, and Frame template instances as child objects when loading a Frame. These are usually the only objects stored in a frame.

```
template FrameTransformMatrix
{
   < F6F23F41-7686-11cf-8F52-0040333594A3 >
    Matrix4x4 frameMatrix;
}
```

FrameTransformMatrix contains a single member, a Matrix4x4 object, which is another standard template. It is an array of 16 floats describing each element of a 4x4 matrix in row major order:

```
template Matrix4x4
{
   < F6F23F45-7686-11cf-8F52-0040333594A3 >
   array float matrix[16];
}
```

With this information in mind, we now have a complete picture of the form a hierarchy takes inside an X file. The following example will use the automobile hierarchy discussed earlier, but we will leave out the bulk of the mesh data because we already know what mesh data in an X file looks like.. For now, just assume that the mesh data would be contained within the curly braces. Fig 9.4 is a reminder of the automobile hierarchy introduced earlier.

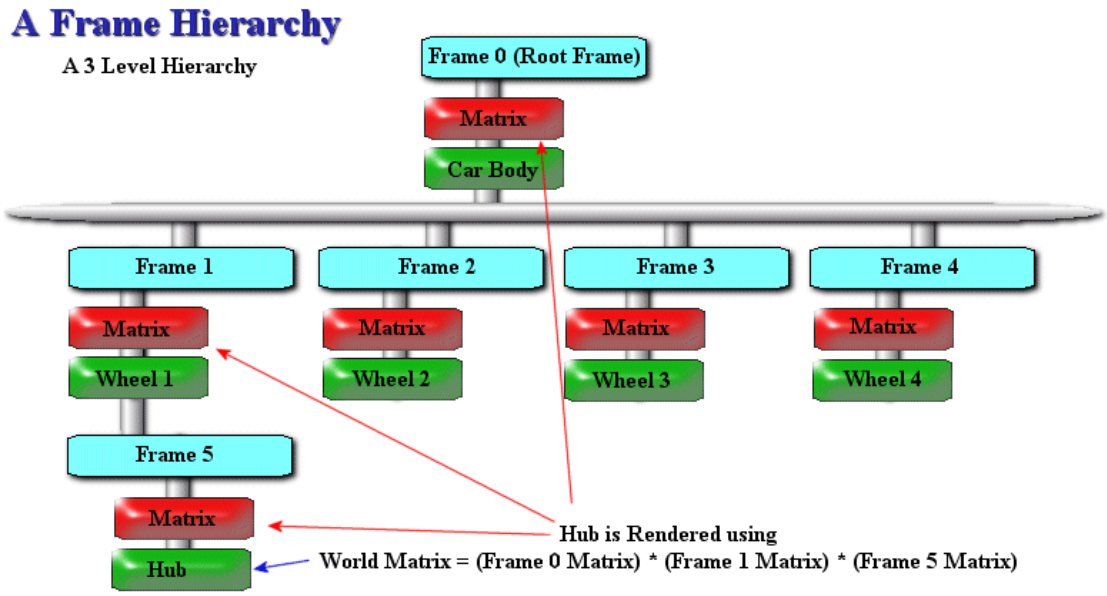


Figure 9.4

The wheel frames will be offset from the root frame at distances of 10 units along the X and Z axes. The root frame has a matrix which sets the object at position (50, 0, 50). Usually, the root frame matrix will either not exist or will be an identity matrix, allowing us to place the entire hierarchy where we want in the world by setting the world matrix and then rendering. However, in this example we will use a root

frame that is offset from the world space origin so that we can see the various matrices being stored in the X file.

Matrix Translation Vectors (Root Matrix now contains position 50,0,50)

Root Matrix	=	**50 , 0 , 50**	(The World Matrix of the Hierarchy)
Wheel 1 Matrix	=	**-10 , 0 , 10**	(RootMatrix * Wheel1Matrix = 40 , 0 , 60)
Wheel 2 Matrix	=	**10 , 0 , 10**	(RootMatrix * Wheel2Matrix = 60 , 0 , 60)
Wheel 3 Matrix	=	**-10 , 0 , -10**	(RootMatrix * Wheel3Matrix = 40 , 0 , 40)
Wheel 4 Matrix	=	**10 , 0 , -10**	(RootMatrix * Wheel4Matrix = 60 , 0 , 40)
Hub Matrix	=	**-5 , 0 , 10**	(Root Matrix * Wheel1Matrix * Hub Matrix = 35,0,10)

And finally, here is the X file that represents everything that we have learned thus far. You should be able to see how the X file is laid out exactly as described by the hierarchy diagram and how the matrices contained in the X file hierarchy are not world transform matrices but are instead relative matrices. Recall that they describe the position of a frame using the parent frame of reference.

X File Sample 8: Hierarchical X File

```
Xof 0303txt 0032

Frame Root
{
  FrameTransformMatrix
  {
    1.000000, 0.000000,  0.000000, 0.000000,
    0.000000, 1.000000,  0.000000, 0.000000,
    0.000000, 0.000000,  1.000000, 0.000000,
   50.000000, 0.000000, 50.000000, 1.000000;;
  }

  Mesh CarBody { mesh data would go in here }

  Frame Wheel1                // Child frame of Root
  {
    FrameTransformMatrix
    {
     1.000000,  0.000000, 0.000000,  0.000000,
     0.000000,  1.000000, 0.000000,  0.000000,
     0.000000,  0.000000, 1.000000,  0.000000,
    -10.000000, 0.000000, 10.000000, 1.000000;;
    }

    Mesh Wheel1 { mesh data would go in here }

    Frame Hub                 // Child frame of Wheel 1
    {
      FrameTransformMatrix
      {
       1.000000, 0.000000, 0.000000, 0.000000,
       0.000000, 1.000000, 0.000000, 0.000000,
       0.000000, 0.000000, 1.000000, 0.000000,
      -5.000000, 0.000000, 0.000000, 1.000000;;
```

```
      }

      Mesh Hub { Hub mesh data goes here }

    } // end child frame Hub

  } // end child frame Wheel 1

  Frame Wheel2                // Child frame of Root
  {
    FrameTransformMatrix
    {
     1.000000, 0.000000, 0.000000,   0.000000,
     0.000000, 1.000000, 0.000000,   0.000000,
     0.000000, 0.000000, 1.000000,   0.000000,
     10.000000, 0.000000, 10.000000, 1.000000;;
   }

   Mesh Wheel2 { mesh data would go in here}

  } // end child Wheel 2

  Frame Wheel3                      // Child frame of Root
  {
    FrameTransformMatrix
    {
     1.000000, 0.000000,   0.000000, 0.000000,
     0.000000, 1.000000,   0.000000, 0.000000,
     0.000000, 0.000000,   1.000000, 0.000000,
   -10.000000, 0.000000, -10.000000, 1.000000;;
    }

    Mesh Wheel3 { mesh data would go in here}

  } // end child frame wheel 3

  Frame Wheel4                      // Child frame of Root
  {
    FrameTransformMatrix
    {
     1.000000,  0.000000,   0.000000,  0.000000,
     0.000000,  1.000000,   0.000000,  0.000000,
     0.000000,  0.000000,   1.000000,  0.000000,
     10.000000, 0.000000, -10.000000,  1.000000;;
    }

    Mesh Whee4 { mesh data would go in here}

  }// end child frame wheel 4

} // end root frame
```

We now have a very solid understanding of how a hierarchy is stored inside an X file. What we will find is that this information is going to be very helpful to us when working with the frame hierarchy after it is loaded. This will be the subject of the sections that follow.

9.3.4 D3DXLoadMeshFromX Revisited

In Chapter Eight we learned how to load single meshes from an X file without making any effort to understand the internals of the file format. However, at the outset of this chapter we stated our concern about the limited nature of the D3DXLoadMeshFromX function. While D3DXLoadMeshFromX will collapse all of the data into a single mesh, it must of course account for the fact that those meshes are individually defined in parent-relative space. So it will need to traverse the hierarchy and cumulatively transform all child meshes using their Frame (and parent Frame) transformation matrices, so that the vertices of the mesh are transformed into the shared hierarchy/scene space. This happens before adding that data to the new D3DXMesh object (which is ultimately returned).

Again, this may be quite acceptable for certain applications that do not require those components to be independently manipulated. This might be the case, using our automobile example, for parked cars that are used as detail objects in a scene. But if we do want to treat those meshes independently, another solution is needed.

There are two ways to extract this hierarchy data from an X file, one more complex than the other. The easiest approach is to use the D3DXLoadMeshHierarchyFromX function, and we will look at this function shortly. The more difficult approach is to use the X file COM objects mentioned earlier in the text and parse the file manually. We will briefly review the COM objects first since this approach provides a more 'behind the scenes' look at the process. There will also be times when you will need to use some of the COM objects alongside the D3DXLoadMeshHierarchyFromX function in order for your application to parse custom data objects. Additionally, you may decide to write your own X File parser, perhaps because you do not want to use the D3DX library functions or because you may not have access to them (or perhaps you wish to write an X File loading module that will compile and work with earlier versions of DirectX). We will only briefly examine these X file interfaces: just enough to provide a high level understanding of the process and to get you started should you decide to pursue the concept on your own at a later date. There is plenty of good reference material in the 'X File Reference' section in the SDK documentation as well, so be sure to consult that documentation as needed.

9.3.5 Loading Hierarchies Manually

DirectX provides several COM objects and interfaces to deal with the manual reading and writing of X files. In order to use these objects and interfaces, we need to include three header files that ship with the SDK. They are contained in the 'Include' directory of the DirectX 9 SDK installation:

1. **dxfile.h** This file contains all the COM interfaces that we need to use to manually process X files.
2. **rmxftmpl.h** This file contains the template definitions for all of the standard X file templates that we have looked at previously, such as 'Frame', 'Mesh' and 'MeshMaterialList' for example.
3. **rmxfguid.h** This file contains all the GUID aliases for the templates so that we can identify object types when processing X file data objects.

NOTE: The X file COM objects are not part of the core DirectX and D3DX libraries. You must make sure that if you intend to use the X File interfaces that you also link the 'd3dxof.lib' file into your project. It can be found in the 'Lib' directory of your DirectX 9 SDK installation directory.

9.3.5.1 The IDirectXFile Interface

The 'd3dxof.lib' library exposes a single global function call to create our initial interface. This interface will serve as the gateway to the rest of the X file interfaces from within our application. This is similar to our use of Direct3DCreate9 to create our initial IDirect3D9 interface, which then serves as a means to create an IDirect3DDevice9. We call the DirectXFileCreate global function with the address of an IDirectXFile interface pointer. On function return, this will point to a valid DirectXFile object.

```
IDirectXFile * pDXFile = NULL;
DirectXFileCreate ( &pDXFile );
```

This simple interface contains three methods. One of those methods handles saving data to an X file and we will not concern ourselves with this for the time being. For now we will focus on the other two methods which are used in the file loading process.

HRESULT RegisterTemplates(LPVOID *pvData*, DWORD *cbSize*);

This method is the primary means by which we inform DirectX of the templates that we wish to parse in the X file. Each template must be registered with the loading system, or else data objects of that type inside the X File will not be recognized later during parsing. The first parameter points to a buffer that contains the template definitions we wish to register and the second parameter indicates the size (in bytes) of this buffer.

All of the standard templates that we looked at earlier (as well as many others) are already contained in a buffer called `D3DRM_XTEMPLATES` that is defined in 'rmxftmpl.h'. This is why we included this header file. The same header file defines the size of this buffer (3278 bytes) and assigns this value to `D3DRM_XTEMPLATE_BYTES`. To register the all of the standard X file templates, we would use code like this:

```
IDirectXFile * pDXFile = NULL;
DirectXFileCreate ( &pDXFile );
pDXFile->RegisterTemplates( (VOID*) D3DRM_XTEMPLATES, D3DRM_XTEMPLATE_BYTES );
```

At this point, if we were not using custom templates we could begin parsing immediately. If the X file does contain custom data objects (i.e. data objects which are not part of the X file standard), then their templates will also need to be registered. We can build the template string manually if we wish and pass it into the function as shown next:

```
char *szTemplates = "xof 0303txt 0032\
               template Contact { \
                  <2b934580-9e9a-11cf-ab39-0020af71e433> \
                  STRING Name ;\
```

```
                  STRING Address1;\
                  STRING Address2;\
                  STRING PostCode;\
                  STRING Country;\
               } \
            template Hobbies  { \
                  <2b934581-9e9a-11cf-ab39-0020af71e433> \
                   DWORD nItems;\
                   array STRING HobbyNames;\
            }";

IDirectXFile * pDXFile = NULL;
DirectXFileCreate( &pDXFile );
pDXFile->RegisterTemplates( (void *) szTemplates , strlen ( szTemplates ) );
```

In this example we have registered two custom templates, called 'Contact' and 'Hobbies', which could be used to store personal information about a person. Once these templates have been registered, if data objects of either type are found in the X file, the loading system will process them as valid objects.

The RegisterTemplates function will search the passed buffer for valid templates and register them with the loading system. Recall from our earlier discussion that it is common practice to define templates at the top of the X file itself, before the actual data object definitions. If this is the case, we could instead load the entire X file into a memory buffer (just performing a block read) and then pass this buffer (containing all X file data) into the RegisterTemplates function. The function will find the templates listed at the top of the file and will automatically register them. This means you will not have to hardcode template definitions into your project code and that you can easily change the template definitions just by altering the templates in the X file. They will automatically be registered each time the application is run. If the X file contained template definitions for all the standard templates that it uses, then registering the standard templates as a separate step would also be unnecessary.

After we have registered any standard and/or custom templates we intend to use, it is time to call the second function of the IDirectXFile interface. This function, IDirectXFile::CreateEnumObject, opens the X file and creates a top level data enumerator object which is returned as a pointer to an IDirectXFileEnumObject interface. We can use this interface to step through each top level data object in the X file one at a time and extract their data.

```
HRESULT CreateEnumObject( LPVOID pvSource,
                          DXFILELOADOPTIONS dwLoadOptions,
                          LPDIRECTXFILEENUMOBJECT *ppEnumObj );
```

The first parameter is a pointer to a buffer whose meaning depends on whether we are loading the X file from disk, resource, or memory buffer. This is specified by the second parameter which should be one of the DXFILELOADOPTIONS flags:

DXFILELOAD_FROMFILE - If this flag is used for the second parameter, then the first parameter to the CreateEnumObject function is a string containing the file name of the X file.

DXFILELOAD_FROMRESOURCE - If this flag is used as the second parameter, the first parameter should be a DXFILELOADRESOURCE structure. For more details on this structure, see the SDK documentation as we will not be covering it here.

DXFILELOAD_FROMMEMORY - If this flag is used as the second parameter, the first parameter should be a DXFILELOADMEMORY structure. For more details on this structure, check out the SDK documentation as we will not be using it in this chapter.

The complete process for registering the standard templates and then creating our enumeration object to process X file data would be as follows:

```
IDirectXFile            *pDXFile     = NULL;
IDirectXFileEnumObject  *pEnumObject = NULL;
DirectXFileCreate ( &pDXFile );
pDXFile->RegisterTemplates( (VOID*) D3DRM XTEMPLATES ,
                                    D3DRM_XTEMPLATE_BYTES );

pDXFile->CreateEnumObject( "MyCube.x" , DXFILELOAD_FROMFILE, &pEnumObject );
```

> **Note:** This section is intended only as a brief introduction to the X file interfaces for those of you interested in parsing your X files manually. In this chapter, our demos will not be using these interfaces, and will instead be doing things the easy way using the D3DXLoadMeshHierarchyFromX function. This information is still useful though as later on we will discuss how we can use the D3DXLoadMeshHierarchyFromX function to parse custom X file templates. In that case, a thorough understanding of the X file format and the IDirectXFileData interface is necessary.

After the code shown above has been executed, the application has a pointer to an IDirectXFileEnumObject interface. This interface exposes methods which the application can use to start processing top level data objects stored in the X file.

9.3.5.2 The IDirectXFileEnumObject Interface

Once we have an IDirectXFileEnumObject interface, we can call its GetNextDataObject method to retrieve each top level data object stored in the X file, one at a time. Typically this will happen in a loop, where the data objects (returned to the application as DirectXFileData objects) are processed until no more top level data objects exist in the file.

For each top level data object that exists in the file (ex. Mesh, Frame, etc.), the data is returned to the application as an IDirectXFileData interface. The application can use the methods of this interface to extract the data as needed. Note that this function only returns top level objects, not child objects like, for example, a MeshMaterialList object which would be embedded as a child object inside a Mesh data object. Because all top level objects must be data objects, and not data reference objects, the enumerator can safely make this assumption.

The following code shows how we might set up a loop to process each top level object in the file.

```
LPDIRECTXFILE           pDXFile;
LPDIRECTXFILEENUMOBJECT pEnumObject;
LPDIRECTXFILEDATA       pFileData;
const GUID *pGuid;

DirectXFileCreate (&pDXFile);
pDXFile->RegisterTemplates( (VOID*) D3DRM_XTEMPLATES , D3DRM_XTEMPLATE_BYTES);
pDXFile->CreateEnumObject( "MyXFile.x" , DXFILELOAD_FROMFILE, & pEnumObject));

while ( SUCCEEDED ( pEnumObject->GetNextDataObject(&pFileData)))
{
     pFileData->GetType(&pGuid);
     if (*pGuid==TID_D3DRMMesh)     GetMeshData  ( pFileData );
     if (*pGuid==TID_D3DRMFrame)    GetFrameData ( pFileData );
     pFileData->Release();
}

pEnumObject->Release();
pDXFile->Release();
```

The code shows that once we have created the top level enumerator object, we repeatedly call its GetNextDataObject method to access the next top level data object in the X file. The actual data contained within the top level object is contained inside the returned IDirectXFileData interface.. Once we have an IDirectXFileData interface, we call its GetType function to test its GUID (i.e. the GUID of its template) against the ones we wish our application to support. In this example you can see that we are testing for top level Mesh and Frame objects only. The aliases for their GUIDs (and all of the standard template GUIDs) are listed in the 'rmxfguid.h' header file. TID_ is an abbreviation for Template ID.

Once we have identified a supported type in the above example code, we call some application defined functions to process the data stored in the returned IDirectXFileData object. In this example it is assumed that the GetMeshData and GetFrameData functions are application defined functions which will use the passed IDirectXFileData interface to extract the data into some meaningful structures (an ID3DXMesh for example). We will study the IDirectXFileData interface in a moment when we learn how to extract the data from a loaded data object.

There are three similar methods in the IDirectXFileEnumObject interface that we can use to access top level data objects defined in the X file. The first is called GetNextDataObject and is the one we have used in the above code example. All three data object retrieval methods of the enumerator interface are listed next:

```
HRESULT GetNextDataObject(LPDIRECTXFILEDATA *ppData);
HRESULT GetDataObjectByName(LPCSTR szName,LPDIRECTXFILEDATA *ppData);
HRESULT GetDataObjectById(REFGUID rguid, LPDIRECTXFILEDATA *ppData);
```

The **GetNextDataObject** function is called to fetch the next top level data object from the file. We saw this being used in the code example shown previously. The function is passed the address of an IDirectXFileData interface pointer which on function return will point to a valid interface which can be

used to extract the data from the data object. If the X file contains six top level data objects, this function would be called six times, each time fetching the next data object until it fails.

The **GetDataObjectByName** function lets us search the X file for an object with the name passed in. The second parameter is the address of an IDirectXFileData interface pointer which, if a top level object with a matching name exists in the file, will be used to extract data from the requested object. The example data object shown below, which is of type 'TestTemplate', would have the name 'MyFirstValues'. Fetching data objects by name is very useful if you know the name of a specific object you wish to load. This allows you to load this data object without having to step through all other top level data objects in the file.

The **GetDataObjectById** works in exactly the same way as the previous function, but searches on data object GUIDs rather than names. Although we have not mentioned it until now, each data object in an X file can also have its own **unique** GUID in addition to the template GUID. For example, let us say that we defined our data for a TestTemplate object as follows:

```
TestTemplate MyFirstValues {
  <78A5640B-1A66-4CD0-B1A4-3E2060429130>
  1.000000;
  2.000000;
  3.000000;
}
```

We have now added a GUID to the actual data object itself. The GUID that we included in the object above (remember this is not a template, but an actual instance of a TestTemplate object) can be used to identify this specific object in the file. Therefore, the template GUID can be used to identify all data objects of the same type in the X file and the data object GUID can be used to identify a single data object. This allows an application to find it easily even if the X file contains many data objects based on the same template.

9.3.5.3 The IDirectXFileData Interface

The IDirectXFileData interface encapsulates the reading and writing of X file data objects. This is true whether we are using the COM interfaces to manually parse the X file as discussed in the last section, or whether we use D3DXLoadMeshHierarchyFromX and custom data blocks have to be processed. Whether we intend to use the X File COM objects for loading X files manually, or intend to use the much easier D3DXLoadMeshHierarchyFromX function, exposure to this interface cannot be avoided. We will see shortly that when the D3DXLoadMeshHierarchyFromX function encounters a custom data object (for which D3DX has no knowledge of how to load), it will use a callback mechanism to send this data object to the application (as an IDirectXFileData). The application can then extract the data contained using the methods of this interface.

We saw earlier that when we use the IDirectXFileEnumObject interface to retrieve top level data objects in an X file, it actually returns this interface for each data object found. This interface encapsulates the data contained in the data object, such as the data for a mesh, frame or material object.

The IDirectXFileData interface exposes eight methods. Three of them are used primarily in the building and saving of X files and five are used for reading and processing the data stored in the X file data object. Since we are discussing loading X files at the moment, we will concentrate on these five methods for now. They are listed next with short descriptions. To make these functions easier to explain, we will use an example template and an object of that template type in our discussions:

```
template TestTemplate
{
  <A96F6E5D-04C4-49C8-8113-B5485E04A866>
  FLOAT TestFloat1;
  FLOAT TestFloat2;
  FLOAT TestFloat3;
}
```

This is a simple custom template with a GUID (which it must have) describing a data object that has three floats. Let us declare an instance of this template to work with:

```
TestTemplate MyFirstValues
{
   <78A5640B-1A66-4CD0-B1A4-3E2060429130>
  1.000000;
  2.000000;
  3.000000;
}
```

Note that this object also has its own GUID and no other object in the file will use this GUID. The template GUID describes the type of object this is (of which there may be many objects of this type in the file) and the object GUID identifies the exact instance of the template object. While per-object GUIDs are optional, it is a handy feature that you will no doubt see being used from time to time.

`HRESULT GetName( LPSTR pstrNameBuf, LPDWORD pdwBufLen );`

This function retrieves the name of the data object as specified in the X file. So in our example, because we specified "`TestTemplate MyFirstValues`" in the X file, the value returned would be "`MyFirstValues`". We are not required to specify a name when we instantiate an object in an X file; in fact, we saw examples earlier that simply instantiated an object using only a template name without assigning the object its own name. However, it can be handy to be able to reference the data later on elsewhere in the file, so it is good practice. Of course, this is often out of our hands as the X file will usually be created by some 3rd party modelling program for which we have no control over its X file export process. In the case of data reference objects, they naturally use per-object names so that they can be referenced by other data objects elsewhere in the file. Let us take a look at its two parameters.

LPTSTR pstrNameBuf
This first parameter is a pointer to a pre-initialised char buffer that should contain the resulting name on function return. It can be as large or small as you require. The name will be copied into this buffer.

LPDWORD pdwBufLen
This is a pointer to a `DWORD` which is used as both an 'In' and 'Out' parameter. The value, contained in the `DWORD` (that you pass in), must specify the maximum length of the string buffer that was passed in to

the previous parameter. When the function returns, this DWORD will be filled with the *actual* buffer length required to store the string, including the terminating character. If the original value you passed in specifies a value that is not large enough to return the entire string, then this function will return D3DXFILEERR_BADVALUE.

Passing NULL for the pstrNameBuf parameter will fill out the DWORD with the length of the buffer required to store the returned string (the complete name of the object). Note that the same rules still apply: the input value you pass in here must still be large enough (even if you do pass NULL as the string parameter). In both cases, an error will be returned if this is not the case. This allows you to test whether a certain buffer length will be large enough to store the returned string.

HRESULT GetType (const GUID **ppguid);

One of the first things we want to know about an IDirectXFileData interface is what type of data object it represents (Mesh, Frame, Material, etc.) so that we can select the appropriate processing path or determine if our application even wants to process it at all. GetType allows us to retrieve a pointer to the GUID of the template type for the data object. In our example, the GetType method would return the GUID of the TestTemplate type. We used this function earlier when enumerating through the file:

```
while ( SUCCEEDED ( pEnumObject->GetNextDataObject(&pFileData)))
{
     pFileData->GetType(&pGuid);
     if (*pGuid==TID_D3DRMMesh)  GetMeshData  ( pFileData );
     if (*pGuid==TID_D3DRMFrame) GetFrameData ( pFileData );
     pFileData->Release();
}
```

We pass in the address of a GUID pointer to retrieve a pointer to the GUID object itself. As you can see, this is a const pointer so we are not allowed to modify the GUID.

HRESULT GetId(LPGUID *pGuid*);

This method will retrieve the GUID for the object being referenced. Unlike the type GUID specified in a template declaration, this is the per-object GUID discussed previously. In our example object ('MyFirstValues'), we have included a GUID, so this is the value that will be retrieved here. We discussed earlier how to use the 'DEFINE_GUID' macro so that you can build aliases (such as TID_D3DRMMesh) for your own GUIDs and use them for type comparison.

LPGUID pGuid
This method takes a single parameter. A pointer to a GUID structure to be filled with the GUID data on function return. If no GUID exists in the data object, it will be filled with zeros.

HRESULT GetData(LPCSTR *szMember*, DWORD **pcbSize*, void ***ppvData*) ;

This is the primary method for retrieving the object data that was loaded from the file. If the data object represented by this interface was a Mesh object for example, the GetData method allows us to access the underlying vertex and index data stored within. It accepts three parameters which are described below.

LPCTSTR szMember

For this parameter, there are two options. The first option is to pass in the name of a member variable in the data object that you wish to retrieve the value for (ex. "`TestFloat1`" in our current example). This allows us to retrieve the variables of a data object one at a time, or get the values only for specific variables that we may be interested in. The second option is to pass NULL, which indicates that all of the data for the entire data object should be retrieved.

DWORD *pcbSize

On function return, this `DWORD` pointer will contain the total size of all of the data retrieved.

void **ppvData

As the third parameter, the application should pass the address of a void pointer. On function return this will point to the start of the data buffer we requested. Whether this buffer contains just a single member variable of the object or the entire data set itself depends on whether or not NULL was passed as the first parameter. The application never gets to own this data as it is issued a pointer to the API's own internal data. For this reason, we must copy the information pointed at into memory that our application does own so that we can modify it and rely on its persistence.

If we request a pointer for the entire data buffer, then the template itself should be our guide for determining the offsets for where each variable is stored in that buffer. Look again at our example template:

```
template TestTemplate
{
  <A96F6E5D-04C4-49C8-8113-B5485E04A866>
  FLOAT TestFloat1;
  FLOAT TestFloat2;
  FLOAT TestFloat3;
}
```

We can see that the buffer will contain three floats. Note that the GUID is *not* part of the data buffer, as there are separate methods for accessing the GUID as previously discussed. Using the above example, we know that the pointer would initially point to the float value 'TestFloat1'. Assuming 32-bit floating point precision is being used, we know that incrementing a BYTE pointer by four bytes would now point it at the value assigned to 'TestFloat2', and so on. Of course, the most common and easiest ways to access the data would be to either use a float pointer to step though the values with a simple pointer increment (for this example) or to build a three float structure that mirrors the data area and do a block copy. The returned void pointer could then be cast to a pointer of this type and the members accessed/copied intuitively. We might imagine we have defined a 3 float structure called 'MyXStruct'. We could then cast the returned void pointer and extract the data into three application owned variables (called foo1, foo2 and foo3) as shown below. vpPointer is assumed to be the void pointer to the data buffer returned by the GetData method:

```
MyXStruct *data = vpPointer;
foo1 = MyXStruct->TestFloat1;
foo2 = MyXStruct->TestFloat2;
foo3 = MyXStruct->TestFloat3;
```

This method is inadvisable however because the compiler may insert extra padding into your structures to make the members 32-bit aligned for increased performance. As this invisible padding is not inserted into the retrieved data object buffer, a mismatch can occur between the offsets of each member in the structure and their corresponding offsets in the data buffer. We will have more discussion about this problem a bit later.

The previous four methods of the IDirectXFileData interface that we have discussed demonstrate how to identify a data object and extract its data. We have discovered however, that a data object in an X file, like Meshes for example, may also contain child data objects. Thus, the IDirectXFileData interface may point to a data object that has child objects. We need a way to not only to process its data as previously discussed, but also a way to perform the same processing on any child objects it may contain. This is definitely necessary as the IDirectXFileEnumObject interface (which we have used for data object retrieval thus far), only retrieves top level data objects. This would leave us no way to access any embedded child objects should they exist.

The IDirectXFileData interface has a method that works very much like the IDirectXFileEnumObject::GetNextDataObject function. It allows us to step through its child objects one at a time retrieving IDirectXFileData interfaces for each of its children. This final method (called GetNextObject) provides the ability for our application to reach any object in a complete hierarchy of X file data objects. Not surprisingly, it is common for the loading of an X file to be done as a recursive process.

```
HRESULT GetNextObject( LPDIRECTXFILEOBJECT *ppChildObj );
```

The GetNextObject method allows us to iterate through all the child objects just as we did earlier for top level objects. We will continue to call this method until it returns `DXFILEERR_NOMOREOBJECTS`. Essentially, the interface maintains an internal 'current object' pointer so that it knows how far through the list it is. Note that this is a one-time only forward iteration process; once we have traversed the list, we cannot restart from the beginning.

This function has a single parameter:

LPDIRECTXFILEOBJECT *ppChildObj
This pointer will store the address of the current child to be processed. We might expect that the child data object would be returned via an IDirectXFileData interface (as with top level objects) but we are actually returned an IDirectXFileObject interface instead. This is because, unlike top level data objects which are always explicit data objects, child data objects may be specified as either actual data objects or data *reference* objects. You will recall from our earlier X file examples that we showed a top level material object being referenced using a child data reference object in a Mesh. The `IDirectXFileObject` interface is actually the base class for three derived classes: `IDirectXFileData` (which we are now familiar with), `IDirectXFileDataReference`, and `IDirectXFileBinary`. We are not told what type of derived interface this object is via this function, so we must `QueryInterface` the base pointer to find out. The GUIDs for the three types of derived interfaces that may be returned by this function are:

```
IID_IDirectXFileData
IID_IDirectXFileDataReference
IID_IDirectXFileBinary
```

If the IDirectXFileData::GetNextObject method returns a data reference object, we can determine this by calling QueryInterface on the base pointer and passing in IID_IDirectXFileDataReference as the interface GUID. If it is a data reference object, a valid IDirectXFileDataReference interface will be returned for the object. If we do find a child data reference object (instead of a normal child data object), we have to handle it slightly differently. A data reference object does not actually contain data. It is analogous to a pointer in many ways, pointing to a data object elsewhere in the file. We must use the IDirectXFileDataReference interface to resolve this reference and point us at the actual data object which contains the data we need to extract.

We have already discussed the first interface type (IDirectXFileData) and know how to extract the data for a child data object. Let us now look at the other two interfaces which may be returned.

9.3.5.4 The IDirectXFileDataReference Interface

While the top level enumerator will always return an IDirectXFileData interface (remember, top level objects cannot be data reference objects or binary objects) this is not the case for child objects. If the IDirectXFileData::GetNextObject function returns a data reference object, then we know that it does not contain data, but rather acts like a pointer to a data object elsewhere in the file. To access the actual data, we will use the following method:

```
HRESULT Resolve( LPDIRECTXFILEDATA * ppDataObj );
```

This function resolves the reference by returning an `IDirectXFileData` interface to the actual data of the object being referenced. From our application's perspective, from this point on, we are dealing with a normal data object. We can work with the IDirectXFileData interface to extract the data as discussed previously, knowing that it is a child of the outer data object.

There are two other methods exposed by this interface: GetID and GetName. They are identical to the calls of the same name that were discussed when we studied the IDirectXFileData interface (both classes derive them from the same base class).

9.3.5.5 The IDirectXFileBinary Interface

This interface allows us to retrieve any binary data objects that may be stored in the X file, even in cases where the file is a formatted text version. We will not be using this at interface at all, so be sure to study the SDK documentation for more information.

You now have more than enough information to get started on your own X file parser if you care to write one. Again, with DirectX 9, there is probably little need given the utility functions provided. But if this is so, you might be wondering why we just spent so much time examining the X file format in such detail. As it turns out, understanding the X file format is still very useful, even when using the D3DXLoadMeshHierarchyFromX function. This is especially true if we need to parse custom objects. Furthermore, if you wanted to write a level editor or an X file exporter in the future, perhaps as a plug-in or tool for a game that you were developing, you would want to have at least this level of familiarity with X files so that you can write your file data correctly. And not to be overlooked, now that you understand how to construct X files manually, you could build simple frame hierarchies using a text editor like Notepad in the absence of an available scene editor.

> **Note:** While we are not going to look at a complete source code example of loading an X file using the 'X File' interfaces discussed, the DirectX SDK ships with source code to a mesh loading class in CD3DFile.h and CD3DFile.cpp which you can use in your own applications. CD3Dfile.cpp can be found in the 'Samples\C++\Common\Src' folder of your DirectX 9 installation. Here you will see all of the interfaces we have discussed being used to manually load an X file and maintain its hierarchical structure.

9.4 D3DXLoadMeshHierachyFromX

D3DXLoadMeshHierarchyFromX is very similar to the D3DLoadMeshFromX function studied in the last chapter. The key difference is that it supports the loading of multiple meshes and creates an entire frame hierarchy in memory. While this function removes the burden of having to become familiar with the X file format and process them using the COM objects discussed in the last section, we still need to understand the X file interfaces if we wish to process custom objects. We will look at some custom objects later in the course, but for now we will concentrate on loading hierarchies that use only the standard templates.

As you can see in the declaration that follows, there are some new D3DX structures and classes we will need to learn about before we can use this function. Furthermore, we must also learn how to derive a COM interface so that we can provide the function with application-defined memory allocation routines for mesh and frame data.

```
HRESULT D3DXLoadMeshHierarchyFromX
(
    LPCTSTR                     Filename,
    DWORD                       MeshOptions,
    LPDIRECT3DDEVICE9           pDevice,
    LPD3DXALLOCATEHIERARCHY     pAlloc,
    LPD3DXLOADUSERDATA          pUserDataLoader,
    LPD3DXFRAME*                ppFrameHierarchy,
    LPD3DXANIMATIONCONTROLLER*  ppAnimController
);
```

We will discuss the parameter list slightly out of order so that we can have a better idea of how it all works.

LPCTSTR Filename
This is a string containing the name of the X file we wish to load. This X file may contain multiple meshes and/or a frame hierarchy. This function will always create at least one frame (the root frame), even if the X file contains only a single top level mesh and no actual frames. In such a case, the mesh will be an immediate child of the root frame (also returned by this function in the sixth parameter).

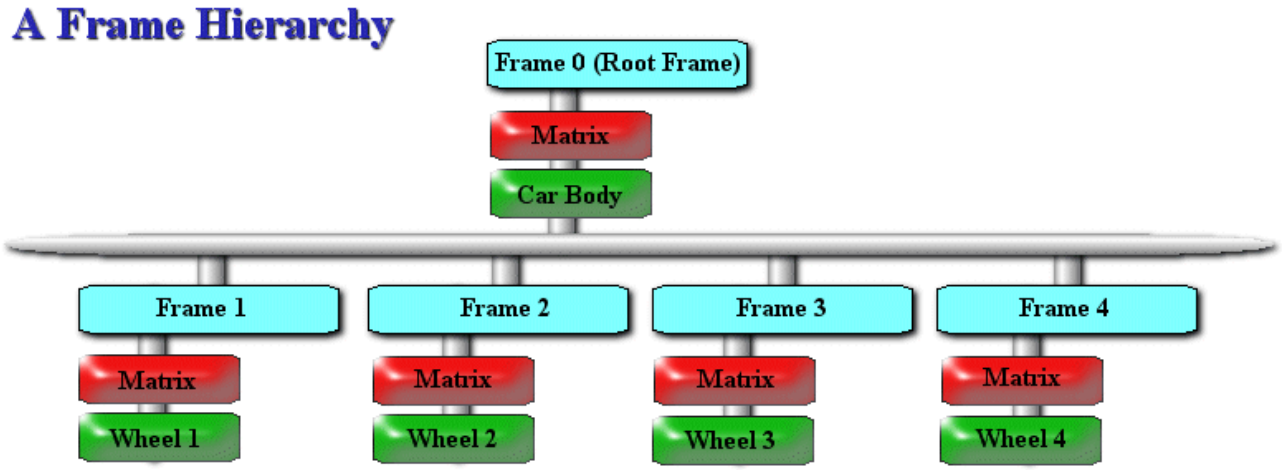
DWORD MeshOptions
These are the standard mesh options that we pass to mesh loading functions. We use a combination of zero or more members of the D3DXMESH enumerated type to specify the memory pools for mesh data, dynamic buffer status, etc.

Note that these flags may be ignored or altered by D3DX, and meshes may be created in a different format. Shortly we will see that during the loading of each mesh, we get a chance to test them (via a callback mechanism) for the desired options. If the mesh format that D3DX has chosen is not what we want, we can clone another mesh and then attach it to the parent frame instead.

LPDIRECT3DDEVICE9 pDevice
This is a pointer to the device that will own the resource memory.

LPD3DXFRAME *ppFrameHeirarchy
The sixth parameter to the D3DXLoadMeshHierachyFromX function is vitally important, since without it we would not be able to access the loaded hierarchy. We pass in the address of a pointer to a D3DXFRAME structure and on function return this will point to the root frame of the hierarchy. We can then use this pointer to access the root frame, its child frames and any child meshes it contains. This will be the doorway to our loaded hierarchy, allowing us to completely traverse the hierarchy to any level we desire.

The D3DXFRAME structure is used by D3DX to store the information for a single frame in the loaded hierarchy. A hierarchy will consist of many of these structures linked together. If you look at the following diagram of a hierarchical X file, followed by the D3DXFRAME structure definition, you will see that the structure itself mirrors the information that the diagram would suggest a frame would need to store.



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

We see that a frame includes a name as its first member. This can certainly be useful if needed to search for a specific component in your hierarchy or if you wanted to use it for display purposes.

The 4x4 D3DXMATRIX member stores the parent-relative transformation we discussed in the early part of the chapter. This is a relative matrix describing the position and orientation of the frame using its parent frame (if one exists) as its frame of reference.

The D3DXMESHCONTAINER structure stores a single set of mesh data (geometry, material, etc.) local to this node. It stores all the data for a single mesh. Note that we will see momentarily that this structure is capable of serving as a node in a linked list of mesh containers (via a pNext pointer) should multiple meshes actually need to be stored in a single frame in the hierarchy. The mesh container pointer in the D3DXFrame structure then is potentially the head of that linked list. It is more typical for a single mesh to be stored in a frame (if any) and therefore this pointer will often point to a single D3DXMESHCONTAINER structure whose next pointer is NULL. We will examine this structure in more detail in a moment.

The last two members are both pointers to other D3DXFRAME structures. The pFrameSibling pointer points to frame structures that are at the same level in the hierarchy as the current frame (i.e., they share the same parent frame). This pointer will be set to NULL for the root frame. If you recall our automobile example, all four of the wheel frames were child frames of the root frame and they are therefore sibling frames. The pFrameFirstChild pointer is essentially the head of a linked list of sibling frames. In the automobile example, the root frame pFrameFirstChild pointer would point at the first wheel frame. That wheel frame would then be linked together with its siblings using the pFrameSibling pointers. So the root frame (Frame 0) will have its pFrameFirstChild pointing at Frame 1. Frame 2 would be pointed to by Frame 1's pFrameSibling pointer. Frame 3 would be linked using Frame 2's pFrameSibling pointer, and so on. Thus the pFrameSibling pointer acts like the generic 'next' pointer in a standard linked list implementation. To traverse the hierarchy from the root down, we start at the pFrameFirstChild after processing the root, step through the sibling linked list, and then process the next set of children in the same fashion, and so on.

So, we have seen that when we use the D3DXLoadMeshHierarchyFromX function, D3DX will construct an entire hierarchy of D3DXFRAME structures in memory. The root frame is returned by the function which can then be used by the application to traverse and render the hierarchy. Wherever a mesh is attached to a frame in the hierarchy, the corresponding D3DXFRAME structure will have a valid pointer to a D3DXMESHCONTAINER structure. This structure contains all of the information for the mesh that D3DX has loaded and attached to the frame on our behalf. It also contains the actual ID3DXMesh interface for the mesh.

The D3DXMESHCONTAINER is defined as follows:

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

LPTSTR Name

Like a frame in the hierarchy, we can assign a name to the mesh stored in the container. This is almost always the name of the actual mesh (e.g. 'Door Hinge 01') that is assigned to the mesh inside the 3D modeller.

D3DXMESHDATA MeshData

This structure stores the mesh object that D3DX has loaded/created. It contains either a pointer to a regular ID3DXMesh, an ID3DXPMesh, or an ID3DXPatchMesh (which we will not discuss until later in the curriculum). Since the mesh can only be one of these types at any given moment, the pointers can share the same memory by specifying them as a union.

```
typedef struct _D3DXMESHDATA
{
   D3DXMESHDATATYPE     Type;
   union
   {
      LPD3DXMESH        pMesh;
      LPD3DXPMESH       pPMesh;
      LPD3DXPATCHMESH   pPatchMesh;
   } // End Union
} D3DXMESHDATA, *LPD3DXMESHDATA;
```

The Type member allows us to determine which of the three mesh pointers is active. It will be set to one of the following values:

```
D3DXMESHTYPE_MESH
D3DXMESHTYPE_PMESH
D3DXMESHTYPE_PATCHMESH
```

LPD3DXMATERIAL pMaterials

This is an array of `D3DXMATERIAL` structures which store the material and texture data used by the mesh object specified by the previous member. This is identical to the information returned by the D3DXLoadMeshFromX function. Each D3DXMATERIAL structure in this array describes the texture and material for the corresponding subset in the mesh. This array is unique to this mesh container and information is not shared between separate containers in the hierarchy. Therefore, we must make sure when processing this data that we do not load the same texture

multiple times. This is easy enough to address as D3DX will provide our application with the ability to process each mesh container before its gets added to the frame hierarchy. This is accomplished via a callback mechanism that D3DX uses to communicate with the application during the hierarchy construction process.

LPD3DXEFFECTINSTANCE pEffects
Just as the D3DXLoadMeshFromX function returned a buffer of effect instances (one for each subset in the mesh), this array represents the same information for the mesh attached to this mesh container. Effects will be covered in the next module in this series.

DWORD NumMaterials
This variable contains the number of subsets in the mesh attached to this mesh container. This tells us both the number of D3DXMATERIAL structures in the pMaterials array and the number of D3DXEFFECTINSTANCEs in the pEffects array.

DWORD *pAdjacency
This is the standard array describing the face adjacency within the mesh.

LPD3DXSKININFO pSkinInfo
This member stores the skin information for the mesh if the mesh uses the skinning technique for animation. We will cover skinning in detail when we discuss skinning and skeletal animation in Chapter 11. We will ignore this parameter at this time.

struct _D3DXMESHCONTAINER *pNextMeshContainer
The mesh container can act as a node in a linked list. This is a pointer to the next mesh container in the list should multiple meshes be assigned to a single frame in the hierarchy.

So we can see that a mesh container stores what is essentially all of the information that we retrieve in a single call to `D3DXLoadMeshFromX`. In effect, D3DX is doing exactly this, with the exception that instead of collapsing all of the mesh data into a single mesh object, this process is performed for each mesh in the hierarchy. Each mesh's materials, effects, adjacency, and other data is loaded into this container, almost as if we had loaded numerous separate single mesh .X files ourselves into a hierarchy of D3DXFRAME structures.

Fig 9.5 gives us another look at our simple automobile hierarchy, using the structures we have discussed thus far. This diagram demonstrates the hierarchy that would be created by the D3DXLoadMeshHierarchyFromX function if the X file being loaded contained our multi-mesh automobile. This is the version that has a hubcap mesh connected to the first wheel to better show a multi-level hierarchy. In the following example, no two meshes share the same immediate parent frame, but if multiple meshes were immediate children of the same parent frame, the parent frame would point to the first of these mesh containers, and then all the remaining mesh containers of this frame would be linked together (much like sibling frames) using their pNextMeshContainer pointers. Pay special attention to how the frames are linked together. Notice how frames 1-4 are linked by their sibling pointers. This entire row of children is attached to the parent via the root node's child pointer (points to frame 1). Also notice how each frame has a mesh container attached which describes all the information and contains all the data associated with that mesh. The root frame would be returned to the calling

application by the loading function. The application can use this root frame to traverse the hierarchy and access and render its mesh containers.

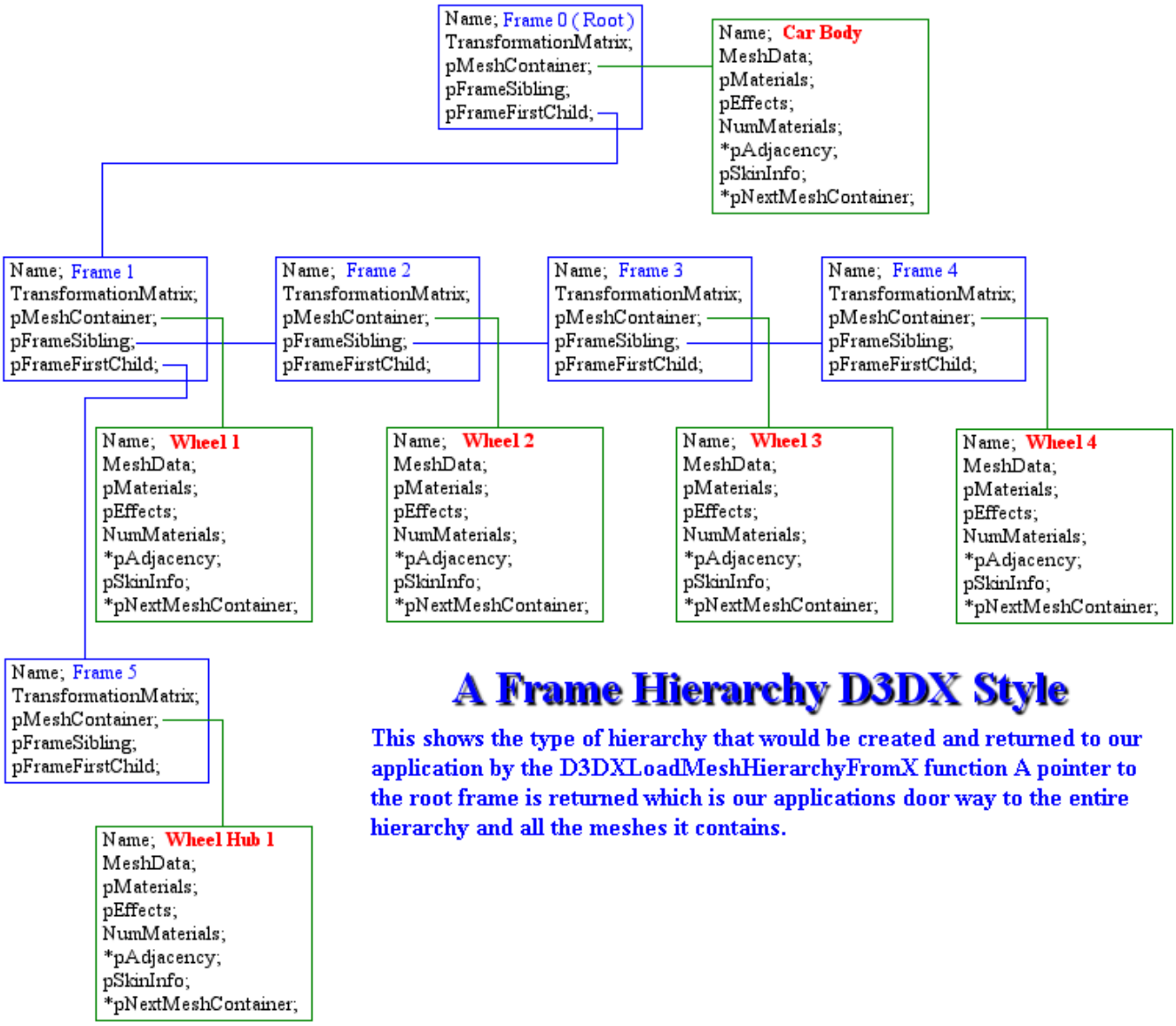


Figure 9.5

While we now know what the hierarchy will look like when the function returns, we still need to finish our examination of the parameters for D3DXLoadMeshHierarchyFromX. As it turns out, hierarchy construction is not completely automated. The application will need to play a role in the allocation of resource data (frames and mesh containers) as the hierarchy is assembled. This is important because we want the application to own the memory used by the hierarchy. If this were not the case, then routine hierarchy alterations or de-allocation of hierarchy components by the application would result in exceptions being generated.

LPD3DXALLOCATEHIERARCHY *pAlloc*

As the fourth parameter to the D3DXLoadMeshHierarchyFromX function, we must pass in a pointer to an application defined class that is derived from the ID3DXAllocateHierarchy interface. It is this parameter which is in many ways the key to the whole loading process. This interface cannot be instantiated on its own because its four member functions are pure virtual functions (they contain no function bodies). We must derive from it, and overload several functions in order to plug our application specific functionality into the D3DX hierarchy loading process.

> **Note:** Some versions of the DirectX 9 SDK suggest that this interface derives from IUnknown (much like the other COM interfaces we have seen in DirectX). This is not the case. It is more correct to think of this interface as being just a pure virtual C++ base class.

This interface is declared using all the standard interface macros to provide C and C++ development environment support (which we will discuss in a moment), so we need to be aware of these macros so that we can derive our interface using them too.

The next code listing is for ID3DXAllocateHierarchy as declared in d3dx9Anim.h.

```
DECLARE_INTERFACE( ID3DXAllocateHierarchy )
{
    // ID3DXAllocateHierarchy
    STDMETHOD(CreateFrame)         ( THIS_ LPCSTR Name,  LPD3DXFRAME *ppNewFrame) PURE;

    STDMETHOD(CreateMeshContainer)( THIS_ LPCSTR Name,
                                    LPD3DXMESHDATA pMeshData,
                                    LPD3DXMATERIAL pMaterials,
                                    LPD3DXEFFECTINSTANCE pEffectInstances,
                                    DWORD NumMaterials,
                                    DWORD *pAdjacency,
                                   LPD3DXSKININFO pSkinInfo,
                                   LPD3DXMESHCONTAINER *ppNewMeshContainer) PURE;

    STDMETHOD(DestroyFrame)        ( THIS_ LPD3DXFRAME pFrameToFree ) PURE;

    STDMETHOD(DestroyMeshContainer)( THIS_ LPD3DXMESHCONTAINER pMeshContToFree ) PURE;
};
```

> **Note:** If you ever have occasion to browse through any of the DirectX header files, you will see that virtually all interfaces are declared just like C++ classes. The difference is that they never have member variables, only member functions. This is exactly what an interface is (for those of you who don't know). The interface exposes only methods to the user of the API and keeps the data and private functionality nicely tucked away. Refer back to our COM discussion in Chapter Two for a refresher if needed.

The declaration may look a little strange, but do not be distracted by the macros, which will be built out during our compiler's pre-process. We will discuss those macros in the next section and it will all become clear. First we will look at how to derive our own class from this interface, which is something we *must* do. It is this derived object that we will pass to the D3DX loading function. D3DX will use the methods of our class to allocate the frames and mesh containers it created in application memory. This allows the application to chose how and where these resources should be allocated.

As the following code shows, deriving a class from an interface is quite similar to normal inheritance models.

```
class CAllocateHierarchy: public ID3DXAllocateHierarchy
{
public:

    // Constructor
    CAllocateHierarchy( int var ) : m_TestVar(var) {}

    STDMETHOD(CreateFrame)           ( THIS_ LPCTSTR Name,   LPD3DXFRAME *ppNewFrame);

    STDMETHOD(CreateMeshContainer)   ( THIS_ LPCTSTR Name,  LPD3DXMESHDATA pMeshData,
                                       LPD3DXMATERIAL pMaterials,
                                       LPD3DXEFFECTINSTANCE pEffectInstances,
                                       DWORD NumMaterials, DWORD *pAdjacency,
                                       LPD3DXSKININFO pSkinInfo,
                                       LPD3DXMESHCONTAINER *ppNewMeshContainer );

    STDMETHOD(DestroyFrame)          ( THIS_ LPD3DXFRAME pFrameToFree );

    STDMETHOD(DestroyMeshContainer)  ( THIS_ LPD3DXMESHCONTAINER pMeshContainerBase );

    // Public Variables for This Class
    int   m_TestVar
};
```

Our derived class adds a constructor and a public member variable (m_TestVar). The four functions of the ID3DXAllocateHierarchy base class have the 'PURE' macro at the end of each function prototype. This macro expands to '=0', which means the function is pure virtual and must be implemented in the derived class. That is all there is to deriving from an interface. In truth, this is not something that you will usually want or need to do, but it is required in this particular case. Of course, we do have to implement these four methods in our derived class to provide memory allocation and deallocation functions that D3DX can use during hierarchy construction. We will discuss the implementation of these four methods shortly. If you would like to better understand the macros seen above, read the next section. It is not vital that you understand these macros in order to use them, but we will discuss them anyway for those students who are interested. For those who are not, feel free to skip this next section on move straight on to discuss implementing the four methods of our ID3DXAllocateHierarchy derived class.

9.4.1 The COM Macros

When deriving a class from the ID3DXAllocateHierarchy interface, our derived class methods seem to be using quite a few strange looking macros: STDMETHOD and THIS_ for example. Technically speaking, we do not have to use them in order to derive a class from an interface, but it does make life easier when matching prototypes for base class functions. We want to ensure that they overload correctly in our derived class.

Let us start by looking at what these macros resolve to in the pre-processor at compile time. They can be found in the header file 'objbase.h' and are part of the standard COM development semantics. As you might suspect, DirectX follows the COM development guidelines and uses these macros throughout its interface declarations.

When we call the STDMETHOD macro, we pass in the name of the class method like so:

```
STDMETHOD( method )
```

When the pre-processor encounters this macro it resolves to:

```
virtual HRESULT STDMETHODCALLTYPE method
```

This expanded macro in turn contains the 'STDMETHODCALLTYPE' macro, which resolves to:

```
__stdcall
```

This declares the standard COM function calling convention. That is, the STDMETHOD fully resolves to the following, where 'method' is the name of the function passed into the macro.

```
virtual HRESULT _stdcall method
```

Thus, the CreateFrame function for our CAllocateHierarchy test class would be defined like so when processed by the compilers pre-processor:

```
virtual HRESULT _stdcall CreateFrame
```

So the macro is used to quickly define overridable interface functions that return an HRESULT. This is the standard COM style as we learned way back in Chapter Two.

The other macro we see used in the ID3DXAllocateHierarchy interface declaration (and every COM interface declaration) is the 'THIS_' macro. It is inserted before the first parameter of each method. This macro makes it possible to use the interface in a C environment that would otherwise have no concept of objects or methods. There are two versions of the 'THIS_' macro, wrapped by a compile time directive. If we remove some of the bits that are not relevant to this discussion, we end up with the following:

```
#if defined(__cplusplus) && !defined(CINTERFACE)
      #define     THIS_
#else
      #define     THIS_       INTERFACE FAR *This,
#endif
```

This is actually quite clever and is not something you will likely see very often, but it is worth discussing.

All of the macros (such as DECLARE_INTERFACE, STDMETHOD, etc.) allow for C style interface declaration and expansion, even if the application is being compiled in a C++ environment. Although all of these macros have been around forever, and are actually defined by the underlying COM object mechanism (objbase.h), DirectX follows them strictly, to allow a C environment to use the COM objects exposed.

As we can see, the expansion of the above 'THIS_' macro, in a C++ compiler (ignoring the && case for a moment), would result in absolutely nothing being injected into the portions of code by the pre-

processor. In other words, nothing would be inserted before the first parameter of each method. We know that with C++ classes, the 'this' pointer is implicitly inserted as the first parameter of all class member functions by the compiler. In a C environment however, this is not the case. This leads to the second case above, where a 'This' pointer is injected by the macro pre-processor during a compile as the first parameter to each method. But how does it know what type the 'This' parameter inserted into the method parameter list should be? We can see right at the start of the inserted code that there is another macro being used: `INTERFACE`. What we also see, is that at the start of every interface declaration in DirectX (or more specifically in this example, just prior to the declaration of the `ID3DXAllocateHierarchy` interface) the following lines of code which assigns a value to the INTERFACE definition:

```
#undef      INTERFACE
#define     INTERFACE   ID3DXAllocateHierarchy
```

If we look at the expansion of the `CreateFrame` member of the ID3DXAllocateHiearchy interface:

```
STDMETHOD (CreateFrame)( THIS_ LPCSTR Name,  LPD3DXFRAME *ppNewFrame)   PURE;
```

in a C environment, we get the following:

```
virtual HRESULT __stdcall CreateFrame(ID3DXAllocateHierarchy FAR *This,
                                      LPD3DXFRAME * ppNewFrame) = 0;
```

As you can see, as the first parameter to each method, a pointer to the actual structure that contains the member variables is inserted so the function can access them in C. This is exactly what the COM object exports, allowing us to use DirectX in a C environment. There is of course a little extra work to do, because we must remember to always explicitly pass the 'this' pointer when calling an interface method and we have to address some of the C++ specific concepts like 'virtual' and '=0'.

If you look at any of the DirectX header files that declare interfaces you will see, before the DECLARE_INTERFACE macro, the lines

```
#undef      INTERFACE
#define     INTERFACE   NameOfInterface
```

This means that when we use the 'THIS_' macro in a 'C' environment, a far pointer of the correct interface type is inserted at the head of each method's parameter list. You will see these lines of code before each interface declaration where 'NameOfInterface' would of course be replaced by the interface about to be declared. This allows for the behavior of the _THIS macro to be altered on a per interface basis.

Looking again at the compile time directives for the '`THIS_`' macro, we see the "`&& !defined(CINTERFACE)`" portion of the first if statement. This define is available for us to use from inside our C++ environment. We can '`#define CINTERFACE`' right at the start of the logical compile path, and develop using the C language thereafter even when using a C++ compiler. This is a nice additional level of portability.

Before moving on, there is one last issue to cover: how to access interfaces or classes in a C program when the language does not support these concepts. The key is the 'DECLARE_INTERFACE' macro.

As you know, C++ defines structs and classes similarly. One might consider a struct in C++ as the equivalent of a public class (classes are private by default). Since structs in C are data containers that cannot include member functions, the DECLARE_INTERFACE macro helps bridge the gap.

DirectX uses the 'DECLARE_INTERFACE' macro wherever it needs to declare an interface. If we imagine we declare a simple interface called ID3DXAllocateHierarchy with a single member function, it would be declared in the DirectX header files something like this:

```
DECLARE_INTERFACE( ID3DXAllocateHierarchy )
{
   STDMETHOD(CreateFrame) (THIS_ LPCSTR Name, LPD3DXFRAME * ppNewFrame) PURE;
};
```

If we look in objbase.h (in your C++ include directory) you will see that the interface declaration shown above is expanded differently depending on whether we are using a C or C++ environment.

```
#if defined(__cplusplus) && !defined(CINTERFACE)

      #define interface                                   struct

      #define DECLARE_INTERFACE(iface)                    \
        interface iface

      #define DECLARE_INTERFACE_(iface, baseiface)        \
              interface iface : public baseiface

            #define STDMETHOD(method)                           \
                    virtual HRESULT STDMETHODCALLTYPE method

            #define PURE                                        = 0

#else
      #define interface                                   struct

      #define DECLARE_INTERFACE(iface)                    \
        typedef interface iface {                  \
          struct iface##Vtbl FAR*lpVtbl;           \
        } iface;                                   \
        typedef struct iface##Vtbl iface##Vtbl; \
        struct iface##Vtbl

      #define DECLARE_INTERFACE_(iface, baseiface)        \
              DECLARE_INTERFACE(iface)

      #define STDMETHOD(method)                           \
              HRESULT (STDMETHODCALLTYPE * method)

      #define PURE

#endif
```

Notice that there are two separate DECLARE_INTERFACE macros, one with an underscore at the end, and one without. The one with the underscore is used wherever a base interface is required.

The above conditional code shows that in the C++ environment, the DECLARE_INTERFACE macro is simply replaced with the keyword 'struct' and, together with the macros used to declare the method (_THIS and STDMETHOD), would expand to a structure called ID3DXAllocateHierarchy which has a single virtual function as its only method. It is a regular structure that has a virtual function which uses the standard COM calling convention and returns an HRESULT as shown below:

```
struct ID3DXAllocateHierarchy
{
   virtual HRESULT __stdcall CreateFrame(LPCSTR Name,LPD3DXFRAME *ppNewFrame) = 0;
};
```

This is perfectly legal C++ and by studying the expansion code above you can see how this structure definition was arrived at.

If a C environment is being used, the DECLARE_INTERFACE macro is expanded to something a bit more cryptic: This is because structures can not have methods in C, so a workaround is put in place. The C section of the expansion would create the following:

```
typedef struct ID3DXAllocateHierarchy
{
  struct ID3DXAllocateHierarchyVtbl FAR * lpVtbl;
} ID3DXAllocateHierarchy;

struct ID3DXAllocateHierarchyVtbl
{
  HRESULT (__stdcall * CreateFrame ) (ID3DXAllocateHierarchy FAR * This,
                                      LPCSTR Name, LPD3DXFRAME * ppNewFrame);
};
```

The C case creates two structures instead of one. The first is the actual ID3DXAllocateHierarchy structure and it contains a single member variable. This is a pointer to the second structure type created by the macro (called ID3DXAllocateHierarchyVtbl). This structure is referred to as the VTABLE and as you can see, it contains one or more pointers to functions as its members. The names of the structures are the same with the exception of the Vtbl appended to the end of the second structure. The Vtable is essentially where functions that were part of the class in the C++ case will be stored. If we were to carry on declaring methods in our interface, this second structure would end up containing callbacks for all of the interface methods. Therefore, we can think of the Vtable structure as being linked to the first structure as a container of function pointers.

In C++ the methods of this interface can be called like so:

```
pAllocHierarchy->CreateFrame (…) ;        // Calling interface methods in C++
```

In C, the function pointers are stored in the Vtable structure which is accessible via the first structure's lpVtbl member. Therefore, we must take this into account when calling member functions of the interface in the C development environment.

```
pAllocHierarchy->lpVtbl->CreateFrame (…); // Calling interface methods in C
```

Inheritance is not implicitly or automatically supported in a C environment. We see this in the **DECLARE_INTERFACE** macro. The two sections (C/C++) of the DECLARE_INTERFACE_ macro (note the underscore) are used to declare an interface that has a base class. The two macros are shown again next:

The C++ case:

```
#define DECLARE_INTERFACE_(iface, baseiface )interface iface : public baseiface
```

This resolves the standard class/struct inheritance code for a declaration:

```
struct iface : public baseiface
```

The C case ignores the base interface:

```
#define DECLARE_INTERFACE_(iface, baseiface)  DECLARE_INTERFACE(iface)
```

This resolves to:

```
struct iface
```

This is unfortunate, but is something that we certainly can live without. All DirectX interfaces re-declare their base class functions anyway, so they are explicitly supported in derived interfaces. We still get access to all of the base class functionality via the VTable callbacks, even if we cannot cast between types in ways that we might take for granted in C++. If you have ever wondered why all of the interfaces in the DirectX header files re-declare all of their base class methods (which should automatically be inherited from the base interface), now you know -- it is for C environment access compatibility where method inheritance is not supported.

Bear in mind that the compiler will automatically assume that it is to compile code for a C environment when the '.c' file extension is used. So be aware of this if you use DirectX in a project which mixes '.c' files, with '.cpp' files.

9.4.2 The ID3DXAllocateHierarchy Interface

The fourth parameter to the D3DXLoadMeshHierachyFromX function is a pointer to an application defined class derived from the ID3DXAllocateHierarchy interface. The derived class must implement the four functions of the base interface so that D3DXLoadMeshHierarchyFromX can call them when mesh containers and frames need to be allocated and de-allocated. This process of calling back out to the application allows for application resource ownership during hierarchy construction. This is important for applications that want to add additional functionality to the interfaces that are provided.

Let us look at an example of an interface derived class:

```
class CAllocateHierarchy: public ID3DXAllocateHierarchy
{
public:

    STDMETHOD(CreateFrame)          ( THIS_ LPCTSTR Name,  LPD3DXFRAME *ppNewFrame);

    STDMETHOD(CreateMeshContainer)  ( THIS_ LPCTSTR Name,  LPD3DXMESHDATA pMeshData,
                                      LPD3DXMATERIAL pMaterials,
                                      LPD3DXEFFECTINSTANCE pEffectInstances,
                                      DWORD NumMaterials, DWORD *pAdjacency,
                                      LPD3DXSKININFO pSkinInfo,
                                      LPD3DXMESHCONTAINER *ppNewMeshContainer );

   STDMETHOD(DestroyFrame)          ( THIS_ LPD3DXFRAME pFrameToFree);

   STDMETHOD(DestroyMeshContainer)  (THIS_ LPD3DXMESHCONTAINER pMeshContainerBase);
};
```

Because this is a class and not merely an interface (a pure abstract base class) we can now add any number of additional methods and member variables to enhance the functionality of the original ID3DXAllocateHierarchy. The above example does not add any new member variables or methods, but it must implement the four base class methods. This allows D3DXLoadMeshHierarchyFromX to use them like callbacks.

9.4.2.1 Allocating Frames

HRESULT CreateFrame(LPCSTR *Name*, LPD3DXFRAME **ppNewFrame*)

This function is called when a new frame is encountered in the X file. The application is responsible for allocating a new D3DXFRAME and then returning it to D3DXLoadMeshHierrachyFromX by means of the second parameter. D3DX will populate the frame we return with data from the X file and attach it to the hierarchy in the correct position. At its simplest, this function just has to allocate a new D3DXFRAME structure and point the second parameter passed in to point at it.

LPCSTR Name
This first parameter is the name of the frame being created and will be passed into the function by D3DXLoadMesHierarchyFromX when it encounters a frame in the X file that is about to be loaded. Recall that there is storage for a string pointer in the base frame structure. In the case of a character model for example, the artist or 3D modeller might name the frames with descriptive labels such as “Left Arm”, “Right Leg” or “Torso” within the X file.

When we allocate the D3DXFRAME structure inside this function, we will want to store this name in the D3DXFRAME::Name member variable so that we can access the frame by name later. However it is important to realize that the memory of the string we are passed is not owned by the application, it is owned by D3DX. As such, we cannot simply store the string pointer we are passed in our D3DXFRAME structure because we cannot make assumptions about what D3DX will do with the memory once we return from the function. For example, if D3DX were to free it, our frame structure

would contain a dangling pointer to a string that no longer exists. Therefore this string must be duplicated in any way you see fit (i.e. strdup or _tcsdup if you want to be Unicode compliant) and stored in the D3DXFRAME structure you create. Remember that if you duplicate the name, you must make sure that when the frame is destroyed you also free the string memory that you created during the frame creation process. We will examine how frames are destroyed in a moment.

LPD3DXFRAME *ppNewFrame

The second parameter to this function is the address of a D3DXFRAME pointer (i.e. a pointer to a pointer). This time the application is being passed a pointer by D3DX, not the other way round as is usually the case with D3DX function calls. The de-referenced pointer should contain `NULL` at this point, although it may not. We use this pointer to return the pointer to the frame that we create, back to the D3DXLoadMeshHierarchyFromX function.

The following code snippet is a simple version of a CreateFrame function implementation. It does everything D3DX requires and a bit more. It allocates a new D3DXFRAME, copies the name passed into the function, sets the frame transformation matrix to an identity matrix, and assigns the passed pointer to the newly created frame. Setting the transformation matrix to identity is purely a safety precaution as it will most likely be overwritten with the matrix data extracted from the X file when the frame data is loaded. However, this is a nice default state.

```
HRESULT CAllocateHierarchy::CreateFrame( LPCTSTR Name, LPD3DXFRAME *ppNewFrame )
{
    D3DXFRAME  *pNewFrame = NULL;

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

    // Allocate a new frame
    pNewFrame = new D3DXFRAME;
    if ( !pNewFrame ) return E_OUTOFMEMORY;

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

    // Store the passed name in the frame structure
    // and set the new frame transformation matrix
    // to an identity matrix by default
    if ( Name ) pNewFrame->Name = _tcsdup( Name );
    D3DXMatrixIdentity( &pNewFrame->TransformationMatrix );

    // Assign the D3DX passed pointer to our new application allocated frame
    *ppNewFrame = pNewFrame;

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

When we examine how to derive from these structures a little later on, you will see that these Create functions can and do serve a greater purpose, allowing us to store other data in our frame objects. When the above function returns, the passed pointer will point to our application allocated frame structure and it will contain the frame's name and an identity matrix. D3DX will then load the frame data from the X

file for this frame and store it in the structure before inserting the frame into its correct place in the hierarchy. This function will be called once for every frame in the hierarchy.

9.4.2.2 Deallocating Frames

If D3DX does not own the frame hierarchy memory, then how can it deallocate that memory when the hierarchy needs to be destroyed? In fact, since our application will only ultimately get back a root frame pointer from the D3DXLoadMeshHierarchyFromX function, how can we ourselves deallocate the hierarchy?

It just so happens that the D3DX library exposes a global function called D3DXFrameDestroy that will help with this process. When we have finished with our frame hierarchy we will call this function to cleanup the memory. We will discuss this function in detail at the end of this section. For now you should know that the D3DXFrameDestroy call takes two parameters. This first parameter is a pointer to a frame. This input frame and all of its descendants in the hierarchy will be destroyed. This would include the deallocation of any meshes as well. To release the whole hierarchy, we would pass in the root frame; to release only a sub-tree in the hierarchy, we would pass the appropriate node.

Because the memory is application owned, the application is responsible for actually physically releasing that memory. The second parameter to the D3DXFrameDestroy function is a pointer to an ID3DXAllocateHierarchy derived object (just as we saw for allocation). The function will step through each frame in the hierarchy and call ID3DXAllocateHierarchy::DestroyFrame. This gives control back to the application defined class for memory release.

```
HRESULT DestroyFrame( LPD3DXFRAME pFrameToFree );
```

The single function parameter is a pointer to the frame that D3DX would like released. This function will be called once for every frame in the hierarchy when the hierarchy is destroyed. The destruction of the full hierarchy is set in motion by the application calling the global D3DXFrameDestroy function with a pointer to the root frame.

Next we see a quick example implementation that matches up with the CreateFrame example we wrote earlier. Notice how the function frees the passed frame and the frame name as well. The name was allocated in the CreateFrame function when _tcsdup was called to make a copy of the passed name. Just as the CAllocateHierarchy::CreateFrame function is called by D3DX for each frame that needs to be created during X file loading, the CAllocateHierarchy::DestroyFrame function is called when the memory for each frame in the hierarchy needs to be released.

```
HRESULT CAllocateHierarchy::DestroyFrame( LPD3DXFRAME pFrameToFree )
{
     // Validate Parameters
    if ( !pFrameToFree ) return D3D_OK;

    // Release memory for name and frame
    if ( pFrameToFree->Name ) free( pFrameToFree->Name ); // '_tcsdup' allocated.
    delete pFrameToFree;                                  // 'new' allocated.
```

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

Note that we used ‘free’ and not ‘delete’ to free the string memory because the _tcsdup function used in the CreateFrame method uses a C memory allocation function (alloc or malloc, etc.). The frame itself was allocated using ‘new’, so we free it with ‘delete’.

9.4.2.3 Allocating Mesh Containers

To handle allocation of meshes and related resources stored in the X file, our application-derived class must override the ID3DXAllocateHierarchy::CreateMeshContainer function. This function is similar to CreateFrame. It is called by D3DX so that the application can allocate and initialize a D3DXMESHCONTAINER structure and return it to the file loading function. Recall that D3DX can automate the loading of the mesh geometry, but it does not manage resources such as materials, textures, or effects (the application is responsible for that task). We saw this in the last chapter when we loaded the textures for the passed texture filenames and stored them in our CScene class.

The initialization and validation code is a bit laborious -- we have to copy, validate, and potentially manipulate all of the mesh related data passed in before we return the mesh container to D3DX. The concept is very similar to what we did in the last chapter when loading meshes using D3DXLoadMeshFromX. Recall that we are passed a loaded mesh and buffers filled with mesh related assets such as materials and texture filenames and we must process those resources ourselves. This function will be called once for each mesh in the X File hierarchy.

```
HRESULT CreateMeshContainer
(
    LPCSTR            Name,
    LPD3DXMESHDATA    pMeshData,
    LPD3DXMATERIAL    pMaterials,
    LPD3DXEFFECTINSTANCE pEffectInstances,
    DWORD             NumMaterials,
    DWORD *           pAdjacency,
    LPD3DXSKININFO    pSkinInfo,
    LPD3DXMESHCONTAINER *ppNewMeshContainer
);
```

LPCTSTR Name
This string stores the name of the mesh as defined inside the X file. We will often want to store this name along with our mesh so we can identify it later. Like the name in the frame structure, we must free the name when the mesh container is destroyed.

LPD3DXMESHDATA pMeshData
D3DX passes us the relevant mesh object using this structure pointer. It will either contain an ID3DXMesh, an ID3DXPMesh, or an ID3DXPatchMesh. We can do whatever we like with the mesh - clone it, manipulate it and then store it in the mesh container that we allocate, or we can simply store the passed mesh in the D3DXMESHCONTAINER structure immediately. It is important to understand that

eventually D3DX will Release() the interface to the mesh passed into this function. Therefore, if you are going to store away the mesh interface and you would like it to be persistent, you must call AddRef().

LPD3DXMATERIAL pMaterials
LPD3DXEFFECTINSTANCE pEffectInstances
DWORD NumMaterials
DWORD * pAdjacency
LPD3DXSKININFO pSkinInfo

These parameters are essentially carbon copies of the information you are required to store in the D3DXMESHCONTAINER structure. They are very similar to the information we get back when we call D3DXLoadMeshFromX. Again, we do not own the memory for these input parameters, and they will be destroyed by the caller (D3DX), so we must duplicate them if we wish to store the resources in the mesh container.

LPD3DXMESHCONTAINER * ppNewMeshContainer

As with the CreateFrame function, this pointer allows us to return our newly allocated mesh container back to D3DX after it has been populated with data passed into the function that we want to retain. For now, let us just look at a quick example that deals with the basics of mesh allocation. We will look at a more complete implementation that handles materials and textures in the workbook accompanying this chapter.

```
HRESULT CAllocateHierarchy::CreateMeshContainer( LPCTSTR Name, LPD3DXMESHDATA pMeshData,
                                                 LPD3DXMATERIAL pMaterials,
                                                 LPD3DXEFFECTINSTANCE pEffectInstances,
                                                 DWORD NumMaterials, DWORD *pAdjacency,
                                                LPD3DXSKININFO pSkinInfo,
                                                 LPD3DXMESHCONTAINER *ppNewMeshContainer )
{
    HRESULT            hRet;
    LPD3DXMESH         pMesh = NULL;
    D3DXMESHCONTAINER *pMeshContainer = NULL;

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

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

    // Allocate a new 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. We can't simply copy the pointer here because the memory
    // for the string belongs to the caller (D3DX)
    if ( Name ) pMeshContainer->Name = _tcsdup( Name );

    // Copy over passed mesh data structure into our new mesh contain 'MeshData' member
    // Remember, we are also copying the mesh interface stored in the 'MeshData' structure
    // so we should increase the ref count
    pMeshContainer->MeshData = *pMeshData;
    pMeshContainer->MeshData.pMesh->AddRef();
```

```
    // Attempt to optimize the new mesh
    pMeshContainer->MeshData.pMesh->OptimizeInPlace(D3DXMESHOPT_VERTEXCACHE ,
                                                    pAdjacency , NULL , NULL , NULL);

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

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

Studying the previous example function shows us that the process is not very complicated. It basically just creates a new mesh container and copies the passed mesh into it. The input pointer is assigned so that D3DX can insert the mesh into the hierarchy. Processing materials and textures will make the function longer, but no more difficult. The same is true if we need to change the mesh format to accommodate data required by the application, such as vertex normals if they are not included in the X file. In this case we would simply clone the passed mesh and store the newly cloned mesh in the mesh container. When this function returns the newly created mesh container to D3DX, it will be attached to the correct parent frame in the hierarchy. This function will be called for every mesh contained in the X file.

9.4.2.4 Deallocating Mesh Containers

During hierarchy destruction, ID3DXAllocateHierarchy::DestroyMeshContainer is called by D3DXFrameDestroy for each mesh container in the hierarchy. This allows the application to free the mesh containers and any application-defined content in an appropriate way. This parallels the requirement to implement a DestroyFrame function for frame cleanup.

HRESULT DestroyMeshContainer(LPD3DXMESHCONTAINER *pMeshContToFree*);

DestroyMeshContainer takes a single parameter: a pointer to the mesh container that D3DX would like us to free from memory. Here is an example that works in conjunction with the CreateMeshContainer function just discussed:

```
HRESULT CAllocateHierarchy::DestroyMeshContainer(LPD3DXMESHCONTAINER pContToFree)
{
    // Validate Parameters
    if ( !pContToFree ) return D3D_OK;

    // Release data
    if ( pContToFree->Name ) free( pContToFree->Name );
    if ( pContToFree->MeshData.pMesh)
         pContToFree->MeshData.pMesh->Release();

    delete pContToFree;

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

9.4.2.5 The D3DXFrameDestroy Function

Most of the resource deallocation was handled in the DestroyFrame and DestroyMeshContainer functions we just examined. Initiating the destruction of the hierarchy, or a subtree thereof, is done with the following global D3DX function:

```
HRESULT D3DXFrameDestroy
(
    LPD3DXFRAME pFrameRoot,
    LPD3DXALLOCATEHIERARCHY pAlloc
);
```

LPD3DXFRAME pFrameRoot
This frame is the starting point for the destruction process. D3DX will start deallocation here at this node and then work its way through all subsequent child frames and mesh containers, cleaning up the resources as it goes. To destroy the entire hierarchy, just pass the root frame that was returned via the call to `D3DXLoadMeshHierarchyFromX`. Be sure to set your root frame variable to `NULL` after calling this function, to ensure that you do not try to access invalid memory later on.

LPD3DXALLOCATEHIERARCHY *pAlloc*
The second parameter is an instance of our derived `ID3DXAllocateHierarchy` class. It does not have to be the same physical object that was passed into the D3DXLoadMeshHierarchyFromX function, and it does not have to be allocated in any special way. For example, this object could be temporarily instantiated on the stack and it will work just as well. This will be the object D3DX calls to access the two Destroy functions (DestroyFrame and DestroyMeshContainer). Be sure to pass in the same type of derived class that was used to create the hierarchy in the first place.

Here is an example of a call that creates a local allocator and then cleans up the entire hierarchy:

```
CAllocateHierarchy Allocator;

if (m_pFrameRoot) D3DXFrameDestroy(m_pFrameRoot, &Allocator);
m_pFrameRoot = NULL;
```

9.4.3 Extending the D3DX Hierarchy

At first it may seem that hierarchy loading is more complicated than is called for. After all, D3DX should certainly be perfectly capable of doing everything itself. But by managing much of the process at the application level, we gain a significant advantage: the ability to extend the data structures stored in the hierarchy.

D3DX will concern itself only with the core navigation components of the hierarchy (`D3DXFRAME::pFrameSibling`, `D3DXFRAME::pFrameFirstChild`, `D3DXFRAME::pMeshContainer` and `D3DXMESHCONTAINER::pNextMeshContainer`), the frame transformation matrix (when working with

animation) and the frame name (when working with skinned meshes). The rest of the data structure can be used at our discretion.

As a result, we can derive our own structures from both `D3DXFRAME` and `D3DXMESHCONTAINER` and store whatever additional information we require. Since we are responsible for both the creation and destruction of these structures, as long as we cast it correctly in all of the right places, D3DX will work correctly without requiring knowledge of our additional structure members.

There are a number of cases where this proves to be useful. Some examples would include:

1) Caching the world matrix. Because the frame matrices are relative to their respective parent's space, they can be quite difficult (and potentially expensive) to work with when it comes to rendering the hierarchy. This is especially true in cases where multiple frames exist at the same level (as a sibling linked list).

   To expedite the process, each derived frame can store a new matrix that is a combination of all parent frame transformation matrices and its own relative matrix. We can build these matrices in a single hierarchy update pass. Then, when we render each mesh in the hierarchy, we can just use this matrix (assigned to its owner frame) as the mesh world matrix and avoid having to combine relative matrices for each mesh that we render.

   So we might derive a structure from D3DXFRAME similar to the following:

```
struct D3DXFRAME_DERIVED : public D3DXFRAME
{
      D3DXMATRIX  mtxCombined;
};
```

2) Hierarchical intersection testing. We can add bounding volume information to our frame structures so that hierarchical tests can be performed. For example, we might choose to store an axis aligned bounding box for collision testing, frustum culling, etc.:

```
struct D3DXFRAME_DERIVED : public D3DXFRAME
{
      D3DXMATRIX  mtxCombined;
      D3DXVECTOR3 aabb_min, aabb_max;
};
```

   The bounding volume at each frame in the hierarchy would be a combination of the frame's immediate bounding volume (built using all of the bounding volumes for meshes stored in the frame) plus the bounding volumes for its list of direct children (which themselves are calculated in the same way). These bounding volumes would all be world space volumes, and the bounding volume hierarchy would be built from the bottom up. If a node animates or changes in some way that would affect position, orientation, or scale, we can just recalculate its bounding volume and propagate that change up the tree to the root, starting from the node that changed. The significant advantage here is that the parent bounding volume totally encloses its children. Thus, if the parent is not visible (via an AABB/Frustum test for example) then none of its children could possibly be visible either. This minimizes rendering calls and is a common optimization in 3D

engines. The same holds true for other collision tests as well. If something cannot collide with the parent volume, it cannot possibly collide with any of its children either. This is one of the most beneficial aspects of a spatial hierarchy and we will encounter it again and again in our 3D graphics programming studies (including later in this very course).

3) Application specific mesh management. Rather than use the provided D3DXMESHDATA structure, we can derive our own mesh container to suit our needs:

```
struct D3DXMESHCONTAINER_DERIVED : public D3DXMESHCONTAINER
{
      CTriMesh    *pMesh;
};
```

Here we can store the mesh as we require and work directly within our engine's pre-established framework. This allows us to plug our mesh class directly into the D3DX hierarchy and store it in a mesh container. The CTriMesh listed above would be an example of a proprietary application mesh class that you might write to manage your triangle based mesh data.

One thing to make sure that you do straight away is update the destroy functions in your hierarchy allocation class. If you are working with a derived structure, you must make sure that you cast the pointer that was passed to the destroy functions to the derived structure type before you `delete` it. If you do not, you risk important destructors not being called, memory not being freed, or any manner of potential general protection faults.

There are some important exceptions to the above cases, where overriding the default behaviour may cause problems. For example, not storing the mesh in the provided D3DXMESHDATA structure can lead to one issue. Consider the function `D3DXFrameCalculateBoundingSphere`. This is a very nice function that iterates through the frame hierarchy starting from a passed frame and tests the mesh data stored there to return a bounding sphere that encapsulates all of the meshes that are children (both immediate and non-immediate). This function will expect the mesh data to be stored in the standard place (inside the D3DXMESHDATA structure of the mesh container) and will not be able to calculate a bounding sphere unless this is the case. Another example is `D3DXSaveMeshHierarchyToFile`. This function iterates through the hierarchy and writes mesh, material, effect, and texture data out to an X file. Note that in both of these cases there is a simple solution -- we can store a reference to the mesh in both the provided data structure (D3DXMESHDATA) and our custom mesh class (CTriMesh in our example above).

9.4.4 D3DXLoadMeshHierarchyFromX Continued

There are still two parameters left to discuss before we use D3DXLoadMeshHierarchyFromX.

```
HRESULT D3DXLoadMeshHierarchyFromX
(
    LPCTSTR                 Filename,
    DWORD                   MeshOptions,
    LPDIRECT3DDEVICE9       pDevice,
```

```
    LPD3DXALLOCATEHIERARCHY       pAlloc,
    LPD3DXLOADUSERDATA            pUserDataLoader,
    LPD3DXFRAME*                  ppFrameHeirarchy,
    LPD3DXANIMATIONCONTROLLER*    ppAnimController
);
```

We will address the animation controller type in the next chapter. For now, we will just assume that the X file contains no animation and we will set this parameter to NULL. This tells the function that we are not interested in getting back animation data, even if it exists in the file.

LPD3DXLOADUSERDATA pUserDataLoader
The fifth parameter is a pointer to an ID3DXLoadUserData interface. Much like ID3DXAllocateHierarchy, this parameter is also going to be passed as an application-defined class that is derived from the ID3DXLoadUserData interface. The methods of this object will be called (by the D3DX loading function) whenever data objects built from custom templates are encountered in the X file.

If you are not using custom data objects, then D3DX will not require this pointer and you can simply pass NULL. In fact, even if the X file does contain custom data objects you can still pass in NULL. In this case, D3DX will simply ignore custom information and just load the standard data objects.

Our ID3DXLoadUserData derived object must implement three functions: LoadTopLevelData, LoadFrameChildData, and LoadMeshChildData. In order to properly derive from this class and process custom data objects, we must be familiar with all the X file topics we discussed earlier. Specifically, we need to be comfortable with the IDirectXFileData interface. While we will not need to use this interface, D3DX will pass our callback functions an ID3DXFileData interface for each custom data object it encounters. This interface is almost identical to the IDirectXFileData interface we discussed earlier. It is a newer interface however that has been wrapped up in the D3DX library to make accessing the data a little simpler. If you understand the IDirectXFileData methods, you will understand how to use all of the analogous methods exposed by the ID3DXFileData interface.

9.4.4.1 The ID3DXLoadUserData Interface

The ID3DXLoadUserData interface is declared in d3dx9anim.h. This is another pure abstract non-COM interface just like ID3DXAllocateHierarchy. However, if we look at the following interface declaration we will notice something different about this declaration when compared to all others.

```
DECLARE_INTERFACE(ID3DXLoadUserData)
{
    STDMETHOD(LoadTopLevelData)    ( LPD3DXFILEDATA pXofChildData) PURE;

    STDMETHOD(LoadFrameChildData) ( LPD3DXFRAME pFrame, LPD3DXFILEDATA pXofChildData) PURE;

    STDMETHOD(LoadMeshChildData)   ( LPD3DXMESHCONTAINER pMeshContainer,
                                     LPD3DXFILEDATA pXofChildData) PURE;
};
```

Compare this to the following interface that more closely follows the rules we discussed earlier about declaring interfaces: (It is probably how ID3DXLoadUserData should have been defined.)

```
#undef INTERFACE
#define      INTERFACE ID3DXLoadUserData

DECLARE_INTERFACE(ID3DXLoadUserData)
{
    STDMETHOD(LoadTopLevelData)   ( THIS_ LPD3DXFILEDATA pXofChildData) PURE;

    STDMETHOD(LoadFrameChildData) ( THIS_ LPD3DXFRAME pFrame,
                                    LPD3DXFILEDATA pXofChildData) PURE;

    STDMETHOD(LoadMeshChildData)  ( THIS_ LPD3DXMESHCONTAINER pMeshContainer,
                                    LPD3DXFILEDATA pXofChildData) PURE;
};
```

So there are two key differences (perhaps by design, perhaps an oversight) -- the interface is not defined before the declaration and the methods have not had the 'THIS_' macro inserted at the head of their parameter lists. As a result, this interface cannot be used in a C environment.

In fact, there are two interfaces in this header file (d3dx9anim.h) where the same declaration style is presented. The other is the declaration of the ID3DXSaveUserData interface. If you need to use either interface in a C environment, your only option is to alter the interface declaration in d3dx9anim.h to the format seen above.

Deriving a class from the ID3DXLoadUserData interface in a C++ environment is straightforward:

```
class CLoadUserData : public ID3DXLoadUserData
{
public:
    STDMETHOD(LoadTopLevelData)   (LPD3DXFILEDATA pXofChildData);
    STDMETHOD(LoadMeshChildData)  (LPD3DXMESHCONTAINER pMeshContainer,
                                   LPD3DXFILEDATA pXofChildData);
    STDMETHOD(LoadFrameChildData) (LPD3DXFRAME pFrame, LPD3DXFILEDATA pXofChildData);
};
```

If you are using a C environment and you have edited and corrected the ID3DXLoadUserData declaration in the d3dx9anim.h header file as mentioned above, then your derived class declaration would look something like this:

```
class CLoadUserData : public ID3DXLoadUserData
{
public:
    STDMETHOD(LoadTopLevelData)  (THIS_  LPD3DXFILEDATA pXofChildData);
    STDMETHOD(LoadMeshChildData) (THIS_  LPD3DXMESHCONTAINER pMeshContainer,
                                  THIS_  LPD3DXFILEDATA pXofChildData);
    STDMETHOD(LoadFrameChildData)(THIS_  LPD3DXFRAME pFrame, LPD3DXFILEDATA pXofChildData);
};
```

9.4.4.2 Loading Custom Top Level Data

HRESULT LoadTopLevelData(LPD3DXFILEDATA *pXofChildData*);

Recall that a top level data object is not defined inside any other data object (i.e., it does not have a parent) in the X file. When the D3DXLoadMeshHierarchyFromX function encounters a top level data object that is not one of the data objects based on a standard template, it calls the LoadTopLevelData function of your passed ID3DXLoadUserData derived object so that the application can process the data as it deems appropriate.

The function receives a pointer to an ID3DXFileData interface which provides all of the methods we need to identify the custom object (GetName, GetType, and GetID) and also allows us to retrieve a pointer to the actual data of the object for extraction (via its Lock and Unlock methods). We discussed the IDirectXFileData interface in detail earlier in this chapter and the ID3DXFileData interface is very similar. Its methods are shown below:

```
GetChildren ( SIZE_T *puiChildren );
GetChild    ( SIZE_T uiChild, ID3DXFileData **ppChild );
GetEnum     ( ID3DXFileEnumObject **ppObj );
GetId       ( LPGUID pId );
GetName     ( LPSTR szName, SIZE_T *puiSize );
GetType     ( const GUID *pType );
IsReference ( VOID );
Lock        ( SIZE_T *pSize, const VOID **ppData );
Unlock      ( VOID );
```

To see how we might implement such a function to handle the processing of top level custom data objects, let us go back to our earlier example using a custom template.

```
template TestTemplate
{
  <A96F6E5D-04C4-49C8-8113-B5485E04A866>
  FLOAT TestFloat1;
  FLOAT TestFloat2;
  FLOAT TestFloat3;
}
```

Let us assume that the X file we are loading has at least one of these objects as a top level object. In this example we will include an object called MyFirstValues:

```
TestTemplate MyFirstValues
{
  1.000000;
  2.000000;
  3.000000;
}
```

In order to store and manage this type of data in our application, we would probably define a structure in our code that mirrors the template definition in the X file:

```
struct TestTemplate
{
    float TestFloat1;
    float TestFloat2;
    float TestFloat3;
};
```

Somewhere else in our code, before we load the X file, we would want to define a GUID that mirrors the values stored in the template. This will allow us to compare GUIDs when we need to determine custom object types when the function is called.

```
// The GUID used in the file for the template, so we can identify it
const GUID IID_TestTemplate = {0xA96F6E5D, 0x4C4, 0x49C8, {0x81, 0x13, 0xB5, 0x48,
0x5E, 0x4, 0xA8, 0x66}};
```

You could also create this GUID using the DEFINE_GUID macro:

```
DEFINE_GUID ( IID_TestTemplate , 0xA96F6E5D, 0x4C4, 0x49C8, 0x81, 0x13, 0xB5, 0x48,
0x5E, 0x4, 0xA8, 0x66 };
```

Now let us look at our LoadTopLevelData function:

```
HRESULT CLoadUserData::LoadTopLevelData(LPD3DXFILEDATA pXofChildData)
{
    // First retrieve the name of the data object
    TCHAR DataName[256];
    ULONG StringSize = 256;
    pXofChildData->GetName( DataName, &StringSize );

    // Next retrieve the GUID of the data object if one exists
    GUID DataUID;
    pXofChildData->GetId( &DataUID );

    // Next up, retrieve the GUID of the template which defined it.
    const GUID * pTypeUID;
    pXofChildData->GetType(&pTypeUID );

    // Ensure that we are importing the type that we expect
    if ( memcmp( pTypeUID, &TestTemplateUID, sizeof(GUID) ) != 0 )
        return D3D_OK;

    // OK, we know it's our type, import and process the data
    TestTemplate * pData;
    ULONG DataSize;
    pXofChildData->Lock( &DataSize, ( const void**)&pData );

    // We can now use the data
    DO SOMETHING WITH DATA HERE (ex. COPY AND STORE IT SOMEWHERE);

    PXofChildData->Unlock();

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

This is the core of our custom data processor. We retrieve the type via a call to `GetType`, the data itself via a call to `GetData`, and then depending on the value of the type `GUID` returned, we cast and process the data pointed to by the returned pointer. If we were using multiple custom templates, a switch statement or some other conditional would probably be in order, but the concept is the same.

Note: Remember that IDirectXFileData::GetID returns the GUID of the object, not the template.

Do not forget that if you are processing data objects that have child data objects (or child data reference objects) then you will want to call IDirectXFileData::GetNextObject to retrieve and process those as well. As the child objects may themselves have child objects, suffice to say, this could be a deeply recursive process which requires some thought on how best to implement this code. Using closed or restricted templates can help minimize code complexity and recursion.

It should be noted that the code example above was written a little bit dangerously in an effort to improve clarity of the process. It assumes that any padding added to the structure TestTemplate by the compiler would also be mirrored in the X file data object, which is not usually the case. Therefore, because file data is not guaranteed to be aligned properly with byte boundaries in your C++ structure, you should access *data* with unaligned pointers. For example, you could step through the data with a BYTE pointer and extract each member by correctly offsetting and casting the de-referenced pointer into another variable type.

Refer to the CD3Dfile.cpp file that ships with the DirectX SDK (Samples\C++\Common\Src) to see how IDirectXFileData can be used to recursively process a hierarchy.

9.4.4.3 Loading Custom Child Data

When a top level object based on one of the open standard templates includes a custom child, D3DX must call out to the application to deal with this data. There are two important functions we must be prepared to implement in our derived ID3DXLoadUserData class. The first function will handle custom children found when processing a frame and the second handles custom children found when processing a mesh.

```
HRESULT LoadFrameChildData(LPD3DXFRAME pFrame,
                           LPD3DXFILEDATA pXofChildData);
```

LoadFrameChildData will be called by D3DXLoadMeshHierarchyFromX whenever an unknown object is encountered by the loading code that is an immediate child of a frame being processed. In addition to the expected ID3DXFileData type that allows us to extract the custom data, the function also takes a pointer to the frame in the hierarchy for which the custom data object is a child. This is useful if our application is using a derived D3DXFRAME structure with additional members that might be needed to store this custom data in some meaningful way. For example, the custom object might contain a string that describes what the frame is used for and whether or not it should be animated. If we were using a derived D3DXFRAME class, we could copy this string into a member variable in the frame structure. Thus each frame would contain a description that could be used for printing out debug information.

The following snippet shows our custom template embedded inside a standard frame object as an immediate child:

```
Xof 0303txt 0032

Frame Root
{
      FrameTransformMatrix
      {
        1.000000, 0.000000, 0.000000,  0.000000,
        0.000000, 1.000000, 0.000000,  0.000000,
        0.000000, 0.000000, 1.000000,  0.000000,
       50.000000, 0.000000, 50.000000, 1.000000;;
      }

      Mesh CarBody { mesh data would go in here }

      TestTemplate MyFirstValues
      {
            1.000000;
            2.000000;
            3.000000;
      }
}
```

When D3DXLoadMeshHierarchyFromX encounters a custom data object as an immediate child of a mesh object, the LoadMeshChildData method is called to facilitate processing. Like its predecessor, the function provides access to the parent mesh in addition to the custom child data.

HRESULT LoadMeshChildData(LPD3DXMESHCONTAINER *pMeshContainer*,
LPD3DXFILEDATA *pXofChildData*);

The following X file snippet demonstrates a custom template embedded in a standard mesh object.

```
Xof 0303txt 0032

Frame Root
{
  FrameTransformMatrix
  {
    1.000000, 0.000000, 0.000000,  0.000000,
    0.000000, 1.000000, 0.000000,  0.000000,
    0.000000, 0.000000, 1.000000,  0.000000,
   50.000000, 0.000000, 50.000000, 1.000000;;
  }

  Mesh MyMesh
  {
    6;                  // Number of vertices
   -5.0;-5.0;0.0;,      // Vertex list ( Vector array )
   -5.0;5.0;0.0;,
    5.0;5.0;0.0;,
    5.0;-5.0;0.0;,
```

```
   -5.0;5.0;-5.0;,
   -5.0;5.0;5.0;;

    2;             // Number of faces
    3;0,1,2;,      // MeshFace Array
    3;3,4,5;;

    TestTemplate MyFirstValues
    {
      1.000000;
      2.000000;
      3.000000;
    }
  } // end mesh
} // end frame ( top level object )
```

9.4.5 Using D3DXLoadMeshHierarchyFromX

Our previous discussions may lead you to believe that D3DXLoadMeshHierarchyFromX is dreadfully complicated. While it is certainly more involved than D3DXLoadMeshFromX, it is still relatively straightforward, especially when you are not processing custom data objects. Once you have derived a suitable ID3DXAllocateHierarchy class to handle frame and mesh allocation, you can load a frame hierarchy stored in a file like so:

```
CAllocateHierarchy Allocator;
D3DXFRAME *pRootFrame = NULL;

D3DXLoadMeshHierarchyFromX("Example.x", D3DXMESH_MANAGED,
                           pDevice, &Allocator,
                           NULL, &pRootFrame, NULL );
```

When this function returns, pRootFrame will point to the root frame of the hierarchy contained in "Example.x". We passed NULL for the pUserDataLoader parameter so that custom data objects will be ignored. We also passed NULL for the ppAnimController parameter since we will not require animation data in this example. We will discuss the ID3DXAnimationController in the next chapter. Note that we have passed D3DXMESH_MANAGED for the mesh creation flag to indicate our desire for managed resources.

> **Note:** The Options passed to D3DXLoadFrameHierarchyFromX are not adhered to as closely as you might like. For example, in one of our testing scenarios, passing D3DXMESH_MANAGED resulted in meshes created with the D3DXMESH_SOFTWARE option (with a dramatic performance hit). To work around this, as you will see in the workbook source code, we forcibly set the Options inside the derived CreateMeshContainer method by cloning the mesh when the options passed by D3DX are not what we asked for.

It is also worth noting that there is a D3DXLoadMeshHierarchyFromXInMemory function available as well. As we have come to expect from the D3DX library, any function that loads data from a file usually has sibling functions that have the same semantics, but different data sources (memory, resources).

9.5 X File Notes

Before wrapping up our X file discussions, there are a few things to keep in mind when working with X files:

- You **must** include the decimal point when working with floating point values in a template. For example, specify '2.' or '2.0' but not '2'. If you do not, the value will be imported incorrectly.

- Be aware of structure padding for byte alignment. This is important to bear in mind when copying data from an IDirectXFileData object into an application defined structure. For example, consider the following structure:

```
struct TestStruct{
    UCHAR foo;
    ULONG bar;
};
```

  You might expect that if we were to do sizeof(TestStruct) that it would return 5, because the structure contains a single ULONG (4 bytes) and a single UCHAR (1 byte). In fact, sizeof would return 8 because by default, structures are byte aligned to make them more efficient. So in memory, our structure looks more like:

```
struct TestStruct{
    UCHAR foo;
    // Three Padding Bytes Here
    ULONG bar;
};
```

  Thus, when laid out in a format similar to the current example, structures are aligned based on the largest member contained therein. Another example:

```
struct TestStruct{
    UCHAR foo;
    // Seven Padding Bytes Here
    __int64 bar;
};
```

  The above structure would return a value of 16 from sizeof, even though we only directly use 9 bytes with our member variables. This is in contrast to our templates -- when D3DX loads a template with members similar to the above structures, it will not pad the data. This means that in some cases we can not simply 'memcpy' the data over, but will need to copy the members individually.

- The SDK documentation states (in places) that restricted template declarators should specify a GUID within square brackets. This is not true -- they should in fact be enclosed in 'less than' and 'greater than' chevron symbols as explained earlier.

9.6 Rendering the Hierarchy

Rendering a D3DX based hierarchy requires stepping through the frames in a particular order. Our function will iterate recursively from frame to frame, combining frame matrices as it goes. We can render meshes stored in the frame in this recursion, or we can alternatively choose to separate the world matrix update pass from the rendering pass. Here is an example of both updating the matrices and rendering in a single pass through the hierarchy:

```
void RenderFrame ( LPD3DXFRAME pFrame, const D3DXMATRIX * pParentMatrix )
{
    // Used to hold world matrix for immediate child meshes of this frame
    D3DXMATRIX WorldMatrix;

    // If there is a parent matrix then combine this frame's relative matrix with it
    // so we get the complete world transform for this frame.
    // Otherwise, this is the root frame so this frame's matrix is
    // the world matrix for the frame (just copy it)
    if ( pParentMatrix != NULL)
      D3DXMatrixMultiply(&WorldMatrix, &pFrame->TransformationMatrix, pParentMatrix);
    else
      WorldMatrix = pFrame->TransformationMatrix;

    // Now we need to render the linked list of meshes using this transform
    pD3DDevice->SetTransform( D3DTS_WORLD, &WorldMatrix );
    for ( pMeshContainer = pFrame->pMeshContainer;  pMeshContainer;
          pMeshContainer = pMeshContainer->pNextMeshContainer )
    {
         RenderMesh ( pMeshContainer );
    }

    // Process Sibling Frame with the same parent matrix
    if ( pFrame->pFrameSibling )
        RenderFrame (pFrame->pFrameSibling, pParentMatrix);

    // Move onto first child frame
    if ( pFrame->pFrameFirstChild )
        RenderFrame ( pFrame->pFrameFirstChild, &WorldMatrix);
}
```

In the above code, the RenderMesh() function is assumed to be a function that handles the actual drawing of the mesh. This can be a function that loops through each subset of the mesh, sets the relevant textures and material for that subset and then calls DrawSubset, or one that sets aside the mesh in some sort of render queue for additional sorting prior to the actual polygon drawing routines. If you choose the latter approach, just remember to store the world matrix for later access (and skip the SetTransform call since it is an unnecessary expense in that case).

To kick off the process above, we would call the function from our application render loop, passing in a pointer to our hierarchy root frame and NULL for the initial matrix pointer.

```
RenderFrame (pRootFrame, NULL);
```

Alternatively, we could pass in a standard world matrix (assuming the root frame matrix is set to identity by default – often a good idea) to move or orient the hierarchy in the world or with respect to some other parent if desired (like our car transporter example discussed earlier). Assuming our hierarchy was wrapped in some application defined object:

```
RenderFrame (object->pRootFrame, &object->world_matrix);
```

Also keep in mind our earlier discussion regarding intersection testing (ex. frustum culling). As mentioned, intersection testing is much more efficient when we use hierarchical spatial data structures due to the bounding volume relationship that ensues (parents completely enclose their children).

Let us look at dividing our rendering strategy into two passes. The first pass will update the world matrices for each frame. This would also be an ideal place to run any animation controllers if they were attached directly to nodes, but we will not worry about that for the moment. The second pass will handle the actual drawing calls, but will include simple visibility testing. Note that because we are separating the two passes, we will need to store the world matrix generated during the initial pass. Otherwise the render pass will not be able to properly transform the child meshes. We will assume in our example that we have derived our own frame type from D3DXFRAME called CMyFrame which adds a WorldMatrix D3DXMATRIX member for this purpose.

```
void Update ( LPD3DXFRAME pFrame, const D3DXMATRIX * pParentMatrix )
{
    // Cast to a pointer to our derived frame type
    CMyFrame *myFrame = (CMyFrame *)pFrame;

    // Could process local animations here…
    //(D3DX does things differently though as we will see)

    // If there is a parent matrix then combine this frame relative matrix with it
    // so we get the complete world transform for this frame.
    // Otherwise, this is the root frame, so this frame's matrix is
    // the world matrix for the frame -- so just copy it
    if ( pParentMatrix != NULL)
           D3DXMatrixMultiply(&myFrame->WorldMatrix, &myFrame->TransformationMatrix,
                              pParentMatrix);
    else
           myFrame->WorldMatrix = myFrame->TransformationMatrix;

     // Process Sibling Frame with the same parent matrix
     if ( myFrame->pFrameSibling )
          Update ( myFrame->pFrameSibling, pParentMatrix );

     // Move onto first child frame
     if ( myFrame->pFrameFirstChild )
           Update ( myFrame->pFrameFirstChild, &myFrame->WorldMatrix);
}
```

With our hierarchy now updated in world space, we can proceed to render it in a separate pass using the following function. Note that because the hierarchy has been updated for this render pass, all the frame structures now contain the actual world transformation matrices in their (newly added) WorldMatrix member. This function does not have to do any matrix concatenation; it just has to set the world matrix for any frame that contains meshes, and then render those meshes.

```
void Render ( LPD3DXFRAME pFrame, Camera  *pCamera )
{
     // Cast to a pointer to our derived frame type
     CMyFrame *myFrame = (CMyFrame *) pFrame;

     // If frame is not visible, neither are its children… just return
     if ( !pCamera->FrustumSphereTest( myFrame->center, myFrame->radius ))
            return;

     // Now we need to render the linked list of meshes using this transform
     pD3DDevice->SetTransform( D3DTS_WORLD, &myFrame->WorldMatrix );
     for ( pMeshContainer = myFrame->pMeshContainer;  pMeshContainer;
           pMeshContainer = pMeshContainer->pNextMeshContainer )
     {
           RenderMesh ( pMeshContainer );
     }

     // Process Sibling Frame with the same parent matrix
     if ( myFrame->pFrameSibling ) Render ( myFrame->pFrameSibling, pCamera );

     // Move onto first child frame
     if ( myFrame->pFrameFirstChild ) Render ( myFrame->pFrameFirstChild, pCamera );
}
```

In this code example, our visibility test is just about the simplest one possible. It is a very rough cull that considers partial intersections to be visible. We could instead implement a more robust system that accounts for cases where the bounding volume is fully outside, fully inside, or partially intersecting the frustum. When the volume is fully inside, no additional tests are needed for the children – they can simply be rendered directly without them having to do tests of their own. When the volume is partially inside, we can run additional tests with tighter fitting bounding volumes (perhaps an AABB) if desired. We will discuss Bounding Volume/Frustum tests again later in this course.

There are other optimizations that can be done with frustum culling as well. For example, given what we know about our parent-child relationship, we can further speed up our routine by tracking which frustum planes the parent tested against and letting the children know this during the traversal (a bit flag works nicely here). If the parent was completely inside the left frustum plane for example, so are its children and they will not need to run that test again -- only partial intersections need to be tested. This reduces the number of tests that need to be run at each node in the tree. Again, a parent being fully outside any single plane means all of its children are as well and traversal stops at that point as we saw above.

Another popular optimization is frame-to-frame coherency which adds even more value to the previous two optimizations. A simple implementation of this requires that we keep track of the last failed frustum plane tested between render calls (usually as a data member in our derived frame class). It is likely that on the next render pass, that frame will fail against the same frustum plane, so we can test it first. If the frame fails that same frustum plane test, we can immediately skip the remaining 5 (out of 6) frustum plane tests and return immediately, letting the caller know the frame is still not visible. We will be incorporating all of these concepts and more into our game engine design during the next course in this series, but you should feel free to try your hand at them sooner if you like. They really can make a big difference in engine performance.

Conclusion

This chapter has certainly been tough going, due mainly in part to the fact that the loading of data is rarely considered very engaging subject matter. We have not even learned anything that will give us instant visual gratification. However, as virtually all game engines arrange their data hierarchically, what we have learned in this chapter is a foundation upon which everything else we do will be built from this point on. Animated game characters for example (Chapter 11) will be represented as frame hierarchies, where each frame in the hierarchy represents an animation-capable joint of the character's skeleton.

As we have now learned how to load, construct, traverse, and render frame hierarchies, we now finally have the ability to load and represent entire scenes as independent objects that can be manipulated as needed.

We will use hierarchies to spatially subdivide the world to make efficient rendering possible as we will see later. In fact, frame hierarchies serve as a great introduction to the topic of scene graphs. Scene graphs are basically a more complex and fleshed out frame hierarchy implementation. We will be discussing them in module III of this course series when we leave our framework behind and migrate towards larger scale graphics engine technologies.

Simply put, you will find that we will be using frame hierarchies almost everywhere from now on. Indeed, we will begin with the very next chapter. In Chapter Ten, we will learn how to use the D3DX animation system to animate our scene hierarchies. The D3DX animation system allows us to perform some very complex tasks, from playing cut scenes using our in game graphics engine, to performing real time animation blending. Hierarchies play a pivotal role in these tasks, so you are now well equipped to move forward with confidence.