Workbook Eleven:
Skinned Meshes and
Skeletal Animation I

Introduction

In the previous lab project we extended our CActor class so that it was possible for an application to use its interface to apply animations to the frame hierarchy. Furthermore, we implemented a system such that these animated actors hierarchies could be efficiently instanced. In this lab project, we will extend CActor to provide automated transformation and rendering support for skinned meshes. Lab Project 11.1 will concentrate on extending the CActor code to support the loading and rendering of skinned meshes using vertex blending and these changes will be demonstrated in an updated viewer application. This will demonstrate our CActor object's ability to play back pre-animated scene hierarchies comprising of both static and skinned meshes. That is, both static and skinned mesh types existing together in a single frame hierarchy.

Lab Project 11.1 – Adding Skinning Support to CActor

You might assume that in order to upgrade our actor code to support the loading and rendering of skinned mesh data, we would need to create a new class; an addition to CActor that is used in special cases for handling skinned mesh data. After all, skinned meshes are processed quite differently from the regular meshes stored in a multi-mesh X file. This is certainly true in the case of actually rendering these entities. However, there is no reason why we cannot extend our actor class elegantly to support both skinned and static mesh hierarchies. This would allow our application to use a single interface for handling both types of mesh data, making the separate tasks usually performed to load and render these two different mesh types invisible to the application.

If you think about how our actor currently works, you should recognize that the changes to the code will not need to be so dramatic. Our actor currently handles the loading and rendering of single or multi mesh (non-skinned) hierarchies. With such a hierarchy, the X file will contain a frame hierarchy which in turn will contain one or more static mesh containers. Each mesh container will contain the data for a single static mesh and will be attached to a single frame in that hierarchy. When this data is loaded from the X file, the data is laid out in the same way inside the actor as we have seen in the previous lab projects.

We know that before we render our actor we must first apply any animations to the relative frame matrices of the hierarchy and then perform a traversal of the hierarchy to update the absolute matrices for each frame. At that point, we are ready to render all the meshes in that hierarchy. The current rendering strategy of our actor similarly involves another traversal of the hierarchy. Currently, when we render our actor, we start at the root frame and traverse the entire hierarchy looking for mesh containers. Whenever we find a frame with a mesh container structure attached, we know that this frame owns a mesh. At this point, we set the frame's absolute matrix as the current world matrix for the D3D device and render the mesh stored there. After we have traversed the hierarchy, all meshes managed by the actor will have been rendered.

So, what needs to be changed in order to render skinned meshes stored in a hierarchy? The first obvious difference is that the absolute frame matrix alone no longer describes the position and orientation of the

skinned mesh within the hierarchy (or world). Instead, the mesh container will contain an ID3DXSkinInfo object which will tell us the connection between vertices in the skin and frames/bones in the hierarchy. More specifically, it will tell us which matrices in the hierarchy should be used to multi-matrix transform (vertex blend) the vertices of each subset of the mesh. Before rendering each subset, we must find the absolute matrices of the frames that influence that subset. We then set these matrices on the device, enable vertex blending, and then render the triangles in that subset. The frames of our hierarchy which influence the transformation of subsets in our skinned mesh are often referred to as the 'bones' of the mesh. The absolute matrices for these frames are referred to as the bone matrices. While this often sounds a bit scary to the uninitiated, we are quite used to working with bones. They are, after all, just frame matrices going by a different name.

At first, it seems that the rendering method of the actor would need to be able to maintain two modes. If it is an actor that contains skinned meshes we should render using the slightly more complex multi-matrix blending method; if not, we render using the single matrix method that our actor has supported for the last few lessons. However, this is not quite the way we want our actor to work. We do not want our actor to be in either one mode or another because it is possible (and not at all uncommon) that we might load an X file (or procedurally build a hierarchy) that contains both static and skinned meshes all within a single hierarchy.

It may be difficult at first not to visualize a hierarchy that contains a skinned mesh, as always being limited to defining the bones (skeletal structure) of a single character or object. It is more intuitive for us to picture an actor as containing a single character bone hierarchy containing a single skinned mesh for which the the hierarchy forms its skeleton. If we were to extend our actor to only work in this way, then we would indeed be able to have an actor that works in one of two modes. However, we would find this extremely limiting and it would actually fail to correctly manage and render a large number of X files and scene configurations which contain both skinned and regular meshes all within the same frame hierarchy.

For example, in lab project 11.1, we show how an X file is loaded into our new viewer application. It is worth remembering that the X file could actually contain an entire scene and not just a single skinned mesh. That is to say, it could contain the static geometry for lots of buildings and multiple skinned meshes (for example). The X file could also contain pre-recorded animation data that when played, make the characters walk about within that scene. This is a single X file, which means it would be managed and rendered by a **single actor**. The actor will have a frame hierarchy that contains not only the frames to position the static meshes in the scene (the rooms, buildings, and tables, etc.) but also, sub-sections of that hierarchy will contain the complete skeletons for two or more character meshes (the skins). While translation keyframes are applied to the skeletons of each character to make them move around the scene independently from each other, they are still attached to the root frame of the scene and share the same frame hierarchy with other static meshes. The character skeletons are now just sub-sections of that entire scene hierarchy. The animation controller defined for this scene would only apply animations to the frames of the hierarchy that comprise the character skeletons. Therefore, the static and skinned meshes are all connected and the entire pre-recorded scene (including the character meshes) could be repositioned somewhere else in world space simply by applying a new world matrix to the actor's root frame.

One thing should be clear from the above discussion. We need to allow our actor's hierarchy to manage both skinned and static meshes and allow them to harmoniously co-exist within the same actor hierarchy. The actor's rendering logic should be extended such that it can recognize and render both types of mesh when doing a render traversal. For example, when traversing the hierarchy during a render pass, it will search for all frames that have a mesh container attached. For each one that is found, it will need to determine whether or not this mesh container stores a skinned or regular mesh and choose a suitable rendering strategy for each case.

As discussed in the textbook, when a frame hierarchy is being loaded and mesh data is attached to a frame (a relationship defined in the X file), the ID3DXAllocateHierarchy::CreateMeshContainer call-back function will be called. This function accepts an ID3DXSkinInfo interface pointer, which we have thus far ignored in previous lab projects. If this pointer is NULL, then it means the mesh currently being loaded is a static mesh and we load it in the same way we have in all previous projects. If this interface pointer is not NULL, then it means there is also skinning information defined for this mesh in the X file. Since the D3DXMESHCONTAINER structure contains an ID3DXSkinInfo pointer member, we can store this ID3DXSkinInfo interface pointer in the mesh container during the hierarchy loading/creation process. We will need it later anyway to figure out which frame matrices in the hierarchy must be used to transform its various subsets. The upshot of this is, during the rendering traversal of the hierarchy, whenever we find a mesh container, we can simply test to see if its ID3DXSkinInfo pointer member is set to NULL. If it is, then it is not a skinned mesh and should be transformed and rendered in the regular way (by simply setting the absolute matrix of the current frame as the device world matrix prior to rendering that mesh). If it is not NULL, then we know we have a skinned mesh and can deploy an entirely new rendering strategy for that mesh. With the latter case, we would use the ID3DXSkinInfo object to determine which matrices to set simultaneously on the device prior to rendering each subset of the skin.

With this type of logic in place, our application can use a single CActor to load any type of scene without having to worry about what that scene contains; it will not have to perform any special processing in the skinned case. The application may use the actor to load an X file which contains a single skinned mesh character (where the hierarchy represents a single character skeleton) for example, or it may use the actor to load an entire scene of static meshes. The actor could also be used to load an X file that contains both a mix of regular and skinned meshes. In every case, the application has no logic to perform; it simply calls CActor::LoadActorFromX and lets the actor handle the details of both loading and rendering that geometry.

An example of a scene that might contain multiple static and skinned meshes might be a forest scene. The basic hills, valleys and buildings comprising the the scene would be arranged hierarchically and would be constructed as static meshes. However, it may also contain a few trees that sway in the wind. We might decide that the trees themselves will be represented as skinned meshes and frames will be added to the hierarchy inside the tree branches to act as the branch bones. The artist might then build animations to rotate and translate those bones in a random pattern so that the branches appear to be swaying in the wind. The actor will handle this automatically and the application will use the actor in exactly the same way as it has in previous projects. The actor will still have a single frame hierarchy and will still apply animations (via its animation controller) to certain frames in that hierarchy. The fact that some of those frames (the ones being animated) are actually being skinned by branch meshes, does not effect the hierarchy representation or the way animations are applied to that hierarchy.

For example, consider a simple example of a winter tree that has a tree trunk and four branches stored in an X file.

Figure 11.1

While this might look like a single tree mesh, assume it consists of five separate meshes. The tree trunk would be one mesh and each of the four branches shooting off of the main trunk might also each be individual meshes. The artist might decide when building this tree that the branches should sway in the wind and therefore might decide to place some frames/bones inside the branches. We will imagine for now that the root frame has been placed inside the trunk. The artist could build subtle animations that would be applied to the branch frames to make them move by some small amount as a result of wind being applied. We might decide however that as the trunk of the tree itself would never actually move, we will not make it a skinned mesh like the four branch meshes. This decision might be made in an attempt to aid rendering performance as rendering a skinned mesh is more computationally expensive. Now, this is a slightly odd example because normally you would make the entire tree a single skinned mesh so that no cracks or gaps appear between the branch meshes and the trunk, but bear with us for the purposes of this example.

We know that when the above tree was exported, we would have an X file that contained a frame hierarchy. In that hierarchy there would exist five meshes -- one static mesh (the tree trunk) and four skinned meshes (the tree branches). Therefore, while the hierarchy described by the X file would essentially contain the bones of the entire tree, not all meshes contained in the hierarchy would need to be multi-matrix transformed when rendered. For the sake of this example, let us imagine the frame hierarchy for the tree was defined as shown below in Figure 11.2.

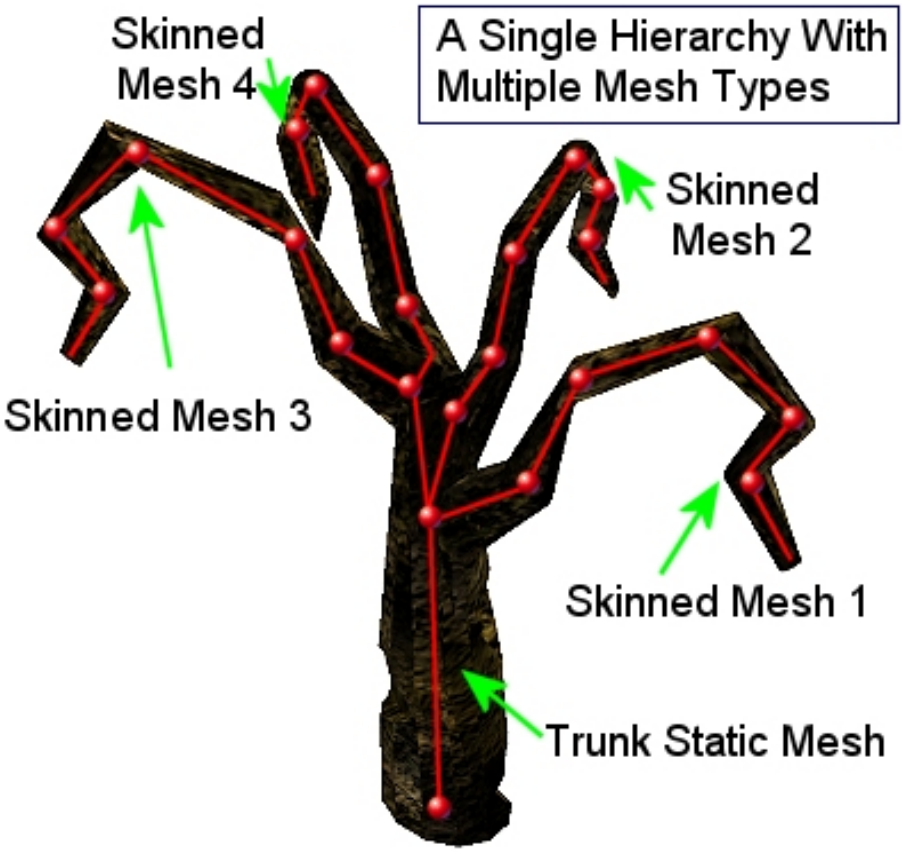

Figure 11.2

The bone (red sphere) at the bottom of the tree trunk is assumed to be the root frame in this diagram and its absolute matrix would be used to directly position the static trunk mesh prior to rendering. We might say then, that this is not a bone at all in the traditional sense. The rest of the frames in the hierarchy are bones as these will be mapped to vertices in the branches such that, when animation data rotates or translates these bones matrices, the branches of the tree will be animated. The root frame however is not a bone in the traditional sense as its absolute matrix is not used directly to transform and render any of the vertices in the branch meshes. That is, we will never set the root frame's absolute matrix while rendering the skins; we only do this when we render the trunk mesh (in which case it will be the only matrix set on the device). However, the root frame is certainly a bone in another sense as any rotations or translations applied to this bone will alter the positions and orientations of all the other bones in the tree also.

Figure 11.3 shows how a tree might be represented as a hierarchy in the X file and inside our actor. Figure 11.3 is not an accurate depiction of the bones used in 11.2, but will serve for the purpose of this example.

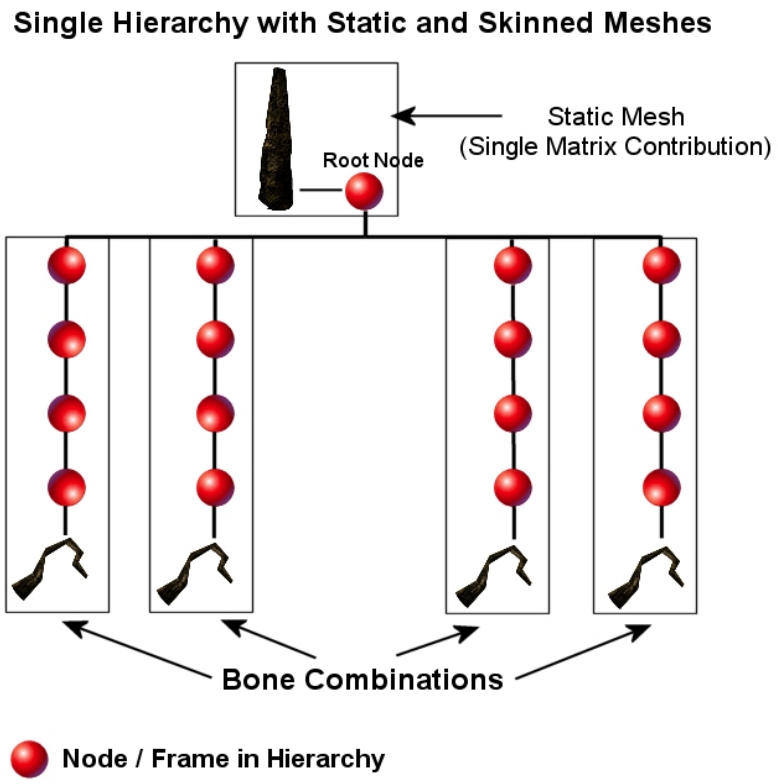

Figure 11.3

In Figure 11.3 we can see how the hierarchy looks with both its static trunk mesh and its four branch skins. In this example, the trunk mesh is in a mesh container attached to the root node. Since this is a static mesh, the absolute matrix of the root node will be used as the world matrix for rendering this static mesh. As children of the root node, there are several frames that will be used as the bones of the branch meshes. The block boxes indicate the frames in the hierarchy that influence each mesh.

Looking at this example, we can see that our actor is using a combination of static and skinned meshes. Let us walk through the rendering strategy, assuming that all animations have been applied and the absolute matrix of each frame has been updated and is now ready for the rendering traversal pass.

Our actor's rendering function would start at the root frame. It would find that there is a mesh container there, but would also discover that this mesh container contains no skinning information. This means it is a static mesh and the absolute matrix stored in the root is the actual world matrix that should be set on the device to correctly transform and render the mesh. So the matrix would be set as the current world matrix on the device and the mesh would be rendered.

We would now discover that this frame has children, so we would then traverse into each of these little sub-hierarchies. We will step through the traversal of the left-most sub-hierarchy first. We would get the first child frame and discover that no mesh container is stored there, so there is nothing to render. We would then step down into its child where the same discovery would be made again. There is still nothing to do so we would step down into the third level of this sub-hierarchy where once again, there is

no mesh data attached and nothing to render. Finally, we would step down into the fourth frame of this sub-hierarchy where the skin (i.e., branch mesh) is actually stored in a mesh container.

At this point we would test to see if this mesh container had skinning information loaded for it and would find that it does indeed have a valid pointer to an ID3DXSkinInfo interface. This means this is a skinned mesh. We would now have to loop through each subset in the mesh and set all the matrices that influence that subset in the device's matrix palette. With all the matrices that influence the subset now bound to the device, we would enable vertex blending and render the subset. In this example, the four matrices that share the black box with the branch mesh (Figure 11.3) would be sent to the device (combined with their respective bone offset matrices) and used to multi-matrix transform the vertices of the subset into world space. For both regular and skinned meshes, we must render every subset. Unlike in the regular case however, when rendering a skinned mesh, each subset may require a different set of bone matrices to be sent to the device before it is rendered. Therefore, when rendering the skinned mesh, we are no longer just concerned with batch rendering by texture and material. We must now also make sure that we also set the correct matrices for each subset that we render, giving us yet another batching key to worry about for efficient rendering.

You will see when we cover the upgraded code for CActor, that each mesh container that stores a skinned mesh will also store a bone combination table. This will be calculated at hierarchy load time (inside the ID3DXAllocateHierarchy::CreateMeshContainer method) and will contain information describing, for each subset in the skin, which matrices must be set on the device in order to render it.

So we now know that our actor should have the ability to render a hierarchy that may contain skinned and regular meshes and we should place no burden on the application to provide any support for this. We want CActor to encapsulate the hierarchy loading, animation, transformation, and rendering just like it always has, including handling the cases of skinned meshes in our scene.

For another more complex example, an X file might contain multiple tree representations like the one previously described. The skeletal structures for each tree could be combined in a single hierarchy as shown in Figure 11.4. In this example the X file contains two trees. Each tree has a static trunk mesh and four branch skins. In this example, each trunk mesh is a child of the root. This shows an example of an X file/hierarcy that contains both multiple static and skinned mesh types.

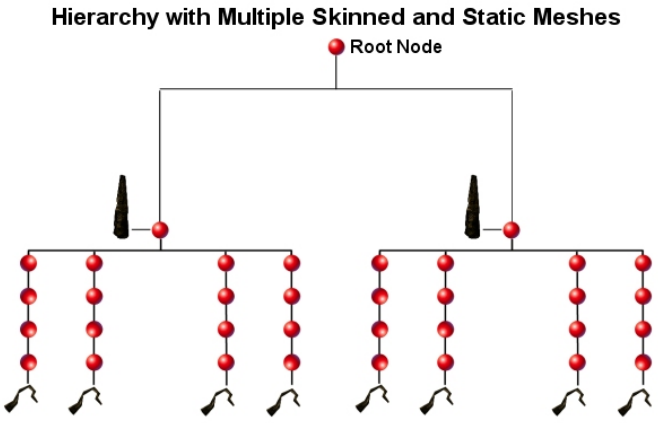

Figure 11.4

Not only should CActor handle skinned mesh support behind the scenes without burdening the application with any significant additional work, it should also support a number of different skinning techniques. We learned in the accompanying textbook that there are three types of skinning techniques we can use when performing character skinning.

We would like our actor to be able to support software skinning. While this is the slowest technique available to us (because the hardware is not utilized for purposes of transformation), it is also the only technique that imposes no limitations on the number of bones which can influence a single vertex, triangle or subset. Therefore, we may sometimes find that if we wish to accurately render a skinned mesh with an extremely complex skeletal structure and maintain its integrity to the best of our ability, we may sacrifice speed for the ability to render an object which has many bone contributions per vertex. When using hardware techniques, we can never have more than four bone influences per vertex, so software skinning support within CActor may come in handy. While it is extremely rare in real-time work that you would ever need to render geometry that uses more than four bones contributions per vertex, let us not shy away from providing that ability. We never know when we might need it.

We will also want CActor to support two types of hardware skinning that is available on most DirectX 9 graphics cards on the market today. Non-indexed skinning is usually the least desirable of the hardware techniques available because, while the hardware is utilized for the multi-matrix vertex transformations, this speed savings can be offset (sometimes) by the fact that we can only set four matrices on the device at any one time. This usually results in the source mesh being divided into many small subsets such that no single subset is influenced by more than four bones (even if the triangles share the same texture and material). This is a shame, since we know that we could usually render all triangles that share the same textures and material with single draw call. This need to render based on bone contribution creates smaller subsets and can reduce batch rendering performance.

One of the saving graces of the non-indexed case is that we can at least optimize the rendering a little by making sure that when we render a subset, we only use the weights and matrices that the subset actually needs. For example, if we render subset A which has four bone contributions, we can set the D3DRS_VERTEXBLEND renderstate such that the three weights (the fourth is created on the fly by the pipeline) are used to weight the contributions of the matrices in matrix slots 1-4 on the device. However, if subset B is only influenced by two matrices, we can adjust the D3DRS_VERTEXBLEND render state so that only one weight (the second weight is generated on the fly) is used to weight the matrix contributions of matrices in device slots 1-2 only. This way we do not waste time transforming vertices by four matrices regardless of whether a subset is actually influenced by all four of those matrices. This is a worthwhile optimization since all of our vertices will share the same vertex format. If one vertex in the mesh is influenced by four bones, every vertex in the mesh will be transformed by four weights (with space for three in its vertex structure). If a given subset is only influenced by two bones, the remaining two unused weights in the vertex structure will be set to zero, and any matrices that are assigned to slots 3 and 4 on the device will have their contribution zeroed out when the vertex is transformed. The problem with this is that without trimming the number of weights used on a per-subset basis (using the D3DRS_VERTEXBLEND renderstate), the vertices would still be transformed by all four matrices even though the weights of the vertex would zero out the contributions of the final two matrices in this example (making them unnecessary). We never want to be performing unnecessary vertex/matrix multiplications on a per-vertex level since this is very inefficient.

We also want our CActor to support hardware indexed skinning.This same optimization cannot be performed in the indexed skinning case, because the vertex weight containing the indices will always be in the same position in each vertex. Therefore, if we have an indexed skin where one vertex is influenced by four bones, the first three weights (the fourth weight is created on the fly) will always be used to weight the contributions of four matrices. The fourth weight in the vertex structure is used to hold the four matrix indices. In this example, even if a subset is only influenced by two bones (which would only require a single vertex weight in the non-indexed case), the indices will still be stored in the fourth vertex weight and the pipeline will need access to them. So we must set the D3DRS_VERTEXBLEND render state so that all weights (up to the weight that stores the indices) are used. This may cause vertex-matrix multiplications to occur unnecessarily for weights that are set to zero. Of course, this downside is often a worthy tradeoff for the fact that in the indexed skinning case, we have a much bigger matrix palette at our disposal and (potentially) the ability to have a single subset influenced by 256 matrices without having to be subdivided. The result is fewer subsets in the indexed skin and thus greatly enhanced batch rendering performance.

So it seems that our actor will need code to load and render static and skinned meshes simultaneously, and it must also have multiple paths of code for rendering skinned meshes using the three skinning techniques available. The code to load, store and render software skins, non-indexed skins and indexed skins is all slightly different and our actor will need to implement all paths.

Intelligent Auto-Detection

Not only will our actor support the three unique flavors of skinning, it should also provide a means for the application to specify which skinning technique is preferred. Of course, while it is definitely necessary in some circumstances for the application to specifically state which skinning technique should be used, in the general case, most applications would simply prefer the most efficient one to be chosen given the limitations of the hardware it is running on. So we can implement a method that would allow the application to specify the type of skinning it would like the actor to use and we will certainly do that. However, we will not stop there. We do not want to place the burden on the application to determine which skinning techniques are available on the current system before informing the actor of its choice. Therefore, we will also implement a useful auto-detection mode that will allow the application to let the actor determine the best skinning technique to use, based on the installed hardware. The application can simply load the X file and let the actor perform the hardware capability tests before choosing an appropriate skinning method. This maintains the actor's plug and play design goal. We wish to use CActor in any application without excess burden placed on the application. As far as the application is concerned, our actor should just be an object in the world that can be animated and rendered each frame (via its interface). The fact that the actor is internally a complex hierarchy of both regular and skinned meshes should not pollute the application code.

In addition to implementing an auto-detection feature in our actor (as well as a means to allow the application to specifically state which skinning mode the actor should use if it wishes), we will also implement a 'hint' system that the application may use to help bias the auto-detection mechanism of the actor. This will help it make the correct choice on the application's behalf.

Without any hints being passed to the actor by the application, the auto-detection mechanism will make the logical choice under most circumstances. If the hardware can support a mesh using indexed skinning, then this will be chosen and the loaded mesh will be converted into an indexed skin. As we know, this is usually the optimal choice. If hardware support for indexed skinning fails, then the actor will test the hardware for non-indexed skinning support. If this is supported then this will be the next best solution. In other words, if we have hardware support for both indexed and non-indexed skinning solutions for the mesh, the auto-detection mechanism will always choose indexed over non-indexed.

There may be times however when you wish to change this behavior. You may decide that you would like the actor to choose non-indexed hardware skinning over hardware indexed skinning if both are supported. Therefore, we will have several 'hint' flags that the application can pass to the actor to suggest this preference. While it may seem as if you would never want to do such a thing, there are times when non-indexed skinning can actually outperform indexed skinning due to the elimination of the zero contribution weights during rendering. Also, it may be the case that the hardware incurs a slight performance hit when performing indexed skinning due to the additional lookup into the matrix palette. You will usually find however, that the larger matrix palette and subsets resulting from indexed skinning allow it to outperform the non-indexed method.

Finally, what should our auto-detection method do if hardware support for both indexed and non-indexed skinning is absent? You might think that we would fall back to the pure software skinning mode in this case, but that is not what we will do. In fact, the auto-detection mechanism of CActor will never fall back to using the pure software skinning technique by itself. If this mode is required, it must be specifically asked for by the application. What the auto-detection mechanism will do when no hardware support for skinning is available, is use software vertex processing versions of the pipeline's skinning methods. Remember, just because the hardware does not support indexed skinning, does not mean that we cannot use DirectX's skinning pipeline. It simply means we will need to enable software vertex processing when processing this mesh such that the vertices of the mesh are transformed in software (using the host CPU) instead of by the hardware. DirectX's indexed and non-indexed skinning methods will always be available to us even if hardware support for skinning is unavailable. Therefore, what our auto-detection mechanism will do in the absence of any hardware skinning methods, is choose the pipeline's non-indexed skinning method using software vertex processing. We have generally found in our own lab tests that software non-indexed skinning is faster than software indexed skinning in a variety of cases. This is likely due to the fact that the redundant matrix multiplications with zero weight vertices in the indexed case incur a much higher performance hit now that the matrix multiplication is being done on the host CPU. However this is quite dependant on the model and as such, is not always the case. Therefore, the application will also be able to instruct the actor to choose software indexed skinning first (if desired) through the use of 'hint' flags to the auto-detection system.

We now have a path that will always locate an applicable skinning technique even if there is no hardware support. Once again though, even in the software case, hint flags can be used to bias the auto-detection process. We will have a flag that will instruct the actor that if no hardware support is available, we would prefer to use indexed skinning with software vertex processing rather than non-indexed skinning with software vertex processing (the default software technique that is used if no hardware is available).

The CActor class now has a new enumerated type that contains a number of flags. The application can pass one or more of these flags into the CActor::SetSkinningMethod member function prior to loading the actor (via CActor::LoadActorFromX). The SetSkinningMethod function simply stores the passed flags in a new actor member variable so that the LoadActorFromX method will be able to access this information when deciding what type of skin to create.

Below you can see the new SkinMethod enumeration which is now part of the CActor namespace. It contains a number of flags that can be passed into the SetSkinningMethod member function to describe how we would like the meshes to be built during the loading process.

```
enum SKINMETHOD    { SKINMETHOD_INDEXED    = 1,
                     SKINMETHOD_NONINDEXED = 2,
                     SKINMETHOD_SOFTWARE   = 3,
                     SKINMETHOD_AUTODETECT = 4,
                     SKINMETHOD_PREFER_HW_NONINDEXED = 8,
                    SKINMETHOD_PREFER_SW_INDEXED = 16 };
```

Below is a description of the various flags and how they will influence the auto-detection mechanism of the actor during loading.

SKINMETHOD_INDEXED
This flag is tells the actor that we wish to use the pipeline's indexed skinning method. This flag disables the auto-detection process. If hardware support for indexed skinning is not available, then the pipeline's software vertex processing pipeline will be used to perform the indexed skinning. This flag is mutually exclusive to all other flags.

SKINMETHOD_NONINDEXED
This flag is tells the actor that we wish to use the pipeline's non-indexed skinning method. This flag disables the auto-detection process. If hardware support for non-indexed skinning is not available, then the pipeline's software vertex processing pipeline will be used to perform non-indexed skinning. This flag is mutually exclusive to all other flags.

SKINMETHOD_SOFTWARE
Specifying this flag will disable the actor's auto-detection mechanism and tells it *not* to use the DirectX pipeline's skinning methods and to prefer the full software skinning technique instead. Specifying this flag is the only way the actor will ever use a full software skinning implementation. When in pure software mode, the DirectX pipeline will not be used in any way to transform the vertices of the skin into world space. Instead, we will perform the multi-matrix transformation of vertices from model space into world space ourselves. Rendering this mesh will require us setting the world matrix to an identity matrix since we will be passing DirectX a world space transformed skin. This is all done inside the actor's rendering code.

This is the slowest skinning method but is useful if you do not wish to have any limitations placed on the geometry. Even when using software vertex processing with DirectX's skinning methods, we are still limited to a maximum of four influences per vertex. With a pure software implementation this is not the case. We are not limited by either the number of bones that influence a vertex or the number of matrices that can be used in total. This flag is mutually exclusive to all other flags.

SKINMETHOD_AUTODETECT
This is the auto-detection mode flag. It can be combined with the final two modifier flags (shown below). This flag cannot be combined with any of the three flags described previously.

This flag places the actor into auto-detection mode; the default mode of the actor. When a mesh is loaded the actor will query the capabilities of the hardware and choose (what it considers to be) the most appropriate method. It will first try to use hardware indexed skinning. If not available, it will try to use hardware non-indexed skinning as the next best option. Finally, if hardware non-indexed skinning is not available, it means there is no hardware support. It will then fall back to using the pipeline's non-indexed skinning technique with software vertex processing.

This is the default behavior of the auto-detection process. However, you can combine any of the following flags with the SKINMETHOD_AUTODETECT flag to alter the way in which the auto-detection mechanism chooses it skinning technique:

SKINMETHOD_PREFER_HW_NONINDEXED
This is a hint flag that can be combined with SKINMETHOD_AUTODETECT. It can also be combined with SKINMETHOD_PREFER_SW_INDEXED. It changes the default auto-detection process of finding a suitable hardware method such that the actor will first favor non-indexed hardware skinning over indexed hardware skinning if both are supported. You may find with some models that non-indexed skinning may actually be faster than indexed skinning. This hint flag gives the application control in these circumstances. If this hint flag is not used, indexed skinning will always be favored in hardware.

SKINMETHOD_PREFER_SW_INDEXED
This is another hint flag that can be combined with SKINMETHOD_AUTODETECT. It can also be combined with SKINMETHOD_PREFER_HW_NONINDEXED. It changes the default auto-protection process for finding a suitable software skinning method if no hardware support is found. Be default, if no hardware support for skinning is available, it will choose the pipeline's non-indexed skinning method using software vertex processing. This flag instructs the actor, in the absence of hardware support, to favor the pipeline's indexed skinning method with software vertex processing.

These flags cover all options and provide our actor with either explicit instructions about which skinning method to use, or asks for the auto-detection mode to be used along with hint flags that can be used to further control the process. This means our actor will always be able to find a suitable skinning technique without burdening the application with any detection code. For example, in lab project 11.1, the following code is now used to load the actor:

```
CActor * pNewActor = new CActor;

pNewActor->RegisterCallback(  CActor::CALLBACK_ATTRIBUTEID,
                              CollectAttributeID, this );

pNewActor->LoadActorFromX( strFileName, D3DXMESH_MANAGED, m_pD3DDevice );
```

As you can see, nothing has changed at all. We simply allocate a new actor, set the callback method to handle the loading and storing of texture and material resources (optional), and then load the actor just like we always have. We have not even set the skinning method of the actor. So how does the actor know which method to use when loading in skinned data from the X file?

It just so happens that the skinning method of the actor is set to SKINMETHOD_AUTODETECT by default in the constructor. Therefore, since this is the mode you will probably use for your actor object's most of the time, we do not even have to bother setting it. The same code our application used to load a simple mesh hierarchy is now being used to load a hierarchy with (potentially) multiple regular and skinned meshes and animation data. Note how our changes to the actor class have not affected the way the application uses it. Our actor will even been rendered in exactly the same way as previous lab projects since it will be the actor's internal rendering code that will handle the detection and rendering of any skinned meshes contained within.

While the above code is probably how your application will most likely use the actor, it does not demonstrate the various flags we just discussed. If you wish to change the skinning method used by the actor, or bias its auto-detection mechanism from the standard path in some way, you must use these flags and the CActor::SetSkinningMethod function to inform the actor *prior to* loading the X file. Let us have a look at some examples that we might use:

Example 1

```
CActor * pNewActor = new CActor;

pNewActor->RegisterCallback(  CActor::CALLBACK_ATTRIBUTEID,
                              CollectAttributeID, this );

pNewActor->SetSkinningMethod ( CActor::SKINMETHOD_INDEXED );

pNewActor->LoadActorFromX( strFileName, D3DXMESH_MANAGED, m_pD3DDevice );
```

In this example, after creating the actor and registering its attribute callback function, we use the SetSkinningMethod to inform the actor that we definitely wish to use indexed skinning. When the actor is loaded, if hardware support for indexed skinning is available, it will be chosen. However, if no hardware support is available then the pipeline's indexed skinning technique will still be used but with software vertex processing. Even if hardware support for non-indexed skinning is available, it will not be chosen.

Example 2

```
CActor * pNewActor = new CActor;

pNewActor->RegisterCallback(  CActor::CALLBACK_ATTRIBUTEID,
                              CollectAttributeID, this );

pNewActor->SetSkinningMethod ( CActor::SKINMETHOD_SOFTWARE );

pNewActor->LoadActorFromX( strFileName, D3DXMESH_MANAGED, m_pD3DDevice );
```

This example shows the only way the application can instruct the actor to completely bypass the DirectX pipeline's skinning techniques and use a pure software implementation. This can be used for special cases where the DirectX skinning techniques would not suffice (e.g. you wish to use a model that has more than four influences per vertex, and do not want it downgraded to fit the pipeline's techniques).

This is the slowest mode when skinned meshes are contained in the actor hierarchy, because all transformations will happen in software.

Example 3

```
CActor * pNewActor = new CActor;

pNewActor->RegisterCallback(  CActor::CALLBACK_ATTRIBUTEID,
                              CollectAttributeID, this );

pNewActor->SetSkinningMethod ( CActor::SKINMETHOD_AUTODETECT |
                               CActor::SKINMETHOD_PREFER_HW_NONINDEXED |
                               CActor::SKINMETHOD_PREFER_SW_INDEXED );

pNewActor->LoadActorFromX( strFileName, D3DXMESH_MANAGED, m_pD3DDevice );
```

In this example we chose the actor's auto-detection mechanism for finding a suitable skinning method. However, rather than the default mode, we also specify two hint flags. With these two flags the auto-detection mechanism will always try to find a hardware skinning solution first. However, if both indexed and non-indexed skinning techniques are available, it will choose non-indexed hardware skinning. The second hint flag will be ignored if a hardware solution is found and will only influence the auto-detection process when searching for a software solution. The normal strategy of the auto-detection process when no hardware solution is found is to use the pipeline's non-indexed skinning technique with software vertex processing. This flag will instruct the actor to use the pipeline's indexed skinning technique with software vertex processing instead.

We now understand how we wish the auto-detection method of the actor to work. This will go a long way towards helping us understand the code when we examine it. However, before we finally delve into the CActor code, we should discuss how textures, materials, and other attribute based properties will be handled for skinned meshes within the hierarchy.

Skinned Meshes and Subset IDs

In our previous implementations of CActor, each mesh (a CTriMesh) in the hierarchy had the ability to be in either managed or non-managed mode. In managed mode, each mesh contained its own list of texture pointers and materials, and the attribute IDs of each triangle in the attribute buffer indexed into this local list. This meant that attribute IDs were local to the mesh. If the mesh had 5 subsets, the attribute IDs of each triangle have values between 0 and 4. In this mode, the mesh is self-rendering since it has everything it needs stored internally. The mesh renders itself by looping through each of its subsets, setting the texture and material in the matching position in its attribute array before rendering the subset.

This causes a slight problem when dealing with skinned meshes because after we have converted the mesh into a skin (using either ConvertToBlendedMesh or ConvertToIndexBlendedMesh) the resulting mesh will contain completely different subsets from the original mesh that was loaded. The mesh may have been subdivided into additional smaller subsets based on bone influence. The problem is that the

original per-subset attribute information is loaded by D3DX with a 1:1 mapping with subset IDs. We know that the first texture and material in the mesh's internal attribute array (texture and material table) is the texture and material for the first subset, and so on. After mesh conversion, we would now have an array of attributes stored in the mesh but we can no longer use the new subset IDs of the returned skinned mesh to index into this attribute table. The 1:1 mapping has been lost.

Imagine that subset 0 in the original mesh (which uses texture[0] and material[0]) was influenced by 8 bones and we are using non-indexed skinning and a managed mode mesh. We know that the maximum number of influences per subset in this case can be four. When we use the ID3DXSkinInfo::ConvertToBlendedMesh function to convert the original mesh to a skin, it would have detected this fact and would have split this subset into two in the resulting mesh. Therefore, what was originally a single subset with an ID of 0, has now been split into two subsets with IDs of 0 and 1 respectively. The attribute IDs of all triangles that were in the **original** second subset (assuming one existed) have been incremented so that they now belong to the third subset, and so on. This is done so that the two new subsets that have replaced the initial subset can be rendered one at a time using four matrices each, which the pipeline can handle.

Now we see the problem. Subsets 0 and 1 in the new mesh originally belonged to the same subset (subset 0) so they both must use texture[0] and material[0] in the mesh's attribute array when rendered. The problem is, in managed mode, we always used a 1:1 mapping between subset IDs and attributes in this array in our rendering logic, and this has now been compromised. To render subset 1 in the new mesh, we must still use texture[0] and material[0] and not texture[1] and material[1]. Of course, this is a simple example but in a more complex case, all the original subsets may have been divided into multiple subsets in the resulting mesh. This would mean that all of the attribute IDs of every triangle in the attribute buffer will have been changed. It would seem as if there is no longer a way for us to determine which attribute should be used with which subset.

Of course, this is not the case, because if it were, converting a mesh to a skin would be useless. As an example of how we must overcome this problem, let us imagine that we get passed a mesh in the CreateMeshContainer method that contains skinning information. Let us also imagine that this mesh originally has five subsets. Because this is a skinned mesh we will need to convert the mesh into a skin by using either the ConvertToBlendedMesh function or the ConvertToIndexBlendedMesh function depending on whether we wish to create an indexed or non-indexed skin.

Assuming we are in managed mode for this example, the first thing we might do is load the textures (using the callback) and materials and store them in the mesh's attribute array. At this point all is well because the attribute IDs of each triangle simply index into this texture and material array we have generated inside the mesh. So before we render subset[4], we would need to set texture[4] and material[4]. However, our final task will be to convert this mesh to a skin using ID3DXSkinInfo::ConvertToBlendedMesh. At this point a new mesh will be created and the original one can be released. Let us also assume that the ConvertToBlendedMesh function decided to subdivide every subset in the original mesh into two unique subsets in the resulting mesh for reasons of coping with the maximum bone influences. In the new mesh we now have ten subsets instead of five. The subset IDs of this mesh will now range from 0 – 10 even though our texture and material array still ranges from 0 – 5.

All would be lost for us at this point were it not for the fact that ConvertToBlendedMesh returns an array of D3DXBONECOMBINATION structures. There will be one element in this array for each subset in the **new** skinned mesh. In our example this means there will be 10 elements in this array. Each element in this array contains the rendering information for each subset in the new mesh. It contains an array of bone matrix indices which must be set on the device prior to rendering the subset and also provides useful information such as where the vertices and faces in this subset start and end in the vertex and index buffers respectively. This allows us to efficiently transform and render only the sections of the vertex and index buffer that is necessary when rendering a subset. Finally, and most important to this discussion, is that each D3DXBONECOMBINATION structure also contains the original subset ID that this new subset was created from. As we originally had a 1:1 mapping between the subset IDs and the attributes in the attribute buffer, this original subset ID member tells us exactly where the texture and material we need to use to render this new subset is stored in the mesh attribute array

```
typedef struct _D3DXBONECOMBINATION {
    DWORD AttribId;
    DWORD FaceStart;
    DWORD FaceCount;
    DWORD VertexStart;
    DWORD VertexCount;
    DWORD *BoneId;
} D3DXBONECOMBINATION, *LPD3DXBONECOMBINATION;
```

The AttribId member is the key. It tells us the original index into the attribute array (the array of textures and materials) that should be used to render this subset. If the original first subset (subset 0) was divided into two unique subsets in the skinned mesh, the AttribId member of the first two D3DXBONECOMBINATION structures will be set to 0. This indicates that the first two subsets should use texture[0] and material[0] in the original attribute array.

With this in mind, when rendering a skin in managed mode, our rendering strategy will need to be changed a little. We will still loop through each subset and render it; the only different now is what we consider to be the number of subsets and what we use to index into the attribute array. Previously, we knew that if a managed mesh had five attributes, then it has five subsets, and we could simply render the mesh using the following pseudo-code strategy:

```
for ( i = 0 ; i < NumSubsets ; i++)
{
     pMesh->DrawSubset[i];
}
```

In the above code, we are looking at how the actor might render the various subsets of a CTriMesh stored in a mesh container. If we remind ourselves of the code inside CTriMesh::DrawSubset we can see the rendering logic is simple:

Excerpt from CObject.cpp in previous lab projects

```
void CTriMesh::DrawSubset( ULONG AttributeID )
{
    LPDIRECT3DDEVICE9 pD3DDevice = NULL;
    LPD3DXBASEMESH    pMesh      = m_pMesh;
```

```
    // Retrieve mesh pointer
    if ( !pMesh ) pMesh = m_pPMesh;
    if ( !pMesh ) return;

    // Set the attribute data
    if ( m_pAttribData && AttributeID < m_nAttribCount )
    {
        // Retrieve the Direct3D device
        pD3DDevice = GetDevice( );

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

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

    } // End if attribute data is managed

    // Otherwise simply render the subset(s)
    pMesh->DrawSubset( AttributeID );

}
```

Take a while to examine the above function and see if you can detect the problem this function will cause going forward. This function implies the 1:1 mapping between the subset ID which is about to be rendered and the texture and material stored in the attribute array. The 'if' statement basically determines whether the mesh's attribute array exists. If it does, this then is a managed mesh and this table contains a texture and material for each subset. As you can see, the subset ID passed into the function is used to fetch the texture and material from the attribute array before making the call to the ID3DXMesh::DrawSubset function to render the triangles. If the attribute array does not exist, then this means this is a non-managed mode mesh and the scene will have already taken care of setting the correct texture and material for this subset. In this case, we simply render the subset.

We have been using this strategy for rendering managed mode meshes all along, and will continue to use the same strategy for managed regular meshes in our actor's hierarchy. However, if the mesh we intend to render is a skin, then its mesh container will contain an array of D3DXBONECOMBINATION structures, one for each subset in the new mesh. The pseudo-code to render it inside the actor would be as follows (non-indexed example). In this example, pMesh is a pointer to a CTriMesh in our hierarchy.

```
for ( i = 0; i < NumSubsets; i++ )
{
    long AttribID = pBoneCombinationArray[i].AttribId;

    pMesh->DrawSubset[ i , AttribId ];
}
```

Of course, this is a pretty vague example but we will get to the actual code soon enough. The important point here is simply that the before rendering each subset in the skin, we fetch the original ID of the subset we are about to render from the bone combination structure. This is also the index of the texture and material that the original subset used, so we can use it to access the correct texture and material in our array before rendering the subset.

In order to facilitate this change, we will slightly modify our CTriMesh::DrawSubset method. Previously, this method had a single parameter, an attribute/subset ID. As we have seen, this is used to fetch a texture and a material from the mesh's array and set it on the device prior to rendering the requested subset. However, in the above example, we can see that when the actor is rendering a skinned mesh, it needs to pass two parameters into the CTriMesh::DrawSubset. The first parameter should be the attribute/subset ID as always. This is the subset in the underlying ID3DXMesh whose triangles we wish to render. However, now the subset IDs no longer match the position of their respective textures and materials in the managed mesh's attribute array. So, as the second parameter, we pass in an attribute override. In other words, the first parameter describes the physical subset in the mesh we wish to render, while the second parameter contains the index of the texture and material in the managed mesh's attribute array. Therefore, what the above code is doing, is looping though each subset in the new skinned mesh, and using its original subset ID (from the non-skinned mesh) to fetch the texture and material that should be used to render the subset.

Let us have a look at the new version of the CTriMesh::DrawSubset method and how this additional parameter is used. This additional parameter has a default value of –1, meaning no material override will be used and the 1:1 mapping between the subset ID and the attribute table is maintained as before. This means the actor can render managed mode regular meshes in exactly the same way as our prior lab projects simply by not passing in the second parameter.

The new CTriMesh::DrawSubset Method

```
void CTriMesh::DrawSubset( ULONG AttributeID, long MaterialOverride /* = -1 */ )
{
    LPDIRECT3DDEVICE9 pD3DDevice = NULL;
    LPD3DXBASEMESH    pMesh      = m_pMesh;
    ULONG             MaterialID = (MaterialOverride < 0) ?
                                    AttributeID : (ULONG)MaterialOverride;

    // Retrieve mesh pointer
    if ( !pMesh ) pMesh = m_pPMesh;
    if ( !pMesh ) return;

    // Set the attribute data
    if ( m_pAttribData && AttributeID < m_nAttribCount )
    {
        // Retrieve the Direct3D device
        pD3DDevice = GetDevice( );

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

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

    } // End if attribute data is managed

    // Otherwise simply render the subset(s)
    pMesh->DrawSubset( AttributeID );
}
```

As you can see, the new CTriMesh::DrawSubset code is almost identical except for the following line:

```
ULONG MaterialID = (MaterialOverride < 0) ? AttributeID : (ULONG)MaterialOverride;
```

What is happening here is quite simple. We are simply saying that if the material override is –1, then there is no material override and we should assume a 1:1 mapping between the subset ID and the element in the attribute array which contains the subsets texture and material (managed mode only). However, if a material override has been passed in (which will always be the case for a skinned mesh) then we are stating that the material override value should be used to index into the attribute array to fetch the correct texture and material for the subset being rendered.

If we have the actor in non-managed mode, things are quite different for skinned meshes. With regular meshes, we simply passed the loaded texture and material to an attribute callback function. This function (implemented by the CScene class in our code) would store the texture and material in some external array and would return the index of the element in this array back to the mesh. The mesh would then lock the attribute buffer of the mesh and would re-map all faces in the current subset being processed to this new ID value. The non-managed mesh did not store the textures or materials and actually had no idea what the global ID assigned to its triangles even meant. It doesn't matter though because the scene will handle setting the correct textures and materials before rendering those subsets, not the mesh itself. It was possible (as we have previously discussed) that after the global re-mapping of a non-managed mesh, its subset IDs might no longer even be zero based. For example, a mesh which was loaded from an X file with three subsets (0, 1, 2) might be remapped by the scene such that the subset IDs end up being (75, 104, 139). These are just arbitrary numbers as far as the mesh is concerned, but to the scene object, these are global IDs into its own texture and material arrays. In this example, the scene would know that the first subset would need to be rendered using texture[75] in its scene global texture array. There may be other non-managed meshes in the scene which also have subsets with an ID of 75. The scene would know that all of these subsets could be batch rendered because they all share the same texture. In other words, the re-mapping makes the subset IDs of a mesh global instead of local, allowing shared subsets across mesh boundaries.

While none of this is new to us, when an actor is in non-managed mode, we must handle skinned meshes very differently from regular meshes (which will still use this same strategy). We will still have the non-managed mode actor working exactly the same from the applications perspective and this will still allow us to re-map the skinned meshes subsets to use a global pool of textures and materials stored at the scene level. However, unlike the case of a regular mesh, we definitely must *not* lock the skinned mesh's attribute buffer and change the subset IDs of its triangles. Remember, the zero based subset IDs in the skinned mesh have a 1:1 mapping with the D3DXBONECOMBINATION array (calculated when the conversion into a skinned mesh happens), so if we go blindly fiddling with the attribute IDs of triangles in a skinned mesh, we will lose the ability to determine which D3DXBONECOMBINATION structure in its array is applicable to each subset in the skin. As a result, we will no longer know which bone matrices must be bound to the device prior to rendering eaxch subset. No, we must leave the actual subset IDs alone in the skinned mesh case so that we know that BoneCombination[5] for example, describes the bone matrices that must be set before we render subset[5] in the skinned mesh.

Of course, all is not lost. In fact, when you think about what we are trying to achieve you find that in many ways handling global subset IDs in the skinned case is actually easier than in the regular case. The whole point of this mode is really just to allow the mesh to use textures and materials stored in the

scene's database. We can do that simply by re-mapping the AttribId member of each bone combination structure instead of having to re-map the attribute IDs of the triangles themselves.

As a quick example, let us assume that our actor is in non-managed mode. We know that this will mean that that an application defined attribute callback function will have been registered with the actor and used to re-map all the attribute IDs into global IDs. The following snippet of code shows how after a mesh is loaded, if it is in non-managed mode, we loop through each subset in the mesh and call the attribute callback function. This function is called for each subset and passed the texture and material which the scene can store somewhere. The function will return a new global ID for that subset. This is a snippet from the CreateMeshContainer method we have been using in previous projects. In this code, you are reminded that we collect all the new global subset IDs in the local pAttributeRemap array. At the end of this code, each element in this array will describe the new global ID's of each subset.

```
for ( i = 0; i < NumMaterials; ++i )
{
   if ( Callback.pFunction )
   {
      COLLECTATTRIBUTEID CollectAttributeID = (COLLECTATTRIBUTEID)Callback.pFunction;
      AttribID = CollectAttributeID( Callback.pContext, pMaterials[i].pTextureFilename,
                                     &pMaterials[i].MatD3D, &pEffectInstances[i] );

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

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

    } // End if callback available

} // Next Material
```

In our previous projects, once this re-map array has been compiled, our next task is to lock the attribute buffer of the mesh and re-map the IDs of each triangle in the mesh so that they now use the global IDs.

Re-mapping the attributes of a regular mesh

```
ULONG * pAttributes = NULL;

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

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

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

// Finish up
pMesh->UnlockAttributeBuffer( );
```

The above code is not new to us and shows what we have always done and will continue to do when working with regular meshes in non-managed mode. It locks the attribute buffer, steps through each face and changes its attribute from a local subset ID to a global subset ID using the re-mapping information stored in the pAttribRemap array compiled in the previous section of code. This code will still be employed for regular meshes, but as discussed previously, if we start fiddling about with the attribute IDs of a skinned mesh, we will loose the 1:1 mapping between subset IDs and elements in the bone combination table. What we do in the non-managed skinned case is the following:

```
LPD3DXBONECOMBINATION pBoneComb =
  (LPD3DXBONECOMBINATION)MeshContainer->pBoneCombination->GetBufferPointer();

for ( i = 0; i < pMeshContainer->AttribGroupCount; ++i )
{
   // Retrieve the current attribute / material ID for this bone combination
   AttribID = pBoneComb[i].AttribId;

   // Replace it with the remap value
   pBoneComb[i].AttribId = pAttribRemap[AttribID];
}
```

In the non-managed skinned case we first fetch a pointer to its bone combination array. Remember, this array will be generated when we use the ID3DXSkinInfo::ConvertToBlendedMesh function. In this example it is assumed that this bone combination table was stored in the mesh container.

We next loop through each subset in the mesh, which is equal to the number of D3DXBONECOMBINATION structures. Instead of re-mapping the actual triangle attribute, we simply re-map the original attribute ID stored in each D3DXBONECOMBINATION structure.

So, where does this leave us? If we have an actor in non-managed mode that contains a single skinned mesh, we will end up with a mesh that has no internally stored attribute information (just as in the regular non-managed mode case). Unlike the non-managed mode regular mesh case, its subset IDs will still be local and zero based. However, what the mesh container will contain is a bone combination structure for each subset in the new mesh. Each bone combination structure will contain (inside the AttribId member) the new global ID that the scene can use to identify the texture and material that should be used to render each subset.

This might sound like a huge shift in our scene's rendering strategy if it wishes to render an actor in managed mode. However, no changes need to be implemented by the scene. Below, you can see a slightly compacted version of the scene's rendering code. For simplicity, each object in the scene is assumed to be a CActor:

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

    …

    // Process each object
```

```
    for ( i = 0; i < m_nObjectCount; ++i )
    {
        CActor    * pActor = m_pObject[i]->m_pActor;

        // Set up actor / object details
        pActor->SetWorldMatrix( &m_pObject[i]->m_mtxWorld, true );


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

           // Set the states

           m_pD3DDevice->SetMaterial( &m_pMaterialList[ MaterialIndex ] );
           m_pD3DDevice->SetTexture( 0, m_pTextureList[ TextureIndex ]->Texture );

           // Render all faces with this attribute ID
          pActor->DrawActorSubset( j );

        } // Next Attribute

    } // Next Object

}
```

This rendering code assumes that all the actors in the scene have been created as non-managed actors. As such, the scene contains global arrays of textures and materials used by all meshes in all actors in the scene. As you can see, our rendering strategy has not changed. We simply loop through each actor and set its world matrix to position it correctly in the scene. For each actor, we loop through every global attribute combination managed by the scene and set the corresponding texture and material for that global attribute combination. Finally, we pass the index of that global attribute ID into CActor::DrawActorSubset where we expect the matching triangles to be rendered with this single call.

When we had only regular meshes in the actor's hierarchy, the CActor::DrawActorSubset method was quite straightforward. It would simply traverse the hierarchy looking for mesh containers. Once a mesh container had been located the global ID passed into the DrawActorSubset method could be directly passed to the ID3DXMesh::DrawSubset method to render the corresponding triangles. This is because triangle attributes IDs would have been mapped to these global IDs. However, with non-managed skins a slightly different rendering approach will be needed since the link between the subset we wish to render and the global attribute ID is in the AttribId member of each element in the D3DXBONECOMBINATION array.

When we wish to draw the subset of a non-managed skin, we will loop through each of its bone combination structures and find the subset that uses this global ID (if it exists). Below we see a very simplified rendering example which will be fleshed out later when we cover the actual source code. It demonstrates how the actor would render the requested subset for a skinned mesh. AttributeID is assumed to contain the subset ID passed in by the scene describing the global subset it wishes to render.

```
LPD3DXBONECOMBINATION pBoneComb =
      (LPD3DXBONECOMBINATION)MeshContainer->pBoneCombination->GetBufferPointer();

for ( i = 0; i < pMeshContainer->AttribGroupCount; ++i )
{
     if (pBoneComb[i].AttribId != (ULONG)AttributeID )
          continue;

     // Set up the bone stuff here ( shown later )

     // Draw CTriMesh
     pMesh->DrawSubset( i , pBoneComb[i].AttribId );
}
```

In this example we are showing special case code to render the subset of a skinned mesh. What is important about this code is that it works for both managed and non-managed modes. To understand why, let us step through what is happening.

We first fetch the bone combination array and loop through each element (subset) stored there. For example, let us imagine that we wish to render global attribute ID 57. We loop through each bone combination structure and test to see if the current subset being processed has 57 in the AttribId member of its bone combination structure. If not, then we skip the current subset and continue the search. Once we do find a subset that has a 57 match, we call the CTriMesh::DrawSubset method which we looked at earlier in this lesson. This function is the means by which this all works out nicely. As the first parameter we pass in the subset we wish to render and as the second parameter we pass in the AttribId member as the material override. If this is a managed mode mesh, then the material override is used to select the correct texture and material in its internal attribute array and send it to the device prior to rendering the subset. If this is a non-managed mode mesh then the material override will be ignored. For example, if we find that subset 4 in this mesh has 57 in the AttribId member of its bone combination structure, this simply means that we need to render subset 4 of the mesh (the material and texture have already been set by the scene). This will all come together very quickly when we look at the new rendering and loading code for CActor later in the workbook.

Summing up, the application's interaction with the actor will not change from earlier projects, even when skinned meshes exist in the hierarchy. The actor loading and rendering code will take care of rendering the special cases. It will use different rendering and loading code if a mesh is a regular mesh or a skinned mesh as well as special case code for managed and non-managed flavors.

Non-Indexed Skinned Mesh Rendering Optimization

As discussed in the textbook, we will sometimes be confronted with a mesh that has some subsets that can be rendered in hardware and others that can only be rendered in software. In our rendering code, we will have to render these meshes using a two pass approach. In the first pass we could disable software vertex processing and loop through each subset in the mesh. For each subset, we would test how many bones influence it, and if it is smaller or equal to the number supported by the hardware, we would render the subset. After we have looped through every subset, we will have rendered all subsets in the mesh that were hardware compliant using hardware acceleration. In the second pass, we would enable

software vertex processing and do a search for subsets that need to be rendered in software, Once again, we would loop through every subset in the mesh, but this time we would count the bone influences of each subset and only render subsets where the number of bone influences is larger than the number supported by the hardware. By the end of this second pass, all subsets will have been rendered.

While this approach is certainly more efficient then just rendering all subsets with software vertex processing, it does require doing two conditional passes through each subset of the mesh. Furthermore, in each pass, for each subset, we need to count the number of bones that influence that subset. This involves counting the number of Bone IDs in the subset's corresponding bone combination structure. All of this looping, counting, and testing is less than ideal for a time critical function, so we will implement a separate re-map buffer in the non-indexed case that contains the subset IDs batched by hardware compliance. For example, imagine that we have a non-indexed skin with ten subsets. Also imagine that instead of performing the tests mentioned above, we perform them at startup when the mesh is first created. Let us assume that in this example, we learn that subsets 2, 4, 5 and 8 have more bone influences than the hardware can handle.

Without optimization, our render logic would look something like this:

```
// First Pass ( Hardware Subset Pass )
pDevice->SetSoftwareVertexProcessing( false );

For ( i = 0; i < NumAttributes; i++)
{
    If ( Subset[i] == Hardware Compliant ) Render Subset[i];
}

// Second Pass ( Software Subset Pass )
pDevice->SetSoftwareVertexProcessing ( true );

For ( i = 0; i < NumAttributes; i++)
{
    If ( Subset[i] != Hardware Compliant ) Render Subset[i];
}
```

This approach is less than optimal. We can perform these tests when the mesh is first created so we should be able to store the subset IDs in an additional array grouped by hardware/software compliance. In fact, that is exactly what we will do. When the mesh is first created, we will loop through the subsets in two passes. In the first pass we will search for all subsets that are supported in hardware and add them to this additional re-map array. In the second pass we will search for all subsets that are not supported by hardware and add them to the array as well.

Using our current example, if a mesh has ten subsets, we will need a re-map array that contains ten subset IDs. As mentioned, subsets 2, 4, 5 and 8 cannot be rendered using hardware. Therefore, in the first pass, we would add subsets IDs 0, 1, 3, 6, 7, and 9 to the re-map array, and in the second pass subset IDs 2, 4, 5 and 8 would be added. The sorted array of subset IDs would now look like this:

[0, 1, 3, 6, 7, 9, 2, 4, 5, 8]

All we are doing is using this array to store the subset IDs in an optimal rendering order. Additionally, our mesh container can store the index into this array where the software subsets start. In that case, our rendering logic for a non-indexed mesh would now look like the following.

```
// First Pass ( Hardware Subset Pass )
pDevice->SetSoftwareVertexProcessing( false );

For ( i = 0; i < SoftwareStartIndex; i++)
{
    Render Subset[ Remap[I] ];
}

// Second Pass ( Software Subset Pass )
pDevice->SetSoftwareVertexProcessing ( true );

For ( i = SoftwareStartIndex; i < NumAttributes; i++)
{
    Render Subset[ Remap[i] ];
}
```

Although it looks like we do two passes here, we only ever have to loop through the entire subset array once. In the above code, it is assumed that SoftwareStartIndex was calculated at mesh creation time and contains the index into the re-map array where the software subsets begin. Remap is assumed to be the array that contains the ordered subset IDs. We thus loop through each subset but use the re-map array to fetch the ID of the subset that must be rendered next.

We have now discussed many of the hurdles we must overcome to implement skinned mesh support in CActor. While some of this information may be a bit vague or seem very complicated at the moment, our detailed walkthrough of the source code should give you all the insight you need.

Lab Project 11.1 - Implementation Goals

1) Extend CActor to provide the loading and managing of hierarchies that contain skinned meshes.

2) Allow CActor to handle the loading and managing of hierarchies that contain mixed geometry types. The actor should be able to render itself even if the hierarchy contains multiple static and skinned meshes.

3) Encapsulate skinned mesh support inside CActor such that providing skinned support does not place any additional burden on the application. The application should be able to simply load the X file using CActor::LoadActorFromX like it always has and have the details of loading and rendering the different geometry types handled behind the scenes.

4) Implement our skinned support such that managed and non-managed actor modes are still available to the application.

Lab Project 11.1 - The Source Code

Most of the changes to the CActor class will be added to existing functions. The primary changes will show up in functions which load and render the actor. These changes are focused around the CAllocateHierarchy::CreateMeshContainer method which is where the code that takes the loaded mesh and coverts it into a skin is located. CActor::DrawSubset and CActor::DrawMeshContainer are also going to be enhanced to facilitate multiple rendering techniques for regular meshes and skinned meshes of various flavors (software, indexed, and non-indexed). A few new helper methods will also be added to CActor to assist in the building process.

Our New Mesh Container

Before we look at the CActor class declaration, we first need to extend our derived D3DXMESHCONTAINER structure. As you will recall, each frame that has a mesh attached will have that mesh stored in a mesh container. Previously this mesh container has had to hold nothing more than a pointer to a CTriMesh, but now we have much more per-mesh information to worry about. For example, a skinned mesh will need to have access to the bone combination table so that it knows which matrices to use when transforming each of its subsets. It will also need to store two arrays of matrix information. As discussed in the textbook, in order to render a skinned mesh, we will need access to each of its bone matrices. The bone matrices are the absolute frame matrices in the hierarchy which will be updated every time an animation is applied to the bones. We do not want to have to traverse the hierarchy each time we need to collect the bones before we set them on the device, so we will traverse the hierarchy at load time and store matrix pointers to each bone in the mesh container. We will also store (in a separate array) the bone offset matrices for each bone used by the mesh. There will be a 1:1 mapping between the bone matrix pointer array and the bone offset matrix array such that, if a subset we are about to render uses bone [5], we will multiply bone[5] with bone offset[5] and set the resulting matrix on the device. There are many more members we will need to add, so let us have a look at our new D3DXMESHCONTAINER derived structure.

```
struct D3DXMESHCONTAINER_DERIVED : public D3DXMESHCONTAINER
{
    CTriMesh   * pMesh;
    bool         Invalidated;
    ULONG        SkinMethod;
    D3DXMATRIX * pBoneOffset;
    D3DXMATRIX** ppBoneMatrices;
    DWORD        AttribGroupCount;
    DWORD        InfluenceCount;
    LPD3DXBUFFER pBoneCombination;
    ULONG        PaletteEntryCount;
    bool         SoftwareVP;
    ULONG      * pVPRemap;
    ULONG        SWRemapBegin;
    LPD3DXMESH   pSWMesh;
};
```

Let us discuss the members of this now quite large structure. Remember, the members you see above are in addition to the members inherited from D3DXMESHCONTAINER. Every mesh in our hierarchy will be contained in one of these structures and attached to a parent frame. If the mesh container contains a regular mesh, most of these members will be unused.

CTriMesh *pMesh (All Meshes)
We have had this member in all previous versions of CActor. It stores the CTriMesh pointer which points to the mesh that will be rendered when this mesh container is encountered during the rendering pass of the hierarchy. This mesh may be in managed or non-managed mode and may contain a regular mesh or a skin. The only difference between a regular mesh and a skin is that the skin will contain vertex weights and the rest of the members in this structure will be used to determine which matrices should be set on the device prior to rendering it. However, what this mesh represents in the various different skinning modes needs some explanation.

Regular Mesh
If the mesh container stores a regular mesh then this is where that regular mesh is stored. It is this mesh that is rendered when the mesh container is encountered during the rendering pass of the hierarchy. It represents the model space mesh data that was loaded from the X file.

Non-Indexed & Indexed Skinning Mesh
If the mesh container stores a mesh that will be used as a skin by the pipeline, this CTriMesh will contain the skin in its reference pose. This is the mesh that was generated with a call to either the ConvertToBlendedMesh or ConvertToIndexedBlendedMesh functions. This mesh will be multi-matrix transformed by the pipeline during the rendering pass of the hierarchy. It does not contain the exact data loaded from the X file; instead it contains the original mesh converted into a skinned mesh. The vertices will contain weights, but the geometry will still represent the reference pose originally loaded from the X file.

Software Skinned Mesh
The use of the CTriMesh in the software skinned case may be slightly confusing at first. You will recall from the textbook that when performing software skinning we actually require two ID3DXMesh's (or at least two vertex buffers). We have a source mesh which contains the original model space mesh geometry loaded from the X file (the reference pose) and we also need a destination mesh to store the transformed vertices generated by the ID3DXSkinInfo::UpdateSkinnedMesh method. This needs to be done whenever bones in the hierarchy have been animated that would cause the shape or position of the skin to change.

Therefore, the source mesh will always contain the untransformed skin that was loaded, and the destination mesh is continually updated when transformations are applied. The destination mesh is always the mesh that gets rendered since it contains the world space version of the skin. When this mesh is rendered, the device world matrix should be set to identity.

You will see in a moment that our new extended mesh container structure stores an additional pointer to an ID3DXMesh interface called pSWMesh. This means the mesh container has the ability to store two ID3DXMesh interfaces (the other is inside the CTriMesh). We can use one mesh to store the original reference pose vertex data and another to transform into and render

from. So, in the software skinning case does the CTriMesh member contain the source mesh or the destination mesh?

The obvious choice would be to use the CTriMesh as the destination mesh since this is the mesh that we render each time we traverse the hierarchy. We could store the model space reference pose geometry in the other ID3DXMesh (pSWMesh) and transform it into the CTriMesh's vertex buffer whenever the hierarchy frames are updated. After all, in managed mode the CTriMesh object contains all the attributes it needs to render itself, so the CTriMesh should contain the transformed data and be used as the desination mesh. However, this is not really consistent with how all other mesh types that we can store in the hierarchy store their data. This makes such a design choice awkward for a number of reasons.

For starters, in both the regular and pipeline skinned mesh cases, the CTriMesh always stores the untransformed mesh (it is transformed in the pipeline). Thus, if the application wanted to explore the hierarchy and access the data of any mesh contained therein, it would expect the mesh data of the CTriMesh to be defined in model space. We would break with this approach if we stored the transformed vertex data in this object in the software skinned case only. But again, this mesh is also where the attribute data is stored and it is the mesh that we would want to use for rendering purposes. This seems to create a small dilemma.

As it happens, we will perform a little trick when rendering a software skin which will allow us to have the best of both worlds. You will recall that the CTriMesh class has an Attach method which allows us to attach and detach any ID3DXMesh as its underlying mesh data. With this in mind we will do the following:

The CTriMesh object will always house the ID3DXMesh object that contains the model space geometry in its reference pose and the additional ID3DXMesh (pSWMesh) will be the mesh that we transform the mesh data into each time the actor's hierarchy is updated. This allows us to stay consistent with how the actor works in all other modes. That is, if an application wished to retrieve the vertex buffer of any mesh in the hierarchy, it will always gain access to the model space vertex data. In the software case, the transformed mesh data will be stored in the additional ID3DXMesh pointed to by the pSWMesh variable. All is well so far, except for the fact that the standalone ID3DXMesh (pSWMesh) that contains the transformed vertices does not have any information about which subsets use which textures and materials. It also does not have the useful rendering methods that automate the setting of textures and materials in managed mode since it is not encapsulated in a CTriMesh. It is CTriMesh that provides us with these tools.

Every time the hierarchy is updated we will need to rebuild any software skins. We will use the ID3DXSkinInfo::UpdateSkinnedMesh method to transform the model space vertices from the CTriMesh vertex buffer into our destination ID3DXMesh (pSWMesh) vertex buffer. However, when it comes time to render this mesh, we will temporarily detach the CTriMesh's underlying ID3DXMesh and in its place we will attach our transformed mesh (pSWMesh). We can then use the CTriMesh methods to render the mesh with full access to all texture and materials. Once we have rendered it, we will detach the transformed mesh and restore the CTriMesh's original ID3DXMesh that contains the untransformed reference pose data. This approach addresses all or our concerns.

bool Invalidated (Software Skinning Only)
This boolean member variable is applicable only to software skinning. In the textbook we learned that when performing software skinning, we use the ID3DXSkinInfo::UpdateSkinnedMesh method to multi-matrix transform vertices from a source mesh into a destination mesh. The pipeline plays no part in the world space transformation process since we essentially hand a world space destination mesh to the pipeline for rendering. However, the problem is that if the bones in the hierarchy are updated by an animation for example, the destination mesh will need to be rebuilt. That is, we will have to transform the vertices from the source mesh into the destination mesh all over again so that the bone position changes are reflected in the new destination mesh. Of course, we do not wish to do this every time we render the mesh since this would hurt performance if we did so needlessly. We only want to do it if the hierarchy has been updated. Therefore, when an animation is applied to the hierarchy, the mesh container will become invalid. During our UpdateFrames method, for each frame we visit during the update, we will set the Invalidated member of any attached mesh container to true.

Before we render a mesh container, we will first test to see if it contains a software skin, and if so, check its validity status. If it invalid, we rebuild the destination mesh (using UpdateSkinnedMesh), set the Invalidated boolean back to false, and then render the mesh. Moving forward, when we render the actor, the destination mesh containing the world space skin will simply be rendered without being rebuilt. It will only have to be rebuilt from the source mesh data when another animation is applied to the hierarchy which invalidates the mesh container all over again. This is a handy flag since it removes any management burden from the application. The application can simply apply animations and render the actor and the actor will automatically rebuild invalidated software skins before they are rendered.

ULONG SkinMethod (All Skinned Meshes)
This member contains the skinning method being used by the mesh stored in this container. This is set using the CActor::SetSkinningMethod function prior to the hierarchy being constructed. The default mode for the actor is SKINMETHOD_AUTODETECT. However, depending on the hardware support available, the actor may create a hierarchy that contains skins created for different skinning techniques. For example, it may find that some can be created as hardware skins and others will have to be created as software skins. So this member may not be the same across all mesh containers in the hierarchy.

```
enum SKINMETHOD    { SKINMETHOD_INDEXED = 1,
                     SKINMETHOD_NONINDEXED = 2,
                     SKINMETHOD_SOFTWARE = 3,
                     SKINMETHOD_AUTODETECT = 4,
                     SKINMETHOD_PREFER_HW_NONINDEXED = 8,
                     SKINMETHOD_PREFER_SW_INDEXED = 16 };
```

This member is not used for regular meshes.

D3DXMATRIX * pBoneOffset (All Skinned Meshes)
This array is allocated to contain the bone offset matrices for each bone matrix in the hierarchy. It is used by all skinned meshes and will be populated at mesh creation time. When D3DX is loading the hierarchy and a skin is encountered in the X file, it will call our ID3DXAllocateHierarchy::CreateMeshContainer callback function passing an ID3DXSkinInfo interface pointer. The ID3DXSkinInfo object contains all the skinning information, such as which weights apply to which vertices for each matrix in the hierarchy. For each bone, it will also store a bone offset matrix.

We will extract those bone offset matrices and store them in this array so that we can access them during the rendering pass. When a subset uses bone[5] for example, it means we need to combine bone offset matrix[5] with bone matrix[5] (stored in the array below) and set this combined matrix in the appropriate matrix slot on the device prior to rendering that subset.

D3DXMATRIX ppBoneMatrices (All Skinned Meshes)**
Every frame in the hierarchy that is attached to vertices in a skin is referred to as a bone. In fact, it will ultimately be the absolute frame matrix (not relative matrix) which is the final bone matrix. Before we set a bone matrix on the device, we must first combine it with its bone offset matrix. The problem is that our bone matrices are stored in the hierarchy and would require a hierarchy traversal every time we need them to render a subset. Instead, at mesh creation time, we will fetch all the bone offset matrices from the ID3DXSkinInfo object and store them in the array described above. For each bone offset matrix, the ID3DXSkinInfo object will also contain the name of the bone (the frame name) in the hierarchy to which it corresponds. Therefore, in a one time pass, we will traverse the hierarchy and store pointers to all the bone matrices (absolute frame matrices) in this matrix pointer array so we will have direct access to them during rendering. By storing matrix pointers it also means this array will not have to be rebuilt when the matrices have animations applied to them. What we will have is a 1:1 mapping between the bone offset array and this bone matrix pointer array. This will allow us to efficiently fetch a bone matrix and its corresponding bone offset matrix before combining them and sending the result to the device.

DWORD AttribGroupCount (Pipeline Skinned Meshes Only)
This member will contain the number of subsets in the skinned mesh. This may be quite different from the number of subsets that were described in the original mesh that was passed into the CreateMeshContainer callback. Remember, the ID3DXSkinInfo::ConvertToBlendedMesh function will convert the original mesh into a skinned mesh and, when doing so, it may need to break the mesh into new subsets based on bone contribution. This member contains the final number of subsets in the mesh, which corresponds to the number of D3DXBONECOMBINATION structures in the bone combination array (described next).

LPD3DXBUFFER pBoneCombination (Pipeline Skinned Meshes Only)
When one of the pipeline skinning methods is being used by our actor, we will need to convert the original ID3DXMesh into a skinned mesh using the ID3DXSkinInfo::ConvertToBlendedMesh or ID3DXSkinInfo::ConvertToIndexedBlendedMesh functions. As we have discussed in the accompanying textbook, this function will create a new mesh with the correct number of weights allocated in the vertices. The resulting mesh will also include the weights and matrices used by each vertex, subdividing subsets where necessary.

These skinned mesh conversion functions will return an ID3DXBUFFER which contains an array of D3DXBONECOMBINATION structures. There will be one element in this array for each subset in the newly created skinned mesh. We will store the buffer in this member variable of the mesh container. We will need access to the bone combination buffer when rendering pipeline skinned meshes because each structure in the array tells us the original subset ID for each subset. We use this to index into a managed mode mesh's attribute table to fetch the texture and material for that subset. Each bone combination structure in this array also informs us of the bone matrices that need to be sent to the device for each subset. These bone indexes are stored in the BoneId array inside the D3DXBONECOMBINATION structure. For example, BoneId[2]=57 informs our rendering code that before rendering this subset, we

must fetch bone matrix[57] from the bone matrix array, bone offset matrix[57] from the bone offset array and combine them together before assigning them to matrix slot[2] on the device. For non-indexed skinned meshes, the BoneId array of each subset will, at most, contain four valid elements, as only a maximum of four matrices can ever be set on the device. For indexed skinning, the BoneId array could contain up to 256 valid elements since we have the ability (hardware permitting) to set 256 bone matrices on the device simultaneously. This member is not used for software skinning or for regular meshes.

DWORD InfluenceCount (Pipeline Skinned Meshes Only)
This member is only used by pipeline skinned meshes. It helps us set the correct D3DRS_VERTEXBLEND render state prior to rendering a given subset. The value of this member will be returned to us by the ConvertToBlendedMesh and ConvertToIndexedBlended mesh functions and has a slightly different meaning in each case:

Non-Indexed Skinning
In the non-indexed case, it informs us of the size of the BoneId array in each D3DXBONECOMBINATION structure. For example, if this value was 4, we would know that the BoneId array for every subset will have 4 elements. Not all elements will necessarily be used, as a given subset may only use 1 or 2 bones for example. In such cases, the unsused elements will be set to UINT_MAX. We need to know how large the BoneID arrays are because our code will need to search through a subset's BoneId array prior to rendering the subset to collect valid bone indices. If we did not know the size, we would not know how many elements in the BoneId array to test. The value returned from ID3DXSkinInfo::ConvertToBlendedMesh will be the maximum number of bones/matrices that influence a single subset in the mesh. This will assure us that the BoneId array will contain this many elements even though all subsets may not use them all. When rendering a non-indexed skinned mesh, we can use this value to test the BoneId array for valid bone indices and also count how many bones that actual subset is using. This allows us to set the D3DRS_VERTEXBLEND render state to the correct number of weights for each subset prior to rendering it. This allows us to rid ourselves of redundant zero weight contribution matrix multiplications.

Indexed Skinning
In the indexed case this value has a slightly different meaning although it is still used to set the D3DRS_VERTEXBLEND render state as in the non-indexed case. In order to render a subset using indexed skinning, we must correctly configure the D3DRS_VERTEXBLEND render state to correctly inform the pipeline of the number of weights in each vertex. Unlike the non-indexed case where we can adjust this render state on a per-subset basis to select only the weights actually used, in the indexed case it must be set such that it informs the pipeline of the number of total weight components our vertex has. This is because the pipeline will need to know where the Matrix Indices Component (the last weight) is stored in the vertex.

Therefore, unlike the non-indexed case, we cannot make the optimization of setting this render state on a per subset basis so that only non-zero weight matrices are multiplied. We must inform the pipeline of the exact layout of weights in the vertex structure. Since the ConvertToIndexedBlendedMesh function will create a vertex structure with enough weights to cater for the vertex with the most bone influences, every vertex will have the same number of weights and have their indices in the same position. Therefore, if this value tells us how we

should set the D3DRS_VERTEXBLEND render state, it also tells us the maximum number of bones that influence a single vertex in the mesh.

Unlike the non-indexed case, this value does not tell us of the size of the BoneId array contained in each subset's D3DXBONECOMBINATION structure. In indexed mode, the two concepts are decoupled. The number of weights in a vertex can still be 4 at most, but a single subset of triangles could index into a palette of, for example, 100 matrices. In this case then, the BoneId array for each subset would have to be large enough to hold 100 elements. So, when setting the bone matrices for an indexed subset, we would need to loop through 100 BoneId elements checking for valid bone matrix IDs. Obviously, not all subsets would use 100 matrices in this mesh, but if one subset did, the BoneId array for every subset would be large enough to hold this many elements, even if many elements were unused.

In the indexed skinning case, the size of the BoneId array is actually equal to the size of the device matrix palette that we wish to use. This is calculated in a separate step and stored in the PaletteEntryCount member of the mesh container (described next).

ULONG PaletteEntryCount (Pipeline Indexed Skinning Only)
This member will be used for indexed skinned meshes and contains the current size of the matrix palette being used on the device. This number also describes the number of elements in each subset's BoneId array (for indexed skins only) since the BoneId array must have one element for each matrix slot on the device that a vertex may index.

bool SoftwareVP (Pipeline Indexed Skinning Only)
This Boolean member is used for pipeline indexed skinning only. It is set during the mesh creation phase to inform our renderer that the device will not be able to transform this mesh in hardware and that we must enable software vertex processing before making the draw call. We will typically set this to true if we discover, during mesh creation, that there exists a face (or many faces) that have more bone influences than there are slots in the matrix palette. Unlike the non-indexed case where we may have some subsets that can be hardware transformed and some that cannot, in the indexed case, if there is a single triangle in that mesh that has more bone influences than the device has to offer, the entire mesh must be transformed using software vertex processing.

We determine whether an indexed skin can be hardware transformed and rendered by testing the maximum number of influences per face against the number of slots available in the matrix palette. For example, if there is a triangle that has 12 bone influences, but the device only support 8 matrix slots, this device cannot transform this mesh in hardware and software vertex processing must be used instead. As long as the device supports at least 12 matrices we will always be able to render the indexed skin in hardware. This is because a triangle has three vertices and each vertex can be influenced by four bones at most. As a result, we will never encounter a situation where a single triangle is influenced by more than twelve bones. As long as at least twelve matrices are available, the ConvertToIndexedBlendedMesh function will be able to divide the mesh into hardware compliant subsets (even if a given subset only has one triangle in it).

ULONG * pVPRemap (Pipeline Non-Indexed Skinning Only)
Earlier we learned that in the non-indexed case, we may have some subsets that can be rendered using hardware vertex processing and others that require software vertex processing. We found out that instead of having to perform two complete passes (hardware and software) through the all the subsets at render time, we could determine (at mesh creation time) which subsets belong to which pool and group the subset IDs in a new array. We could loop through all the subsets, find the hardware capable ones and add them to the array first, then loop through the subsets again, find all the non-hardware capable subsets and add them to this same array. At the end of this process, we would have all the hardware capable subsets stored at the beginning of the array followed by the software ones. When rendering the non-indexed skin, we can simply render all the hardware capable subsets stored at the beginning of the array first, then enable software vertex processing and render the rest of the subsets in the array. This provides a nice optimization in our rendering code beyond what we discussed in the textbook.

This member will point to an array of subset IDs in the mesh, ordered into two groups (hardware and software subsets). The following member of the mesh container will store the index into this array where the software subsets begin so that during rendering we know exactly when to enable software vertex processing. This will eliminate all hardware compliance tests that were originally performed per-subset in the rendering code examples we discussed in the textbook.

This member is used only by non-indexed skins, since we can never render an indexed skin using a mixture of hardware and software subsets.

ULONG SWRemapBegin (Pipeline Non-Indexed Skinning Only)
This member is used alongside the array described previously. It stores the index into the pVPRemap array where the software subset IDs begin. We know that all subsets in the array preceding this index can be rendered with software vertex processing disabled, while all subsets starting from this index (to the end of the array) must be rendered with software vertex processing enabled.

This member is used only by non-indexed skins, since we can never render an indexed skin using a mixture of hardware and software subsets.

LPD3DXMESH pSWMesh (Software Skinning Only)
When performing pure software skinning we will need to transform the model space mesh data into world space ourselves prior to sending the vertex data to the pipeline for rendering. We perfom this task using the ID3DXSkinInfo::UpdateSkinned mesh. This method takes an array of bone and bone offset matrices, a source mesh, and a destination mesh. The function uses the passed matrix information to multi-matrix transform the model space vertices in the source mesh into world space vertices in the destination mesh. The destination mesh then contains the geometry that we pass to the pipeline. Therefore, unlike other skinning techniques, where transformation is performed in the pipeline, when performing software skinning we need an additional mesh (or at least an additional vertex buffer) to receive the results of the world space transformation.

This member (pSWMesh) is the destination mesh when software skinning is being used. Everytime the hierarchy is updated and the bones of the mesh have changed, we rebuild the world space version of the mesh using the ID3DXSkinInfo::UpdateSkinnedMesh method. The model space geometry is stored in our CTriMesh's underlying ID3DXMesh. It is this destination mesh that we render.

You will note that this is a regular ID3DXMesh and not a CTriMesh, so it would seem that we lose access to the handy rendering functions and texture and material information for each subset normally stored in the CTriMesh class. However, as discussed earlier, during rendering we will temporarily detach the CTriMesh's current ID3DXMesh and attach this destination mesh containing the world space version of the geometry. We will then render the CTriMesh normally which will use the world space geometry. After the mesh has been rendered, we will detach the destination mesh and re-attach the CTriMesh's original model space ID3DXMesh.

We have now covered the new additions to our mesh container structure. As you can see, it has now become a lot more complex with many more members. Also remember that the base class from which our mesh container is derived also has an ID3DXSkinInfo pointer member which we have not used in previous lab projects. This will now be used to store the ID3DXSkinInfo interface pointer for a mesh that is passed to the CreateMeshContainer method during the X file loading process. We will need this interface in many places, especially in the software skinning case where this interface actually contains the UpdateSkinnedMesh method that performs the software transformation of vertices from the model space source mesh into the world space destination mesh.

Source Code Walkthrough - CAllocateHierarchy

In previous lab projects, we have used CAllocateHierarchy (derived from ID3DXAllocateHierarchy) as our callback class. An instance of this class was passed to D3DXLoadMeshHierarchyFromX by our actor and its methods were used for frame and mesh container creation during hierarchy construction. You will hopefully remember that this class should implement the four functions of the base class: CreateFrame, CreateMeshContainer, DestroyFrame and DestroyMeshContainer.

You will also recall that every time a frame is encountered in the X file during the loading process, D3DX will call our CAllocateHierarchy::CreateFrame method. This will allow us to allocate the frame memory in whatever way best suits our application. We can then hand it back to D3DX for hierarchy attachment. This allows us to allocate frame structures derived from D3DXFRAME where we can add additional member variables and tailor it to our applications needs. For example, right from the beginning of our hierarchy usage, we used a derived frame structure that had an additional matrix that would be used to store the world space transform of the frame/bone. Likewise, whenever a mesh is encountered by D3DX during the rendering process, the CAllocateHierarchy::CreateMeshContainer function would be called and passed all the mesh information from the X file (e.g., the mesh geometry, the textures and materials, and any skinning information).

Our CAllocateHierarchy::CreateFrame function is unchanged in this lab project since we have no need for new members in our frame structure. However, the CAllocateHierarchy::CreateMeshContainer function will change significantly since we have to add code that creates managed and non-managed flavors of indexed skinned meshes, non-indexed skinned, meshes and software skinned meshes (in addition to the managed/non-managed standard meshes from before).

CAllocateHierarchy::CreateMeshContainer

We will cover the CreateMeshContainer function before we cover the code to CActor::LoadActorFromX. Our reason for doing this is because one of the first things the LoadActorFromX function does is called D3DXLoadMeshHierarchyFromX which then calls the CAllocateHierarchy::CreateMeshContainer method for every mesh contained in the X file. Therefore, with respect to program flow, it makes more sense to cover this function first since it is where all the mesh data is actually created, where all the bones are assigned to the skin, and where the mesh container's members are assigned their values. By the time the D3DXLoadMeshHierarchyFromX function returns program flow back to CActor::LoadActorFromX, any meshes (skins or regular) will have been created and attached to the frame hierarchy. There will be a tiny bit of additional work the actor will need to do after the hierarchy has been loaded to complete the setup process for skinned meshes, but we will get to that in a moment.

This function was already getting pretty large even before skinned support was added. Fortunately however, we have covered all of this code in detail previously and the first ¾ of the function code is unchanged from the prevous lab project. The new skinning support has been conveniently bolted onto the end of the function, for the most part. We will show the entire function in this section and discuss it. The first few parts of the function will only be briefly reviewed since we have already covered this code in previous lab projects. This function also makes calls into a few new CAllocateHierarchy helper functions, so we will cover the code to these functions immediately after.

D3DX passes this function the name of the mesh in the X file that is currently being loaded and the D3DXMESHDATA structure that contains the ID3DXMesh (which contains the geometry loaded from the file). It is also passed an array of D3DXMATERIAL structures describing the texture and material used by each subset in that mesh and an array of effect instances (one for each subset). We will ignore the effects array until Module III when we learn about rendering with effect files. The number of materials (subsets) used by the mesh and the mesh face adjacency information is also passed in. The penultimate parameter is one we have ignored in previous projects -- it is a pointer to an ID3DXSkinInfo interface which will be non-NULL when the mesh has skinning information defined in X file. Finally, we are passed the address of a D3DXMESHCONTAINER structure which, on function return, we will assign to point at the new mesh container that we are about to allocate and populate with the passed information.

```
HRESULT CAllocateHierarchy::CreateMeshContainer
                            ( LPCTSTR Name,
                              CONST D3DXMESHDATA * pMeshData,
                              CONSTD3DXMATERIAL*pMaterials,
                              CONST D3DXEFFECTINSTANCE *pEffectInstances,
                              DWORD NumMaterials,
                              CONST DWORD *pAdjacency,
                              LPD3DXSKININFO pSkinInfo,
                              LPD3DXMESHCONTAINER *ppNewMeshContainer)
{
    ULONG                       AttribID, i;
    HRESULT                     hRet;
    LPD3DXMESH                  pMesh          = NULL, pOrigMesh = NULL;
    D3DXMESHCONTAINER_DERIVED *pMeshContainer = NULL;
    LPDIRECT3DDEVICE9           pDevice        = NULL;
```

```
    CTriMesh                    *pNewMesh       = NULL;
    MESH_ATTRIB_DATA            *pAttribData    = NULL;
    ULONG                       *pAttribRemap   = NULL;
    bool                         ManageAttribs  = false;
    bool                         RemapAttribs   = false;
    CALLBACK_FUNC                Callback;

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

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

    // Store the original mesh pointer for later use (in the skinning case)
    pOrigMesh = pMesh;

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

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

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

    // Copy over the name
    // for the string belongs to the caller (D3DX)
    if ( Name ) pMeshContainer->Name = _tcsdup( Name );

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

The above section of code is essentially unchanged from previous lab projects. It first tests to see that it is not a progressive mesh that we have been passed (which we will not support in this lab project). If it is, we return immediately. We fetch the passed ID3DXMesh interface into a local pointer for ease of access and then test that this mesh has a valid FVF vertex format (once again, we exit if this is not the case). We will discuss non-FVF meshes in Module III. One minor change is the additional line that stores the original mesh pointer. We will use this pointer only when we are working with skins because the skinning process will change the mesh subsets and various D3DX utility functions will require that we have the original (pre-skinned) data available (for saving to file for example). Once these tests are out of the way, we allocate a new derived mesh container and initialize all its members to zero for safety. Then we copy the name of the mesh that was passed into the function into the 'Name' member of the mesh container. Finally, we allocate a new CTriMesh which will eventually be used to store our mesh data and attached to the mesh container.

Before attaching the passed ID3DXMesh to our new CTriMesh object, we test the FVF flags to see if the mesh contains normals. We require normals in our demo so that we can use the lighting pipeline. If no normals exist then we will clone the passed mesh into a new mesh that contains vertex normals and attach this cloned mesh to our CTriMesh object. If the passed mesh already contains normals then (as you can see in the 'else' code block), we simply attach it to our new CTriMesh.

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

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

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

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

        // Compute the normals for the new mesh
        if ( !(pMesh->GetFVF() & D3DFVF_NORMAL) )
            D3DXComputeNormals(pMesh, pAdjacency );

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

        // Attach our specified mesh to the new mesh
        pNewMesh->Attach( pCloneMesh );

        // We can release the cloned mesh interface here,
        // CTriMesh still has a reference to it so it wont be deleted
        pCloneMesh->Release();

    } // End if no vertex normal, or options are hosed.
    else
    {
        // Simply attach our specified mesh to the CTriMesh
        pNewMesh->Attach( pMesh );

    } // End if vertex normals
```

At this point we have a new CTriMesh object whose underlying ID3DXMesh contains the geometry initially loaded from the X file. The next section of code we will look at is also unchanged from the previous lab project.

We start by testing whether this actor (technically, its meshes) is in managed mode or non-managed mode. You will recall from previous lessons that a CTriMesh (or CActor) is automatically placed into non-managed mode by the registration of the CALLBACK_ATTRIBUTEID callback function. If this function is not registered (pointer is NULL), then the mesh should be created in managed mode and we set the local Boolean variable ‘ManageAttribs’ to true. If the callback has been registered, then the callback function pointer will not be NULL and the Boolean variable will be set to false.

After determining a mesh is in managed mode, the following instructs the CTriMesh object to allocate an internal attribute data array to store the textures and materials for each subset. After the CTriMesh::AddAttributeData method is called to allocate the array, we call the CTriMesh::GetAttributeData method to get a pointer to the start of this array. We store this pointer in the local variable pAttribData which will be used later to step through the mesh's attribute array and copy textures and materials. Remember, at this point, the attribute array for the managed mesh is empty.

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

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

        pAttribData = pNewMesh->GetAttributeData();

    } // End if managing attributes
```

The next section of code is the 'else' block of the conditional code shown above. It is executed if the mesh is in non-managed mode (i.e., the CALLBACK_ATTRIBUTEID function has been registered with the actor). This code is also unchanged from the previous lab projects. It basically allocates an array that is large enough to hold a global subset ID for each subset in the mesh (pAttribRemap). These global IDs are returned to us by the scene object's attribute callback function. This array is used to re-map the subsets of the mesh to use global IDs instead of local ones. In this section of code however, once the re-map array is allocated, we initialize it with the current subset IDs of the mesh. These may be overwritten with global IDs later, but this is done just so we safely initialize the array.

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

        // Default remap to their initial values.
        for ( i = 0; i < NumMaterials; ++i ) pAttribRemap[ i ] = i;
    } // End if not managing attributes
```

So, if our mesh is in managed mode, we now have a pointer to an empty attribute table that will need to be filled with texture and material information. If the mesh is a non-managed mode mesh, we have a temporary re-map array which we will use to collect global IDs for each of its current subsets.

In the next section of code (also unchanged) we process the materials for each mesh subset. In the managed case, this means extracting the material information (diffuse, specular etc) from the passed material buffer and storing it in the corresponding entry in the managed mesh's internal attribute array. If a CALLBACK_TEXTURE callback function has also been registered by the application, then it will be called to process the texture for each material/subset. Below we see the first section of the material processing loop (handling the managed case):

```
    // Loop through and process the attribute data
    for ( i = 0; i < NumMaterials; ++i )
    {
        if ( ManageAttribs == true )
        {
            // Store material
            pAttribData[i].Material = pMaterials[i].MatD3D;
            pAttribData[i].Material.Ambient = D3DXCOLOR( 1.0f, 1.0f,
                                                         1.0f, 1.0f );

            // Request texture pointer via callback
            Callback = m_pActor->GetCallback( CActor::CALLBACK_TEXTURE);

            if ( Callback.pFunction )
            {
              COLLECTTEXTURE CollectTexture =(COLLECTTEXTURE)Callback.pFunction;

              pAttribData[i].Texture =
              CollectTexture( Callback.pContext, pMaterials[i].pTextureFilename );

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

        } // End if attributes are managed
```

The above code takes care of populating the managed mesh's attribute array with the texture pointers and material information.

The second part of the loop is the 'else' code block of the conditional which processes each material for non-managed meshes. This code is also unchanged from previous projects.

```
        else
        {
            // Request attribute ID via callback
            Callback = m_pActor->GetCallback( CActor::CALLBACK_ATTRIBUTEID );
            if ( Callback.pFunction )
            {
                COLLECTATTRIBUTEID CollectAttributeID =
                                  (COLLECTATTRIBUTEID)Callback.pFunction;

                AttribID =
                CollectAttributeID( Callback.pContext,
                                    pMaterials[i].pTextureFilename,
                                    &pMaterials[i].MatD3D,
                                    &pEffectInstances[i] );

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

                // Determine if any changes are required so far
                if ( AttribID != i ) RemapAttribs = true;
            } // End if callback available
        } // End if we don't manage attributes
```

```
    } // Next Material
```

We have now processed all the materials. In the case of a managed mesh, the internal attribute table of the mesh now contains all the textures and materials the mesh needs to draw itself. In the non-managed case, we have a temporary array describing the new global ID of each subset. We have not yet applied these global IDs to the mesh data.

In the next section of code we test to see if we were passed any face adjacency information for the mesh by D3DX (which we should have). If so, then we have to copy this face adjacency information into our mesh container's face adjacency array. We never know when we might need the face adjacency information for a mesh, so it is a wise thing to store it in the mesh container just in case. However, we must make a copy of the data and not simply assign the mesh container's member pointer to point straight at it because the adjacency information we have been passed is in memory owned by D3DX and will be destroyed when the function returns. This would leave our mesh container with a dangling pointer. So we allocate a new face adjacency array and copy over the data. Notice how this new array is pointed to by the mesh container's pAdjacency member discussed earlier.

```
    // Copy over adjacency information if any
    if ( pAdjacency )
    {
        pMeshContainer->pAdjacency=new DWORD[ pMesh->GetNumFaces()*3];

        if ( !pMeshContainer->pAdjacency )
        { hRet = E_OUTOFMEMORY; goto ErrorOut; }

        memcpy( pMeshContainer->pAdjacency,
                pAdjacency,
                sizeof(DWORD) * pMesh->GetNumFaces() * 3 );

    } // End if adjacency provided
```

From this point forward, we will start to see the new code that has been added to deal with skinned meshes. Skinned meshes have to be handled very different from regular meshes in a number of different ways, as we will now see.

The next section of code is executed if the mesh we have been passed is a skinned mesh. If we were passed a regular mesh, our job is almost done since the mesh stored in the CTriMesh is pretty much exactly what we want it to be (with the exception of some attribute buffer re-mapping in the non-managed case, which happens towards the end of the function). However, if it is supposed to be a skinned mesh, then the CTriMesh's underlying ID3DXMesh will have to be converted.

The following code tests the ID3DXSkinInfo interface pointer we have been passed by D3DX. If it is not NULL, then it means this ID3DXSkinInfo object contains skinning information for this mesh and that we have encountered a skin in the X file. The first thing we will do in this case is store the ID3DXSkinInfo interface pointer in the mesh container since we will need access to the bone and vertex weight information in many places in our code. We are also careful to increment the reference count when we copy the interface pointer into the mesh container.

```
    // Is this a 'skinned' mesh ?
    if (pSkinInfo != NULL)
    {
        LPD3DXMESH pSkinMesh;

        // first save off the SkinInfo and original mesh data
        pMeshContainer->pSkinInfo = pSkinInfo;
        pSkinInfo->AddRef();
```

We also know from the textbook that the ID3DXSkinInfo object contains the bone offset matrix for every bone used by the skin. We already discussed why we need efficient access to these matrices during rendering which is why we added a bone offset matrix pointer to our mesh container. What we will do is use this pointer to allocate an array of bone offset matrices in the mesh container and copy the bone offset matrices from the ID3DXSkinInfo object into this new array.

```
        // Allocate a set of bone offset matrices, we need to store
        // these to transform any vertices from 'character space' into 'bone space'
        ULONG BoneCount = pSkinInfo->GetNumBones();

        pMeshContainer->pBoneOffset = new D3DXMATRIX[BoneCount];

        if ( !pMeshContainer->pBoneOffset )
            { hRet = E_OUTOFMEMORY; goto ErrorOut; }

        // Retrieve the bone offset matrices here.
        for ( i = 0; i < BoneCount; ++i )
        {
            // Store the bone
            pMeshContainer->pBoneOffset[i] =*pSkinInfo->GetBoneOffsetMatrix(i);

        } // Next Bone
```

The above code first determines how many bones influence this skin, using the ID3DXSkinInfo::GetNumBones method. Since there is a matching bone offset matrix defined for every bone used by the mesh, the number of bones value also tells us how many bone offset matrices there are in the ID3DXSkinInfo object. Once this array is allocated, the code loops through each bone and uses the ID3DXSkinInfo::GetBoneOffsetMatrix method to fetch the bone offset matrix for the corresponding bone offset index. This bone offset matrix is then copied into the mesh container's bone offset matrix array at the same index position. It is very important that we keep a 1:1 mapping between the indices of bone offset matrices stored in our mesh container and their indices in the ID3DXSkinInfo object since this mapping will be essential later when fetching the actual bone matrix pointers from the hierarchy.

In the next section of code we finish off the bulk of the skinned mesh processing through the use of the CAllocateHierarchy::BuildSkinnedMesh helper function. This function (which we will examine next) generates the skinned mesh from the original ID3DXMesh that we currently have. We pass this function the mesh container, the original ID3DXMesh that we have been currently working with, and also the address of an ID3DXMesh interface pointer which on function return will point to a new ID3DXMesh in skinned format. The new mesh (pSkinMesh) will contain vertex weights and possibly vertex matrix indices (in the indexed skinned mesh case). The function will also populate the new members of the mesh container that are necessary during rendering.

Upon return from the BuildSkinnedMesh function, we will have a skinned ID3DXMesh, but it will not be attached to our CTriMesh object. The CTriMesh's underlying ID3DXMesh pointer is still pointing to the original source mesh, so we will replace it so that the CTriMesh now points to the newly generated skin. Notice in the following code that the Attach method of our CTriMesh has an additional Boolean parameter that we set to true. This is because we have added a new behavior to this method. Originally, when we attached a new ID3DXMesh to our CTriMesh object, the attribute table would also be released (for managed meshes). That is, all data that currently existed in the CTriMesh was flushed. However, in this new case, we simply want to replace the geometry (the ID3DXMesh) and leave all the texture and material data intact. Therefore, if true is passed as this parameter, we simply replace the ID3DXMesh pointer and leave the attribute data of the managed mode alone. After all, the skinned mesh will still be using this texture and material data.

```
        // Build the skinned mesh
        hRet = BuildSkinnedMesh( pMeshContainer, pMesh, &pSkinMesh );
        if ( FAILED(hRet) ) goto ErrorOut;

        // pass True to the bReplaceMeshOnly parameter.
        pNewMesh->Attach( pSkinMesh, NULL, true );
        pMesh = pSkinMesh;

    } // End if skin info provided
```

At this point in the code our CTriMesh has all the correct geometry contained and will contain either a skinned or regular ID3DXMesh. The next section of code that we will execute was also found in previous lab projects, but has some additional code to examine.

As in our previous lab projects, one of the last tasks this function has to perform is the re-mapping of the mesh attribute buffer using the global IDs for each subset we compiled earlier. This is only done when the actor/mesh is in non-managed mode. You will recall that we used the attribute callback function to fetch the global IDs for each subset and stored them in the local pAttribRemap array. This array contains an entry for every subset in the original mesh. When we were dealing only with regular meshes, we would simply lock the mesh attribute buffer, loop through each face, and set it to its new global ID. For regular meshes we will still perform this same task. In fact, if we are using software skinning, we will also do the same thing because we are really just using the original source mesh and transforming into a destination mesh ourselves prior to rendering. So in the case of software skinning, the triangles will still be grouped into the same subsets as the original mesh. In the next section of code we test to see if the mesh is a regular or a software skinned mesh and also whether the mesh is in non-managed mode. If so, we adopt the strategy we have all in previous CActor versions and remap the attribute buffer.

```
    // Remap attributes if required
    if ( pAttribRemap != NULL && RemapAttribs == true )
    {
        // Is a skinned mesh ?
        if ( !pSkinInfo ||
             pMeshContainer->SkinMethod == CActor::SKINMETHOD_SOFTWARE )
        {
            ULONG * pAttributes = NULL;

            // Lock the attribute buffer
```

```
            hRet = pMesh->LockAttributeBuffer( 0, &pAttributes );
            if ( FAILED(hRet) ) goto ErrorOut;
            // Loop through all faces
            for ( i = 0; i < pMesh->GetNumFaces(); ++i )
            {
                // Retrieve the current attribute ID for this face
                AttribID = pAttributes[i];

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

            } // Next Face
```

If the mesh is a software skinned mesh then we must remember that we also have an additional destination mesh stored in the mesh container. This is the mesh that we will be transforming into when we transform the mesh into world space. Therefore, this mesh will also have to have its attribute buffer updated in the same way. After all, we want the subsets in both the source and destination meshes to be the same (the only difference between the two is that one is in model space and the other is in world space). They both use the same textures and materials for each subset, so let us reflect the global subset changes in the destination mesh as well.

The next section of code locks the attribute buffer of the destination mesh and performs a memory copy from the CTriMesh's attribute buffer (which is still locked) into the destination mesh attribute buffer. Remember, at this point we have already updated the source mesh's attribute buffer so we can just copy the attribute buffer directly into the destination mesh such that they are now identical.

```
            // In the software skinning case, we need to reflect any
            // changes made into our SW Mesh
            if ( pSkinInfo &&
                 pMeshContainer->SkinMethod == CActor::SKINMETHOD_SOFTWARE )
            {
                ULONG * pSWAttributes = NULL;

                // Lock the attribute buffer and copy the contents over
                hRet = pMeshContainer->pSWMesh->LockAttributeBuffer(0,
                                                                    &pSWAttributes
                                                                    );
                if ( SUCCEEDED(hRet) )
                {
                    memcpy( pSWAttributes,
                            pAttributes,
                            pMesh->GetNumFaces() * sizeof(ULONG) );
                    pMeshContainer->pSWMesh->UnlockAttributeBuffer( );

                } // End if succeeded
            } // End if software skinning

            // Finish up
            pMesh->UnlockAttributeBuffer( );

        } // End if not skinned
```

Finally, we unlock both the attribute buffers and our mesh is ready to go. However, all we have seen above is the section of the conditional that handles the attribute re-map for non-managed meshes that are either standard meshes or software skins.

In the next section of code we will see the 'else' code block of the conditional which is executed when we are in non-managed mode (as above), but we are using one of the pipeline skinning techniques (Indexed or Non-Indexed Skinning). Why do we need special case code for pipeline skins? Why can't we just alter the attribute buffers for these meshes to contain the new global subset IDs as we did in the regular and software skinning cases?

When we call CAllocateHierarchy::BuildSkinnedMesh, if we have chosen one of the pipeline skinning techniques, the appropriate ID3DXSkinInfo function (ConvertToBlendedMesh or ConvertToIndexedBlendedMesh) will be called to create a new ID3DXMesh which has vertex weights in the vertices. Of course, these functions do much more than just add weights to vertices; they also calculate which bone matrices each subset must use. This information is returned in the bone combination array. These functions may decide that a given subset cannot be rendered because the faces in that subset collectively use more matrix slots than the device has to offer and in this case, the function will often break the subset into multiple subsets, where each uses a smaller set of matrices that the device can handle in hardware. This is where we find our problem…

The returned mesh may have more subsets than the original mesh. In fact, this is usually the case. This is not really a major problem of course, because we are provided with a D3DXBONECOMBINATION structure for each subset in the new mesh. Each structure will contain the original subset ID that this new subset was generated from. This means when we render a managed mode skin, we still have access to the index of the original texture and material in the mesh's internal attribute array that was used by the subset that this new subset was spawned from. However, it is the non-managed mode mesh we are processing now, and we want the scene to be able to batch render this mesh using global subset IDs across mesh boundaries. But what we cannot do is physically change the attribute buffer of the skinned mesh because the local subset IDs are mapped directly to the bone combination table that was returned from the skinned mesh conversion functions.

Of course, all we are really interested in doing is providing a way for the application to call CActor::DrawSubset with the global ID that it generated when the re-map buffer was being compiled and have it all work properly. Since the scene is responsible for setting the texture and material that corresponds to this global ID, all it is really asking is that the actor draw any faces that are connected to this global ID. It is not as if this global ID is used internally by the mesh to set textures and materials as in the managed mesh case, so this is actually very easy to achieve. In fact, it takes less work than in the regular mesh case.

We will not bother altering the values of the mesh attribute buffer at all; internally, all subset IDs will remain zero-based local IDs which map 1:1 with the bone combination table. In the managed mode case, we can use the AttribId member of each bone combination table to index into the texture and materials in the attribute table of the mesh. In the non-managed mode case, we can just store the new global ID here instead:

```
        else
        {
             LPD3DXBONECOMBINATION pBoneComb =(LPD3DXBONECOMBINATION)
                       pMeshContainer->pBoneCombination->GetBufferPointer();

            for ( i = 0; i < pMeshContainer->AttribGroupCount; ++i )
            {
                // Retrieve the current attribute ID for this bone combination
                AttribID = pBoneComb[i].AttribId;

                // Replace it with the remap value
                pBoneComb[i].AttribId = pAttribRemap[AttribID];

            } // Next Attribute Group

        } // End if Skinned mesh

    } // End if remap attributes
```

Basically, we fetch the bone combination table and loop through each of its elements. There will be one element in this array for every subset in the skin. Also remember that this bone combination table will have been created in the CAllocateHierarchy::BuildSkinnedMesh function (which we will cover shortly). Each bone combination structure contains the original subset ID that this new subset was created from. However, in a non-managed mode mesh we no longer need this information since we have no intention of letting the mesh set its own textures and materials. So we will replace the current subset ID with the new global ID compiled for the original subset during the re-mapping process. Our bone combination structures now contain the global ID that was generated for the original (material-based) subset ID which the new (bone-based) subset was generated from. This connection between global IDs and local subsets in a non-managed pipeline skinned mesh and is simple, but may be a little hard to understand at first. Things will fall into place a little more when we actually see this being used in the rendering code later on.

At this point in the code, our CTriMesh is complete. It either holds a regular mesh, a software skin, or one of the pipeline skin types. We no longer need the temporary array we used to store subset re-map information for the non-managed case so we delete it. Also, if we have created a skinned mesh, then the pMesh local interface pointer (that points to our CTriMesh's underlying ID3DXMesh) can also be released as we no longer need it. We release it in the skinned case, because we incremented the skin's reference count earlier in the function when we assigned this local pointer to point at the new skin. Finally, if the CTriMesh is anything other than a software skinned mesh, we perform the usual optimizations (vertex welding and optimize in place). Do you know why we do not perform these optimizations when we have a software skin?

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

    // If this is a skinned mesh, we must release our overriden copy
    // only the CTriMesh will now reference a copy.
    if ( pSkinInfo ) pMesh->Release();

    // Attempt to optimize the new mesh ( unless software skin ).
    if ( !(pSkinInfo && pMeshContainer->SkinMethod==CActor::SKINMETHOD_SOFTWARE) )
```

```
    {
        // Optimize
        pNewMesh->WeldVertices( 0 );
        pNewMesh->OptimizeInPlace( D3DXMESHOPT_VERTEXCACHE );

        // Release the adjacency information, it is no longer valid
        if ( pMeshContainer->pAdjacency ) delete []pMeshContainer->pAdjacency;
        pMeshContainer->pAdjacency = NULL;

    } // End if not software skinned mesh
```

We do not optimize a software skin because, as we know, this will sometimes remove or move vertices and indices inside the vertex and index buffers for better cache coherency and to clean the mesh of any un-used vertices. The problem we have if we do this with a software skinned mesh, is that we need to use the ID3DXSkinInfo object to transform the mesh. This object stores all the vertex index and bone index information which is used by the ID3DXSkinInfo::UpdateSkinnedMesh method. If we start removing vertices from this mesh, all the per-vertex information stored in the ID3DXSkinInfo object will be invalidated and will no longer correctly index the actual vertices inside the mesh. We could implement some re-mapping logic here if we wanted to, but we have decided to live with the limitation in this demo. It is very rare that you will be using the actor for software skinning, so we will just put up with the fact that a software skin will not have its geometry optimized.

The next section of code is virtually unchanged from previous lab projects, although there is one small modification. Like before, it stores our new CTriMesh in the mesh container, but this time what it stores in the mesh container's MESHDATA member varies depending on whether this is a skin. If it is not, then we store a copy of the CTriMesh's underlying ID3DXMesh just as before. However, if it is a skin, then we need to store the original mesh data instead since our CTriMesh subsets have been modified during the skin creation process. This is why we made a copy of the original mesh data pointer at the start of the function. This allows D3DX functions (e.g., D3DXSaveMeshHierarchyToFile) to find the correct mesh data exactly where they expect to find it. In the managed case, we also copy over the texture names and material information into the mesh container's pMaterials array. Once again, this is where D3DX will expect the per-subset material information to be if the actor is saved out to disk. Remember, saving the actor only works with managed mode actors since this is the only mode in which the actor is even aware of the textures and materials that are being used to render it.

```
    // Store our mesh in the container
    pMeshContainer->pMesh = pNewMesh;

    // Store the details so that the save functions have access to them.
    if ( pSkinInfo )
    {
        // Here we store the original mesh, because the actual mesh stored
        // in our CTriMesh has been split up into multiple subsets and prepared
        // for skinning. We need the original data so that we can save the correct
        // data back out.
        pMeshContainer->MeshData.pMesh = pOrigMesh;
        pOrigMesh->AddRef();

    } // End if skinning
    else
    {
```

```
        // When not skinning, we are save to save out our optimized mesh data.
        pMeshContainer->MeshData.pMesh = pNewMesh->GetMesh();

    } // End if not skinning

    // Save the remaining details.
    pMeshContainer->MeshData.Type  = D3DXMESHTYPE_MESH;
    pMeshContainer->NumMaterials   = NumMaterials;

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

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

        } // Next Material

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

Finally, we assign the passed mesh container pointer to point at our new mesh container structure so that it will be accessible to D3DX on function return.

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

    // Success!!
    return D3D_OK;

ErrorOut:

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

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

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

At the bottom of the function you see the section of code that is jumped to when an error occurs. It simply cleans up any memory being used when the error occurred before returning failure.

CAllocateHierarchy::BuildSkinnedMesh

A new function that has been added to our CAllocateHierarchy class is the BuildSkinnedMesh method. We learned while covering the previous function that this function is called to convert the regular ID3DXMesh into a new skinned mesh. The function accepts three parameters. The first parameter is a pointer to the mesh container in which the skin information will be stored. Remember, there are many other new members in our mesh container now that hold information about the skin; this function will assign these members the correct values for the type of skin created. The second parameter is a pointer to an ID3DXMesh containing the regular mesh which needs to be converted into a skin. We saw in the last function, that for this parameter we pass in the ID3DXMesh that was loaded from the X file and assigned to the CTriMesh. As the final parameter, we pass the address of an ID3DXMesh interface pointer which will be assigned the address of the new skinned mesh we create. When this function returns the skinned mesh to the calling function, the original mesh is removed from the CTriMesh and the new skinned mesh (pMeshOut) will be attached in its place. The original mesh can then be deleted.

Let us now have a look at this method a section at a time. The function first retrieves the device being used by the actor so that we can retrieve the capabilities of the device to test it for the varying levels of skinning support. Once we have retrieved the capabilities, we release the device interface since we no longer need it.

```
HRESULT CAllocateHierarchy::BuildSkinnedMesh
                                    ( D3DXMESHCONTAINER_DERIVED * pMeshContainer,
                                      LPD3DXMESH pMesh,
                                      LPD3DXMESH * pMeshOut )
{
    D3DCAPS9            Caps;
    HRESULT             hRet;
    ULONG               i, j, k;
    LPD3DXSKININFO      pSkinInfo  = pMeshContainer->pSkinInfo;
    LPDIRECT3DDEVICE9   pDevice    = NULL;

    // Retrieve the Direct3D Device and poll the capabilities
    pDevice = m_pActor->GetDevice();
    pDevice->GetDeviceCaps( &Caps );
    pDevice->Release();
```

In the next line of code we call the other new member function of our CAllocateHierarchy class, called DetectSkinningMethod. This function is a very simple utility that returns one of the following:

SKINMETHOD_INDEXED
SKINMETHOD_NONINDEXED
SKINMETHOD_SOFTWARE

If the current skinning method for the actor has been set to anything other than SKINMETHOD_AUTODETECT, this detection function will simply return the mode that was specified. In other words, the function returns the skin method chosen by the application for the actor. However, if the actor's skinning method has been set to SKINMETHOD_AUTODETECT, this function will query the hardware to find the best skinning method available. This will be either SKINMETHOD_INDEXED or SKINMETHOD_NONINDEXED.

```
    // Calculate the best skinning method and store
    pMeshContainer->SkinMethod = DetectSkinningMethod( pSkinInfo, pMesh );
```

At this point, we now have the skinning method that will be used for this mesh (stored in our mesh container's SkinMethod member). The remainder of the BuildSkinnedMesh function is divided into three sections that create the mesh as either an indexed skin, a non-indexed skin, or a software skin. The first section creates an indexed skinned mesh if that is what has been requested.

The first thing we do in the indexed case is fetch the index buffer of the source mesh so that we can pass it into the ID3DXSkinInfo::GetMaxFaceInfluences function. This function will use the index buffer to determine the maximum number of bones that influence a given triangle in the mesh. We need to know this because the device must have at least this many matrix slots, otherwise the indexed skin will not be able to be transformed with hardware vertex processing. Remember, the ConvertToIndexedBlendedMesh function will subdivide the mesh into smaller subsets if necessary so that each subset can be transformed with the number of matrix slots available, even if it has to break the mesh down into lots of one triangle subsets. However, if there are one or more triangles in this mesh that have more influences than the device has matrix slots, then this mesh can not be transformed by the hardware (not even if it is broken down into one triangle subsets). In such a case, we will have to revert to software vertex processing when rendering this mesh.

```
    // Built differently based on the type of skinning required
    if ( pMeshContainer->SkinMethod == CActor::SKINMETHOD_INDEXED )
    {
        ULONG MaxFaceInfluences;
        ULONG Flags = D3DXMESHOPT_VERTEXCACHE;

        LPDIRECT3DINDEXBUFFER9 pIB;
        hRet = pMesh->GetIndexBuffer(&pIB);
        if (FAILED(hRet)) return hRet;

        // Retrieve the maximum number of influences for the mesh index buffer
        hRet = pSkinInfo->GetMaxFaceInfluences( pIB,
                                                pMesh->GetNumFaces(),
                                                &MaxFaceInfluences);
        pIB->Release();
        if (FAILED(hRet)) return hRet;
```

In the above code, we fetch the maximum number of influences for a given face in the mesh and store that in the MaxFaceInfluences local variable. We then release the index buffer interface we fetched earlier since we no longer need it.

We now know the maximum number of influences a single face uses in the mesh and we know that if this is larger than the number of matrices supported by the device in hardware then this mesh must be rendered using indexed skinning. Actually, that is not quite true! It is possible in some cases that the number of influences for a given face could actually be larger than 12. However, as the pipeline only supports four influences per vertex (i.e., 12 per face), we know that if this is the case, the ConvertToIndexedBlendedMesh function will simply ignore any per-vertex influences over the maximum of four. Even if we have skinning information that tells us a face is influenced by more than 12 matrices, the ConvertToIndexedBlendedMesh function would downgrade the resulting mesh so that

at most there would only ever be 12 influences per face. We are going to compare the MaxFaceInfluences member we just calculated against the number of matrices available on the device to see if the resulting indexed mesh can be rendered in hardware. The problem is, if MaxFaceInfluences is higher than 12, we will fail the hardware case and drop into software mode, even though the resulting mesh will have at most 12 influences per face.

For example, imagine that we have a device that supports 12 matrices and that we have determined that the maximum number of influences per face is 20. We would compare 12 against 20 and find that the maximum face influences is larger than the number of matrices available. In this case we would fail and fall back to software vertex processing. However, this would technically be unnecessary because we know that ConvertToIndexedBlendedMesh will drop those surplus influences as no face can ever have more than 12 influences. Thus, the generated skinned mesh would have a maximum face influence count of 12. We have 12 matrices on our example device matrix palette so this mesh could have been rendered in hardware. Therefore, we must take this into account when performing the comparison against the number of matrices available on the device.

In order to do this we modify the MaxFaceInfluences value so that it can never be larger than 12. We do this by assigning MaxFaceInfluences the minimum between its current value and 12. If maximum face influences is less than 12, its value will be unchanged; if the maximum face influence is greater than 12, it will be assigned 12. As long as the device supports 12 matrices, we will always be able to render the indexed skin in hardware, even if it is broken into one triangle subsets.

```
        MaxFaceInfluences = min( MaxFaceInfluences, 12 );
```

Next we fetch the number of matrices the device supports in hardware. The device capability member that tells us this is the MaxVertexBlendMatrixIndex cap. This is the highest matrix index that the device supports in hardware. Since this is a zero-based number, we know that if this value is 31 for example, the device supports 32 matrices (matrices 0 – 31). To calculate the number of matrices supported by the device, we simply add one to this value.

```
        ULONG MaxSupportedIndices = Caps.MaxVertexBlendMatrixIndex + 1;
```

Next we test to see if the mesh has normals. If it does then it means lighting is being used and the normals must also be transformed. When this is the case, half of the matrix slots will be taken up with matrices to transform the normals. Thus, we actually lose half the number of bones we can transform simultaneously. For example, if the device supports 256 matrices but the vertices have normals, the driver will need to keep reserve half that amount (128) for storing additional copies of the matrix (the inverse transpose of each bone matrix) to transform the normals. While we do not have to set these normal matrices ourselves, we have to be aware that the hardware will need space for them and take this into account. In the next line of code we divide the number of available hardware matrix slots the device can use if normals are present.

```
        if ( pMesh->GetFVF() & D3DFVF_NORMAL) MaxSupportedIndices /= 2;
```

Finally, we test to see if the number of matrices supported by the device (MaxSupportedIndices) is smaller than the maximum number of influences a single face uses. If it is, then the mesh has one or

more triangles that cannot be transformed using hardware vertex processing. In the following code snippet, you can see that when this is the case, we set the mesh container's SoftwareVP member to true so that we know during rendering we must enable software vertex processing before rendering this mesh. We also combine the local Flags member with the D3DXMESH_SYSTEMMEM flag. These flags will be used in a moment to describe the resource pool we would like the skinned mesh to be created in. Since this is going to be a software vertex processing mesh, we want the mesh to be created in system memory so that the CPU can efficiently access the data.

```
        if ( MaxSupportedIndices < MaxFaceInfluences )
        {
            // HW does not support indexed vertex blending. Use SW instead
            pMeshContainer->PaletteEntryCount = min(256,
                                                    pSkinInfo->GetNumBones());
            pMeshContainer->SoftwareVP = true;
            Flags |= D3DXMESH_SYSTEMMEM;
        }
```

In the above code you can also see that we store the number of matrices that will be used for rendering in the mesh container's PaletteEntryCount. Because this is the software vertex processing case, we always have 256 matrix slots on the device to call upon. However, this value will be passed into the ConvertToIndexedBlendedMesh function and will ultimately decide the size of each subset's BoneId array in the returned bone combination table. Certainly we do not want every subset to have a 256 entry BoneId array if we never use more than 20 matrices for a given subset. This is very important, since we will have to loop through and test for valid bone IDs during the rendering of a subset. There is no point in testing every element of a 256 element bone ID array when we only use a handful of bones per subset. So, although we can technically use up to 256 bone matrices, we will never need more matrix slots than the number of bones in the skin. Therefore, the palette size is calculated to be only as large as we need it to be (i.e., either 256 or the number of bones in the skinned mesh, whichever is smaller). When this value is passed to the CovertToIndexedBlendedMesh function, the BoneID array of each subset will contain this many elements. (Of course, each subset may not use all those elements.)

In the following code, we see the 'else' block to the above conditional. It gets executed if the hardware matrix palette is large enough to cater for the the maximum number of matrices that influence a given face.

```
        else
        {
            pMeshContainer->PaletteEntryCount = min( MaxSupportedIndices,
                                                     pSkinInfo->GetNumBones() );
            pMeshContainer->SoftwareVP = false;
            Flags |= D3DXMESH_MANAGED;
        }
```

Do not forget that MaxSupportedIndices is the number of matrix slots available on the device, which may have already been divided in half if normals are being used. We then set the mesh container's SoftwareVP member to false so that we know we can render this mesh in hardware. As this is a hardware vertex processing mesh, we add the D3DXMESH_MANAGED flag to the flags member so that the mesh we create will be allocated in the managed resource pool (video memory).

We have now done all the preparation work for creating the indexed skinned mesh. All that is left to do is create the mesh itself using the ID3DXSkinInfo::ConvertToIndexedBlendedMesh function:

```
        // Convert the mesh
        pSkinInfo->ConvertToIndexedBlendedMesh( pMesh,
                                                Flags,
                                                pMeshContainer->PaletteEntryCount,
                                                pMeshContainer->pAdjacency,
                                                NULL, NULL, NULL,
                                               &pMeshContainer->InfluenceCount,
                                               &pMeshContainer->AttribGroupCount,
                                               &pMeshContainer->pBoneCombination,
                                                pMeshOut );

    } // End if Indexed Skinning
```

The first parameter is the source ID3DXMesh that was passed into the function. This will contain the original geometry that was loaded from the X file and is the mesh we wish to clone into a skinned mesh.

The second parameter contains the mesh creation flags which we have seen many times before. As we just discussed, if we have determined we need to software vertex process this mesh, these flags will specify that the new mesh should be created in the system memory resource pool. If the hardware can transform and render this mesh, the flags will instruct the function to create the new mesh in the managed resource pool.

The third parameter is the palette size we calculated earlier. This tells the function how many matrix slots it can utilize and influences how the function breaks the mesh up into subsets. The larger the palette size, the better chance we have of subsets not being split into smaller subsets based on bone combination. However, as discussed above, this should not be larger than is needed.

The fourth parameter is the adjacency information for the source mesh which was calculated and stored in the mesh container prior to this function being called (we were passed this array by D3DX in the CreateMeshContainer method).

We pass NULL as the next three parameters since we are not interested in receiving the face adjacency information or the face and vertex re-map information of the output mesh (the skin).

As the eighth parameter, we pass the address of our mesh container's InfluenceCount member, which the function will fill with the maximum number of bones that influence a given **vertex**. This tells us how many weights will be in the vertex format of the output mesh and is used during rendering to set the vertex blending render state to allow the pipeline to account for the correct number of weights in each vertex it transforms.

The ninth parameter is the address of the mesh container's AttributeGroupCount member. On function return, it will contain the number of subsets in the new output mesh. This will also describe the number of D3DXBONECOMBINATION structures in the bone combination table.

The tenth parameter is the address of our mesh container's ID3DXBuffer interface pointer (pBoneCombination). On function return, it will contain an array of D3DXBONECOMBINATION structures; one for each subset in the new mesh. Each element in this array will describe the bone and bone offset matrices used by the subset. We must set these bone matrices on the device prior to rendering the subset.

As the final parameter, we pass in the address of an ID3DXMesh interface pointer which, on successful function return, will point to our new skinned mesh.

At this point our indexed skinned mesh has been successfully created and there is nothing else to do in the indexed skinned case.

We will now discuss the section of code that is executed if a non-indexed skinned mesh is to be created. This will be the case if the appliction specifically set the actor to use SKINMETHOD_NONINDEXED or if the auto detection function returned SKINMETHOD_NONINDEXED as the most efficient skinning method supported by the hardware.

In the non-indexed case, creating the mesh is a lot easier. In the following code you can see that we immediately call ID3DXSkinInfo::ConvertToBlendedMesh to create the mesh and we do not have to worry about calculating palette sizes or anything like that. The function is passed exactly the same parameters as in the non-indexed case.

```
    else if ( pMeshContainer->SkinMethod == CActor::SKINMETHOD_NONINDEXED )
    {
       ULONG CurrentIndex = 0;
       // Convert the mesh
       pSkinInfo->ConvertToBlendedMesh(pMesh,
                                       D3DXMESH_MANAGED | D3DXMESHOPT_VERTEXCACHE,
                                       pMeshContainer->pAdjacency,
                                       NULL, NULL, NULL,
                                       &pMeshContainer->InfluenceCount,
                                       &pMeshContainer->AttribGroupCount,
                                       &pMeshContainer->pBoneCombination,
                                        pMeshOut );
```

Once thing to watch out for in the above code is the value returned in the mesh container's InfluenceCount member since it has a slightly different meaning than in the indexed case. In the indexed case, this is the maximum number of bones that influence a given vertex in the mesh. It can be used during rendering to set the D3DRS_VERTEXBLEND render state correctly. In the non-indexed case, this value will be set to contain the maximum number of bones that influence a given **subset**. It is implicitly also the palette size (which is 4 at most). In the non-indexed case, InfluenceCount describes the size of each subset's BoneId array, while in the indexed case, the palette size member contained this value.

At this point we have created our new non-indexed mesh. However, unlike the indexed case, where the mesh is either in hardware or software vertex processing mode, in the non-indexed case it is possible to have a mixture of subsets -- some of which can be rendered in hardware and some which cannot. So our next task will be to find which subsets use more bone influences than the device can handle (4 at most)

and store them as subsets that need to be rendered with software vertex processing enabled. You will recall from our earlier discussions that this will involve stepping though the bone combination structure in two passes. In the first pass we will find all hardware capable subsets and store them in our mesh container's pVPRemap array. We will then take another pass though the subsets searching for any software-only subsets and add them to the pVPRemap array as well.

At the end of this process, the pVPRemap array will contain the indices of all hardware capable subset IDs followed by all the software-only subset IDs. We will have them stored in two batches that can be efficiently rendered without having to perform these tests each time we render. Before we render the batch of software-only subsets, we will have to enable software vertex processing.

So how do we determine which subsets in our new skinned mesh are hardware supported? Well, first we fetch a pointer to the bone combination array that was returned from the ConvertToBlendedMesh function call. We then allocate the subset re-map array to store a number of elements equal to the number of subsets in the new mesh:

```
        LPD3DXBONECOMBINATION pBoneComb = (LPD3DXBONECOMBINATION)
                              pMeshContainer->pBoneCombination->GetBufferPointer();

         // Allocate space for the vertex processing remap table
         pMeshContainer->pVPRemap = new ULONG[ pMeshContainer->AttribGroupCount ];
         if ( !pMeshContainer->pVPRemap )
             { (*pMeshOut)->Release(); return E_OUTOFMEMORY; }
```

Now we have a pointer to the bone combination table (one element for each subset) and an empty array of ULONG's which we are about to populate with the mesh's subset IDs in hardware/software order.

We will need to loop through each subset twice. In the first pass through the outer loop we will be searching for hardware subsets and adding their IDs to our re-map array. In the second iteration of the loop we will be adding software subsets.

```
        // Loop through each of the attributes and determine how many bones
        // influence the vertices in this set, and determine whether it is
        // software or hardware.
         for ( i = 0; i < 2; ++i )
         {
             // First pass is for hardware, second for software
             for ( j = 0; j < pMeshContainer->AttribGroupCount; ++j )
             {
```

Our next task will be to fetch the bone combination structure for the current subset being processed and check each element in its BoneId array to see if it holds a valid index (UNT_MAX == invalid). As discussed earlier, the mesh container's InfluenceCount member now holds the number of elements in each BoneId array, so we will use this to loop through each element. For every valid bone ID we find, we will increment the local loop variable InfluenceCount to keep a running tally for the current subset of how many bones it is influenced by.

```
                ULONG InfluenceCount = 0;
```

```
                // Loop through for each of our maximum 'influence' items
                for ( k = 0; k < pMeshContainer->InfluenceCount; ++k )
                {
                   // If there is a bone used here, increase our max influence
                    if ( pBoneComb[ j ].BoneId[ k ] != UINT_MAX) InfluenceCount++;

                } // Next influence entry
```

At the end of the inner loop shown above, we have calculated the number of bones needed to render the current subset being processed and stored the result in the local InfluenceCount variable. We now test this value against the device capability MaxVertexBlendMatrices, which tells us how many of the possible 1-4 matrices used by the non-indexed skinning technique are supported by the device. If this subset has more influences than the hardware can support then we have found a subset that must be rendered with software vertex processing. Therefore, if we are on the second pass (i == 1) of the outer loop (the pass that searches for software subsets) we add this subset's index to the re-map array. Alternatively, if we have found a subset that can be rendered in hardware and we are on the hardware search pass (i == 0) we add the subset ID to the re-map array.

However, notice that in the first pass (the hardware pass), whenever we add a subset to the re-map array, we also increment the mesh container's SWRemapBegin variable. The idea is that at the end of the first pass, this value will contain the index into the re-map array where the block of software subsets begin. This allows us to easily render the hardware and software subsets in two blocks during the rendering traversal of the hierarchy.

```
                if ( InfluenceCount > Caps.MaxVertexBlendMatrices )
                {
                    // If it exceeds, we only store on the software pass
                    if ( i == 1 ) pMeshContainer->pVPRemap[ CurrentIndex++ ] = j;

                } // End if exceeds HW capabilities
                else
                {
                    // If it is within HW caps, we only store on the first pass
                    if ( i == 0 )
                    {
                        pMeshContainer->pVPRemap[ CurrentIndex++ ] = j;
                        pMeshContainer->SWRemapBegin++;

                    } // End if HW pass

                } // End if within HW capabilities

            } // Next attribute group

            // If all were found to be supported by hardware, no need to test for SW
            if ( pMeshContainer->SWRemapBegin >= pMeshContainer->AttribGroupCount )
                break;

        } // Next Pass
```

Note at the bottom of the outer loop (the pass loop) that we test to see if the SWRemapBegin member of the mesh container is equal to the number of subsets in the mesh. If this test is performed at the end of

the first pass (the hardware pass) and is found to be true, it means that all subsets can be hardware rendered so there is no need to execute the second pass because no software subsets exist. When this is the case, we simply break from the loop instead of continuing on to the second iteration of the loop.

At this point, we have created our non-indexed mesh and determined which subsets need to be hardware or software rendered. Our job is almost done except for one small matter. If we found that some of the subsets needed to be rendered in software, then it means that we need to enable software vertex processing before we render those subsets. But in order for us to render a mesh in the managed resource pool using software vertex processing, the mesh must have been created with the D3DXMESH_SOFTWAREPROCESSING flag. You can be sure that the ConvertToBlendedMesh function did not create our mesh with this flag, so if software subsets are needed, we must clone our skinned mesh so that we can add this new flag. This is what the next section of code does. It clones our skinned mesh into a new skin capable of software vertex processing. We then release the original skin and use the clone instead. After that, our job is done; we have successfully created our non-indexed skinned mesh.

```
        // If there are entries which require software processing, we must clone
        //  our mesh to ensure that software processing is enabled.
        if ( pMeshContainer->SWRemapBegin < pMeshContainer->AttribGroupCount )
        {
            LPD3DXMESH pCloneMesh;

            // Clone the mesh including our flag
            pDevice = m_pActor->GetDevice();
           (*pMeshOut)->CloneMeshFVF( D3DXMESH_SOFTWAREPROCESSING |
                                      (*pMeshOut)->GetOptions(),
                                      (*pMeshOut)->GetFVF(),
                                     pDevice,
                                      &pCloneMesh );
            pDevice->Release();

            // Validate result
            if ( FAILED( hRet ) ) return hRet;

            // Release the old output mesh and store again
            (*pMeshOut)->Release();
            *pMeshOut = pCloneMesh;

        } // End if software elements required

    } // End if non indexed skinning
```

The next and final section of code is executed if a pure software skinned mesh is to be used. The only way this case can ever be executed is if the application specifically set the actor's skinning method to SKINMETHOD_SOFTWARE since the auto detection code will never fall back to this mode. This is because the auto-detection method favors the pipeline skinning techniques using software vertex processing over pure software skinning.

If we are using software skinning, then we already have the mesh created. The source mesh passed into the function (which is currently attached to our CTriMesh) already contains the model space vertices of the skin in its reference pose, and there is nothing else we have to do to it. However, we will need an

additional mesh (a destination mesh) that we can use to transform our geometry into prior to rendering. Therefore, we simply clone the current mesh to create a copy and assign it to the mesh container's pSWMesh member. While this mesh currently contains a copy of the vertex data in the source mesh, the vertex data itself will be overwritten as soon as we call ID3DXSkinInfo::UpdateSkinnedMesh in response to a hierarchy update. However, by cloning it in this way, our destination mesh will have the same index and attribute buffer as the source mesh, which is exactly what we want. After all, it is only the vertex buffer of the destination mesh that will be overwritten each time the mesh is updated; the index and attribute information should be identical for both meshes.

```
    else if ( pMeshContainer->SkinMethod == CActor::SKINMETHOD_SOFTWARE )
    {
        // Clone the mesh that we'll be using for rendering software skinned
        pDevice = m_pActor->GetDevice();

        hRet = pMesh->CloneMeshFVF( D3DXMESH_MANAGED,
                                    pMesh->GetFVF(),
                                    pDevice,
                                    &pMeshContainer->pSWMesh );
        pDevice->Release();

        // Validate result
        if ( FAILED( hRet ) ) return hRet;

        // Add ref the mesh, we're going to pass it straight back out
        pMesh->AddRef();
        *pMeshOut = pMesh;

    } // End if Software Skinning

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

Finally, we now have covered the two largest and arguably most intimidating functions in this lab project. These are the functions that load the mesh, populate the mesh container, and generate a skinned mesh in any of our supported flavors. But we do still have one gap in the coverage thus far -- at the very top of the BuildSkinnedMesh function, a call was made to CAllocateHierarchy::DetectSkinningMethod. This function determines which skinning method the actor should used based on the current skinning method of the actor set by the application and the capabilities of the hardware (if the actor is in the default auto detect mode).

CAllocateHierarchy::DetectSkinningMethod

This method is called at the top of the BuildSkinnedMesh function to determine the skinned mesh creation strategy that should be employed. The first part of the function simply tests to see whether auto-detection of a skinning method is even required. The default skinning method of the actor is SKINMETHOD_AUTODETECT (set in the constructor), which means the function should try to determine the most efficient skinning method for the current hardware. However, the application can change the skinning mode of the actor prior to the load call to specifically state that that it would like to

use indexed, non-indexed, or software skinning. In such cases, this function has no work to do since auto-detection is not required. The function will simply return the current skin mode of the actor as set by the application.

In the first section of this function we retrieve the skinning mode of the actor. If it set to anything other than SKINMETHOD_AUTODETECT, then auto detection is not required and this function can simply return the actor's skinning method back to BuildSkinnedMesh:

```
ULONG CAllocateHierarchy::DetectSkinningMethod( ID3DXSkinInfo * pSkinInfo,
                                                LPD3DXMESH pMesh ) const
{
    HRESULT hRet;
    ULONG   SkinMethod = m_pActor->GetSkinningMethod();

    // Autodetecting?
    if ( !(SkinMethod & CActor::SKINMETHOD_AUTODETECT) )
    {
        // We are not to autodetect, so just return it
        return SkinMethod;

    } // End if no auto-detect
```

If the SKINMETHOD_AUTODETECT method is chosen, then the 'else' block of this conditional will be executed (shown below). The rest of this function simplifies the auto-detection process by first finding out if both indexed skinning and non-indexed skinning are supported in hardware for this mesh. Once we have compiled this information, we can decide which one to use based on any modifier flags that may have been passed to influence the process.

We initially create two local boolean variables called bHWIndexed and bHWNonIndexed which will be initially set to false (meaning none are supported). We will then perform the appropriate tests for each skinning technique and set these variables to true if we find the mesh is supported in hardware by these skinning techniques.

In the first section of the 'else' block we get the capabilities of the current device being used by the actor. Our first test will be to see if hardware indexed skinning is supported for our mesh.

```
    else
    {
        LPDIRECT3DINDEXBUFFER9  pIB;
        D3DCAPS9                Caps;
        LPDIRECT3DDEVICE9       pDevice;
        ULONG                   MaxInfluences;
        bool                    bHWIndexed = false, bHWNonIndexed = false;

        // Retrieve the Direct3D Device and poll the capabilities
        pDevice = m_pActor->GetDevice();
        pDevice->GetDeviceCaps( &Caps );
        pDevice->Release();

        // First of all we will test for HW indexed support
        bHWIndexed = false;
```

We will get the index buffer of the mesh and pass it into the ID3DXSkinInfo::GetMaxFaceInfluences function. This will return (in the MaxInfluences local variable) the maximum number bones influencing a single triangle in the mesh. If this number is greater than the number of hardware matrix slots on the device, hardware indexed skinning is not supported for this mesh. If the maximum face influences is less than or equal to the number of matrix device slots, then we can perform hardware indexed skinning, so we set the bHWIndexed boolean to true.

```
        // Retrieve the mesh index buffer
        hRet = pMesh->GetIndexBuffer(&pIB);
        if ( !FAILED(hRet) && Caps.MaxVertexBlendMatrixIndex > 0  )
        {
            // Retrieve the maximum number of influences for the mesh index buffer
                    pSkinInfo->GetMaxFaceInfluences( pIB,
                                                     pMesh->GetNumFaces(),
                                                     &MaxInfluences );

            pIB->Release();


             if (Caps.MaxVertexBlendMatrixIndex + 1 >= MaxInfluences )
                 bHWIndexed = true;

        } // End if no failure
```

Now we will test to see if this mesh can be rendered in hardware using non-indexed skinning. We use the ID3XSkinInfo::GetVertexInfluences method to retrieve the maximum number of bones that influence a single vertex in the mesh. This will be a maximum of 4 in the non-indexed case. If we have fewer matrices to use than this, then it means there are vertices in this mesh that require more matrices/weights than are supported by the hardware. For example, if the maximum vertex influence is 4, then we need all four matrix slots to be used to transform this mesh in hardware. If we only have two matrix slots on the device, then hardware non-indexed support for this mesh is not available. If this is not the case and we do have enough matrices, we set the Boolean variable bHWNonIndexed to true.

```
        // Now we will test for HW non-indexed support
        bHWNonIndexed = false;

        // It should be safe to assume that if the maximum number of vertex
        // influences is larger than the card capabilities, then at least one
        // of the resulting attribute groups generated by ConvertToBlendedMesh
        // will require SoftwareVP
        hRet = pSkinInfo->GetMaxVertexInfluences( &MaxInfluences );

        // Validate
        if ( !FAILED(hRet) && Caps.MaxVertexBlendMatrices >= MaxInfluences )
              bHWNonIndexed = true;
```

At this point, we now know which skinning techniques are supported in hardware (none, one, or both). All we have to do now is choose the best one. In the next section of code you can see that if hardware indexed support is available and non-indexed hardware support is not available, then our job is easy -- we want to use the indexed skinning method. Even if both were supported in hardware this would be preferable.

```
        // Return the detected mode
        if ( bHWIndexed == true && bHWNonIndexed == false )
        {
            // Hardware indexed is supported in full
            return CActor::SKINMETHOD_INDEXED;

        } // End if only indexed is supported in full in hardware
```

If this is not the case, then we try the opposite test to see if non-indexed is supported in hardware but indexed is not. Once again, this is an easy choice for us since we are always going to favor the hardware option.

```
        else if ( bHWNonIndexed == true && bHWIndexed == false )
        {
            // Hardware non-indexed is supported in full
            return CActor::SKINMETHOD_NONINDEXED;

        } // End if only non-indexed is supported in full in hardware
```

If we get this far, then it might mean that both skinning types are supported in hardware and we have to make a choice. Usually, we will always choose indexed over non-indexed skinning. However we discussed earlier that when we set the actor's skinning method, we can also specify the SKINMETHOD_PREFER_HW_NONINDEXED modifier flag. If this flag is set, it means that the application would like to favor hardware non-indexed skinning over hardware indexed skinning when both are available. So in the next section of code, you can see that if both hardware skinning methods are available, we return SKINMETHOD_INDEXED unless this modifier flag has been set. If it is, we return SKINMETHOD_NONINDEXED instead.

```
        else if ( bHWNonIndexed == true && bHWIndexed == true )
        {
            // What hardware method do we prefer since they are both supported?
            if ( SkinMethod & CActor::SKINMETHOD_PREFER_HW_NONINDEXED )
                return CActor::SKINMETHOD_NONINDEXED;
            else
                return CActor::SKINMETHOD_INDEXED;
        }
```

Finally, if we get this far without returning a skinning method, it means none of the pipeline skinning methods are supported in hardware. Now we will need to choose which one to use with software vertex processing. Again, remember that auto-detection always returns a pipeline skinning technique and never returns pure software skinning mode.

It is difficult to make a blanket statement about which skinning technique will be more efficient when using software vertex processing. Indexed skinning allows much better batch rendering potential, but then again, the multiplication of redundant matrices with zero weights could really hurt performance now that those matrix multiplications are being done on the CPU. Non-indexed skinning can often result in smaller subsets, but it also benefits from being able to set the D3DRS_VERTEXBLEND render state on a per-subset basis during rendering.

For our purposes, we make a design decision that says that the default auto-protection process would choose software non-indexed skinning over software indexed skinning. However, the application can specify the SKINMETHOD_PREFER_SW_INDEXED modifier flag if desired. If this flag is not set, we will return SKINMETHOD_NONINDEXED; otherwise we will return SKINMETHOD_INDEXED instead. Below we see the remainder of the function:

```
        else
        {
            // What software method do we prefer since neither are supported?
            if ( SkinMethod & CActor::SKINMETHOD_PREFER_SW_INDEXED )
                return CActor::SKINMETHOD_INDEXED;
            else
                return CActor::SKINMETHOD_NONINDEXED;

        } // End if both provide only software methods

    } // End if auto detect
}
```

We have now covered the complete hierarchy and mesh creation process for a CActor class which supports skinned meshes. All of the functions we covered in this section are executed when the actor calls the D3DXLoadMeshHierarchyFromX function. So let us now have a look at the CActor::LoadActorFromX function to see if anything has changed. This is the function that is called from the application to load an X file into an actor.

CActor::LoadActorFromX

We know that CActor::LoadActorFromX calls the D3DXLoadMeshHierarchyFromX function to load X files. We also know that the D3DX function will call CAllocateHierarchy::CreateMeshContainer for every mesh found in the file. Therefore, all of the functions we have discussed earlier (CreateMeshContainer, BuildSkinnedMesh, and DetectSkinningMethod) are executed at this time; once for each mesh in the file. When D3DXLoadMeshHierarchyFromX returns program flow back to CActor::LoadActorFromX we might assume then that our work is done and no changes to this function would need to be made. While this is mostly true, some small modifications will be required.

After the hierarchy and its meshes have been loaded, any skinned mesh containers that exist in the hierarchy will not yet have all the information they need. You will recall that our mesh container structure stores an array of bone matrix pointers and an array of corresponding bone offset matrices. We saw in the CreateMeshContainer method how we populated the mesh container's bone offset array without any difficulty -- we extracted the bone offset matrices that were stored in the ID3DXSkinInfo object and copied them into the mesh container. We can do this for bone offset matrices because they are stored inside the file and never change. However, we have not yet copied all the bone matrix pointers from the hierarchy into the mesh container's bone matrix pointer array. Why did we not do this at the same time as we copied over the bone offset matrices? The reason is quite simple…

During hierarchy construction, the frame hierarchy has not been fully loaded yet, so we might not have access to all of the frames used as bones by the skin. For example, imagine a skinned mesh uses five bones and its mesh container is attached to the first (parent) bone. When D3DX is loading the hierarchy, it will do so one frame at a time. It will call CAllocateHierarchy::CreateFrame first to allow our callback to allocate the frame structure. Next it will test to see if a mesh is present, and if so, will call CAllocateHierarchy::CreateMeshContainer. Herein we find our problem; the mesh we are creating uses five frames as bones and yet four of those bones have not even been loaded or processed yet. So how can we possibly copy all the bones into the mesh's bone matrix array when the skeleton has not yet been fully loaded? Of course, we cannot. That is why we must collect the bone matrices for each mesh as a final step *after* the entire hierarchy has loaded and D3DXLoadMeshHierarchyFromX has returned program flow back to us.

The CActor::LoadActorFromX function has had a single function call added to handle this task. It is called CActor::BuildBoneMatrixPointers and it will be invoked right after the loading function returns. We will look at this function in a moment, but its job is very simple. For each mesh container in the hierarchy, it finds the matching bone matrix in the hierarchy for each bone offset matrix (these are linked by bone name) stored in the mesh container. It then copies that bone matrix pointer into the array in the mesh container. Remember, the bone matrices are the absolute matrices of the frames in the hierarchy being used as bones by the skin. At the end of the BuildBoneMatrixPointers function, every mesh container in the hierarchy will contain an array of bone offset matrices and a matching array of bone matrix pointers with a 1:1 correspondence between the two arrays.

Before we cover the CActor::BuildBoneMatrixPointers function, let us have a quick look at modifications to CActor::LoadActorFromX so that you can see where we have added this new function call. The code to this function will not be explained since we have now it many times before. We have highlighted in bold the new code that has been added.

```
HRESULT CActor::LoadActorFromX( LPCTSTR FileName, ULONG Options,
                                LPDIRECT3DDEVICE9 pD3DDevice,
                                bool bApplyCustomSets /* = true */ )
{
    HRESULT hRet;
    CAllocateHierarchy Allocator( this );

    // Release previous data!
    Release();

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

    // Load the mesh heirarchy and the animation data etc.
           D3DXLoadMeshHierarchyFromX( FileName,
                                       Options,
                                       pD3DDevice,
                                       &Allocator,
                                       NULL,
                                       &m_pFrameRoot,
```

```
                                        &m_pAnimController );

    // Build the bone matrix tables for all skinned meshes stored here
    if ( m_pFrameRoot )
    {
        BuildBoneMatrixPointers( m_pFrameRoot );
    }

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

    // Apply our derived animation sets
    if ( m_pAnimController )
    {
        // Apply our default limits if they were set before we loaded
        SetActorLimits( );

        // If there is a callback key function registered, collect the callbacks
        if ( m_CallBack[CALLBACK_CALLBACKKEYS].pFunction != NULL )
            ApplyCallbacks();

        // Apply the custom animation set classes if we are requested to do so.
        if ( bApplyCustomSets == true )
        {
            ApplyCustomSets();
        }
    } // End if any animation data

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

CActor::BuildBoneMatrixPointers

This is a recursive function that is called once by the CActor::LoadActorFromX function, where it is passed a pointer to the root frame. The function will then recurse until it has visited every frame in the hiearchy. It is only interested in finding frames that have mesh containers that store skinned meshes. Once a skin is found, it will collect the bone matrices that it uses and store them in the mesh container's bone pointer array.

The first section of the function tests to see if the frame we are currently processing contains a mesh container that stores a skinned mesh (i.e., the pSkinInfo member will is non-NULL).

```
HRESULT CActor::BuildBoneMatrixPointers( LPD3DXFRAME pFrame )
{
    HRESULT hRet;

    // Has a mesh container?
    if ( pFrame->pMeshContainer )
    {
        D3DXMESHCONTAINER_DERIVED * pContainer =
                            (D3DXMESHCONTAINER_DERIVED*)pFrame->pMeshContainer;
```

```
        LPD3DXSKININFO pSkinInfo  = pContainer->pSkinInfo;

        // if there is a skinmesh, then setup the bone matrices
        if ( pSkinInfo != NULL )
        {
```

At this point we know there is a skin stored at this frame so we need to get the addresses of all bone matrices it needs to use so we can store them in the mesh container's array. This will provide us easy access to them during rendering.

The first thing we must do is allocate the mesh container's bone matrix pointer array so that there are enough elements to hold a matrix pointer for every bone it uses. The ID3DXSkinInfo for this mesh will tell us that information via its GetNumBones member.

```
        ULONG BoneCount = pSkinInfo->GetNumBones(), i;

        // Allocate space for the bone matrix pointers
        pContainer->ppBoneMatrices = new D3DXMATRIX*[BoneCount];
        if ( pContainer->ppBoneMatrices == NULL ) return E_OUTOFMEMORY;
```

We are now ready to start searching for bone matrices. The important point however is that we wish to find and add them to the array in the right order so that we have a perfect pairing between the bone offset matrix array and the bone matrix pointer array. This is not a problem because the ID3DXSkinInfo object contains every bone offset matrix and the name of the bone/frame it belongs to. Since we simply copied over the bone offset matrices into the mesh container (in CreateMeshContainer) in the same order, we can take advantage of this fact. We just have to loop through each bone in the ID3DXSkinInfo object and retrieve its name. We then search for a frame in the hierarchy with the same name and store the address of its absolute matrix in the array.

```
        // Store the bone pointers
        for ( i = 0; i < BoneCount; ++i )
        {
               // Find the matching frame for this bone
               D3DXFRAME_MATRIX * pMtxFrame =
                 (D3DXFRAME_MATRIX*)D3DXFrameFind( m_pFrameRoot,
                                                   pSkinInfo->GetBoneName(i) );
               if ( pFrame == NULL ) return E_FAIL;

               // Store the matrix pointer
               pContainer->ppBoneMatrices[ i ] = &pMtxFrame->mtxCombined;

        } // Next Bone
```

The above code does all the work. We set up a loop to iterate through each bone in the ID3DXSkinInfo and use the D3DXFrameFind function to perform a search (starting from the root) for a frame with the same name as the bone name. This function will return a pointer of that frame and we can copy its absolute matrix address into the bone pointer array. When this loop ends, all bone matrices for this mesh will have their pointers stored in the mesh container.

Usually this would be the end of the road, but we do need to perform one additional test to cater for the pure software skinning case. You will recall that when performing software skinning, we have to manually combine the bone matrices and the bone offset matrices into a temporary array before handing this combined array off to the ID3DXSkinInfo::UpdateSkinnedMesh function. Since all software skins only need to use this buffer temporarily during the update of their skin, we can create one matrix buffer that can be used by all software skinned meshes. Therefore, we have added a new matrix pointer m_pSWMatrices to our CActor class that can be used as a temporary matrix mixing buffer. We have also added a member called m_nMaxSWMatrices which will store the current size of this array. Since this array will only be used for software skins, it will be initially set to NULL and the m_nMaxSWMatrices variable set to zero.

This buffer must be large enough to cater for all the matrices used by the software skin with the maximum number of bones. We can use this same buffer for skins that have fewer bone combinations as this simply means that some of the elements in this array will not be used.

In the next section of code we test to see if the current skin being processed is a software skin. If it is, then we test to see if the current size of this temporary matrix array is large enough to facilitate this mesh. If not, then the temporary matrix array is resized to the number of bones used by the mesh. Obviously, the first time a software skin in encountered, this will always be the case because the array size will be set to zero. However, when processing other mesh containers later, we mind find other software skinned meshes that use even more bones and the array will be resized again. By the time the BuildBoneMatrixPointers function returns back to LoadActorFromX, this temporary matrix buffer will be large enough to handle any software skin in the hierarchy.

```
            // If we are in software skinning mode, we need to allocate our
            // temporary storage. If there is not enough room, grow our temporary
            // array.
            if ( pContainer->SkinMethod == SKINMETHOD_SOFTWARE
                 && m_nMaxSWMatrices < BoneCount )
            {
                // Release previous memory.
                if ( m_pSWMatrices ) delete []m_pSWMatrices;
                m_pSWMatrices = NULL;

                // Allocate new memory
                m pSWMatrices = new D3DXMATRIX[ BoneCount ];
                m_nMaxSWMatrices = BoneCount;

                // Success ?
                if ( !m_pSWMatrices ) return E_OUTOFMEMORY;

            } // End if grow SW storage

        } // End if skinned mesh

    } // End if has mesh container
```

At this point we have fully populated the mesh container and there is nothing left to do at this frame. So, using the recursive behavior we have seen so many times before, we continue to walk the hierarchy, first visiting the sibling list and finally visiting the child list.

```
    // Has a sibling frame?
    if (pFrame->pFrameSibling != NULL)
    {
        hRet = BuildBoneMatrixPointers( pFrame->pFrameSibling );
        if ( FAILED(hRet) ) return hRet;

    } // End if has sibling

    // Has a child frame?
    if (pFrame->pFrameFirstChild != NULL)
    {
        hRet = BuildBoneMatrixPointers( pFrame->pFrameFirstChild );
        if ( FAILED(hRet) ) return hRet;

    } // End if has child

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

At the end of this function, the entire hierarchy is complete. Our actor's mesh containers will have all the information they need in order to be transformed and rendered. The only new part of CActor that is left for us to look at is the rendering code.

CActor::DrawMeshContainer

Although our actor is now much more complex than it used to be, nothing has changed from the perspective of the application regarding how the actor is rendered. As in all of our previous lab projects that have used CActor, the scene is managing the textures and materials, so rendering each actor consists solely of looping through each subset and calling the DrawSubset method with the global ID for the subset we wish to draw.

You will hopefully recall that CActor::DrawSubset is a simple wrapper function around the first call to the recursive function CActor::DrawFrame. It passes in the root frame and the subset ID that was requested to be drawn by the application. The CActor::DrawFrame function then recursively walks the hierarchy looking for mesh containers. Once a mesh container is found, the CActor::DrawFrame function calls the CActor::DrawMeshContainer function, passing in the mesh container and the subset ID that needs to be rendered. It is in the CActor::DrawMeshContainer function that all the rendering work is done. This function has changed significantly from previous lab projects since it has been expanded to render both regular meshes and skins (indexed, non-indexed skins, and pure software). This function now also manages the setting of the bone matrices on the device (pipeline skinning) before the requested subset is rendered. In the software skinning case, the function updates the destination mesh in software if the hierarchy has been updated since the last frame.

DrawMeshContainer is divided up into four sections. Each section takes case of rendering a different mesh type. The first section handles the rendering of the mesh subset if it is an indexed skin. The second section contains the rendering logic for a non-indexed skin. The third section is renders a pure software skin, and the final section handles regular meshes. This is now a pretty large function (although not too complex), so we will cover it a piece at a time.

We will begin with the indexed skinning case. You can see below that we first test to see if the mesh container we are about to render contains a skinned mesh or not. If the mesh container's pSkinInfo member is set to NULL then we will skip the entire skinned code block and proceed to the bottom of the function where the regular mesh will be rendered normally. If a skin does exist, we first test to see if the skinning method is indexed.

```
void CActor::DrawMeshContainer(LPD3DXMESHCONTAINER pMeshContainer,
                               LPD3DXFRAME pFrame,
                               long AttributeID /* = -1 */ )
{
    ULONG                        i, j, MatrixIndex;
    D3DXFRAME_MATRIX           * pMtxFrame  = (D3DXFRAME_MATRIX*)pFrame;
    D3DXMESHCONTAINER_DERIVED * pContainer(D3DXMESHCONTAINER_DERIVED*)pMeshContainer;
    CTriMesh                   * pMesh      = pContainer->pMesh;
    LPD3DXSKININFO               pSkinInfo  = pContainer->pSkinInfo;
    D3DXMATRIX                   mtxEntry;

    // Is this a skinned mesh ?
    if ( pSkinInfo != NULL)
    {
        if ( pContainer->SkinMethod  == SKINMETHOD_INDEXED )
        {
            // We have to render in sofware
            if ( pContainer->SoftwareVP )m_pD3DDevice->SetSoftwareVertexProcessing( TRUE );

            // Set the number of blending indices to be used
            if ( pContainer->InfluenceCount == 1 )
             m_pD3DDevice->SetRenderState( D3DRS_VERTEXBLEND,D3DVBF_0WEIGHTS );
            else
             m_pD3DDevice->SetRenderState(D3DRS_VERTEXBLEND,pContainer->InfluenceCount-1);


            // Enable indexed blending
           if( pContainer->InfluenceCount )
              m_pD3DDevice->SetRenderState( D3DRS_INDEXEDVERTEXBLENDENABLE, TRUE );
```

The first thing we do in the indexed skinning case is test to see if the mesh container's SoftwareVP boolean is set to true. If it is, then it means we determined during the loading phase that this mesh has one or more triangles that are influenced by more bones than the device has matrix slots in its palette. If so, this mesh must be rendered using software vertex processing and we call the appropriate API call to make that so.

Our next task in the above code is to correctly switch on vertex blending in the pipeline using the D3DRS_VERTEXBLEND render state. We must set it for the value that describes the vertex format of the mesh. This assures that the pipeline knows the location of the indices within each vertex structure. The mesh container also stores the influence count which tells us the maximum number of bone influences per-vertex. This also tells us (if you subtract 1) the number of weights contained in the vertex structure. Remember that the last weight is calculated on the fly by the pipeline, so a three weight vertex structure would actually use four matrices. So, you can see that we first test to see if the influence count is equal to 1. If so, then every vertex in this mesh is influenced by only one bone. When this is the case, no actual vertex blending occurs since vertex blending assumes the influence of multiple matrices on a single vertex. Thus, the vertex structure will have no weights defined at all because the pipeline will assume a weight of 1.0 for the only matrix used for transformation.

However, if the influence count is not 1, then we know that by subtracting one from this amount, we have the actual number of weights stored in the vertices. This is exactly the value we should set the vertex blend render state to. It is very important that we do this correctly in the indexed skinning case because if we pass in a vertex blend renderstate of *n*, the pipeline will expect the indices to be stored in weight *n* + 1. If we set this value incorrectly, we are giving bad information to the pipeline about the format of our vertices, and it may try to fetch the matrix palette indices from the wrong weight causing all manner of incorrect transformations.

Finally, the last thing we do in the above code is test to see if the maximum vertex influence count is greater than 1. If so, we enable indexed vertex blending. This is another important point to remember. It is possible to render an indexed skinned mesh without using any form of vertex blending. This is typically the case if every vertex in the mesh is only influenced by a single bone. For example, we could have a skin that collectively uses a palette of 200 bones but no one vertex ever uses any more than one of those bones for its transformation process. In this case, while we are still rendering a skin using indices into the matrix palette for the mesh as a whole, no vertex blending needs to be performed for its vertex transformations. After all, if a vertex is only to be transformed by a single matrix there is nothing to blend. In such a case we simply transform the vertex by the single bone matrix. This is exactly why such a mesh would also have no need for weights in its vertices. So, we only enable indexed vertex blending when necessary.

Now that we have prepared the device to render our skin, our next task is to loop through each subset and render it. The information for each subset is stored in the mesh container's bone combination table, so we fetch a pointer to that first. We then loop through each subset/bone combination structure and process it. Take a look at the following code that renders each subset and we will talk about it afterwards:

```
            // Get a usable pointer to the combination table
            LPD3DXBONECOMBINATION pBoneComb = (LPD3DXBONECOMBINATION)
                                    pContainer->pBoneCombination->GetBufferPointer();

            // Loop through each of the attribute groups and calculate the
            // matrices we need.
            for ( i = 0; i < pContainer->AttribGroupCount; ++i )
            {
             if ( AttributeID >= 0 && pBoneComb[i].AttribId !=(ULONG)AttributeID )
                    continue;

                // First calculate the world matrices
                for ( j = 0; j < pContainer->PaletteEntryCount; ++j )
                {
                  // Get the index
                  MatrixIndex = pBoneComb[i].BoneId[j];

                  // If it's valid, set this index entry
                  if ( MatrixIndex != UINT_MAX )
                  {
                      // Generate the final matrix
                      D3DXMatrixMultiply( &mtxEntry,
                                          &pContainer->pBoneOffset[ MatrixIndex ],
                                           pContainer->ppBoneMatrices[ MatrixIndex ] );

                        // Set it to the device
                        m_pD3DDevice->SetTransform( D3DTS_WORLDMATRIX( j ), &mtxEntry );
```

```
                } // End if valid matrix entry

            } // Next Palette Entry

            // Draw the mesh subset
            pMesh->DrawSubset( i, pBoneComb[i].AttribId );

        } // Next attribute group
```

The first thing we do when we enter the subset loop is test to see if this subset is the subset that we actually want to render. Although we are looping through each subset, if the caller has requested that only subset *n* is to be rendered, we will skip any others. If no subset ID has been passed then this usually means that we are using managed meshes. When this is the case we wish to render every subset in the mesh since the CTriMesh will take care of setting the texture and material on the device before it renders the triangles. You can see in the above code that if the passed attribute ID is less than zero, no subsets wil be skipped and all will be rendered. This is the value that gets passed to the DrawMeshContainer function when managed meshes are being rendered.

Once we have decided that we wish to process a subset, our next task is to loop through each element in the BoneID array of the subset (which will have PaletteEntryCount elements). We do this to fetch the indices of the bones that influence the subset which need to be sent to the device prior to rendering. If any of the BoneId elements for a given subset are set to UINT_MAX then this subset does not use that matrix slot and the element can be ignored. The index of the BoneId element in the array describes the matrix slot that the bone index it contains should be set to. For example, if BoneId[5]=10, it means bone 10 in the mesh container's bone matrix array should be assigned to matrix slot 5 on the device. You can see that we use the bone index stored in each BoneId element to fetch the correct bone matrix and its corresponding bone offset matrix from the mesh container's array. We then multiply them together to create a single combined matrix which is set in its appropriate position in the matrix palette.

After we have looped though every element in the BoneId array for the current subset we are processing, we will have assigned all the bone matrices to their correct places in the device's matric palette and we are ready to render the mesh subset. We do this with a call to CTriMesh::DrawSubset. Notice that we pass in two parameters. The first parameter is the local subset of the mesh we wish to render. (Remember, we never re-map subset IDs when using skins.) The second parameter, which is used only in managed mode, describes the index for the texture and material used by this subset in the mesh's internal attribute table. Thus, the mesh can correctly set the texture and material on the device before rendering the subset. Obviously, if the mesh is in non-managed mode, this second parameter is unnecessary since the mesh assumes that the application has set the texture and material before making the draw call. If this is confusing, just remember that earlier in the lesson, we discussed how the AttribId member of a bone combination structure contains the original subset ID that this new subset was created from when the mesh was converted to a skin. This describes the index into the mesh's attribute data array for the texture and material used by that subset.

At this point, we have rendered our indexed skinned mesh (or a requested subset) and our job is done. Before we exit however, we should reset those device states we changed and set them back to their defaults. As you can see, in the closing statements of the indexed skinning case, we disable indexed vertex blending, set the vertex blending render state back to zero and return software vertex processing back to its default state.

```
            // Reset the blending states
            m_pD3DDevice->SetRenderState( D3DRS_INDEXEDVERTEXBLENDENABLE, FALSE );
            m_pD3DDevice->SetRenderState( D3DRS_VERTEXBLEND, 0 );

            // Reset back to hardware vertex processing if required
            if ( pContainer->SoftwareVP )
                 m_pD3DDevice->SetSoftwareVertexProcessing( FALSE );

        } // End if indexed skinning
```

In the next section we will look at the code that handles the non-indexed case. This code is very similar except for a few minor changes here and there. As before, the code starts by getting the mesh's bone combination table so that it has access to the bone and attribute information for each subset. We then loop through each subset to process and eventually render it as we did before. Once again, in the following code you can see that if a specific subset ID was passed in and the current subset we are processing does not match, we skip it. This ensures only the requested subset is rendered for non-managed mode meshes.

```
        else if ( pContainer->SkinMethod == SKINMETHOD_NONINDEXED )
        {
            // Get a usable pointer to the combination table
            LPD3DXBONECOMBINATION pBoneComb =
              (LPD3DXBONECOMBINATION)pContainer->pBoneCombination->GetBufferPointer();
            // process each subset
            for ( i = 0; i < pContainer->AttribGroupCount; ++i )
            {
                // Retrieve the remapped index
                ULONG iAttribID = pContainer->pVPRemap[ i ];

                // Enable software processing if we reached our SW begin index
                if ( i == pContainer->SWRemapBegin )
                    m_pD3DDevice->SetSoftwareVertexProcessing( TRUE );

                if ( AttributeID >= 0 &&
                     pBoneComb[ iAttribID ].AttribId != (ULONG)AttributeID ) continue;
```

At the center of the subset loop there is a fundamental change. With indexed skinning, the mesh is either in software or hardware mode, but with non-indexed skinning there may be a mixture of both types. You will recall that to speed up the switching process, we created a subset re-map array so that the subsets are stored and processed in order of hardware capability (hardware capable subset IDs followed by software subset IDs). So we will process the subsets not in the usual 0 to *n* order, but in the order they are stored in this re-map array. The code above does just that.

We fetch the bone combination table and then set up a loop to iterate through the number of subsets. At the start of each iteration we fetch the next subset we wish to process from the re-map array. We then test the current loop variable 'i' to see if it is equal to our mesh container's SWRemapBegin. We calculated this value earlier as the position in the re-map array where hardware subsets end and software subsets begin. If we hit that index, then it is time to enable software vertex processing for the rest of the subsets we process. Finally, we test the subset ID against the subset ID passed into the function (AttribID). If this is not the subset our application specifically wished to render then we skip to the next iteration of the loop.

Now we know which subset we are processing and we also know which bone combination structure to use, so we can fetch the bone matrix indices it uses. We set up a loop to loop through each element in the BoneID array. This will be a maximum of four in the non-indexed case, but may be less if the hardware does not support all four matrix slots. This value will be stored in our mesh container's InfluenceCount member as in the indexed case. In the non-indexed case, this value actually contains the total number of bones that influence a given subset, but is pretty much the same thing as far as determining the size of the palette being used is concerned.

As discussed earlier, in the non-indexed case we do not have matrix indices in the vertices to worry about, so we can set the D3DRS_VERTEXBLEND render state on a per-subset level so that we do not unnecessarily blend using matrices that have zero weights defined in the vertex. However, in order to perform this optimization, we must know how many matrices this subset uses (among all its vertices). In the following loop, we simply test each BoneId element for this subset to see if a valid BoneId is stored there. Each time we find one, we set the local variable MaxBoneIndex equal to loop variable 'j'. At the end of this loop, MaxBoneIndex will contain a value between 0 and 3 describing the highest matrix index that influences this subset. If this was 2 for example, then we know that we can set the D3DRS_VERTEXBLEND render state to process only 1 weight (the second weight is calculated by the pipeline). This would save us having to incorporate matrices 2 and 3 into the calculation unnecessarily as their contribution would have zero weight anyway. Remember, without doing this step, if one vertex in the mesh was influenced by four bones, but all the rest were influenced by only one, we would multi-matrix transform every vertex in the mesh using four matrices, even if those matrices contribute nothing.

```
ULONG MaxBoneIndex = UINT_MAX;

// Count the number of bones required for this attribute group
for ( j = 0; j < pContainer->InfluenceCount; ++j )
{
    // If this is not an 'empty' bone, increase our max bone index
    if ( pBoneComb[ iAttribID ].BoneId[ j ] != UINT_MAX ) MaxBoneIndex = j;

} // Next Influence Item
```

We now know how many matrix slots this subset uses. So let us take another pass through the BoneId array and actually build the matrices this time. For each BoneId element in the array, we fetch the Bone index it describes. We then use this index to fetch the bone matrix and the bone offset matrix for the mesh container's arrays and bind them to the device in the correct matrix slot.

```
// First calculate the world matrices
for ( j = 0; j <= MaxBoneIndex; ++j )
{
    // Get the index
    MatrixIndex = pBoneComb[ iAttribID ].BoneId[j];

    // If it's valid, set this index entry
    if ( MatrixIndex != UINT_MAX )
    {
        // Generate the final matrix
        D3DXMatrixMultiply( &mtxEntry,
                            &pContainer->pBoneOffset[ MatrixIndex ],
                            pContainer->ppBoneMatrices[ MatrixIndex ] );

        // Set it to the device
```

```
                    m_pD3DDevice->SetTransform( D3DTS_WORLDMATRIX( j ), &mtxEntry );

                } // End if valid matrix entry

            } // Next Palette Entry
```

At this point the device has been assigned all the matrices needed to transform and render this subset. We will now set the vertex blend render state to process MaxBoneIndex weights and finally draw the actual subset using CTriMesh::DrawSubset.

```
            // Set the blending count to however many are required / in use
            m_pD3DDevice->SetRenderState( D3DRS_VERTEXBLEND, MaxBoneIndex );

            // Draw the mesh subset
            pMesh->DrawSubset( iAttribID, pBoneComb[ iAttribID ].AttribId );

        } // Next attribute group
```

With every subset of the mesh now processed and rendered, all that is left to do before exiting the non-indexed skinning case is reset the render states we may have changed to their default settings.

```
        // Reset the blending states
        m_pD3DDevice->SetRenderState( D3DRS_VERTEXBLEND, 0 );

        // Disable software processing if it was enabled
        if ( pContainer->SWRemapBegin < pContainer->AttribGroupCount )
             m_pD3DDevice->SetSoftwareVertexProcessing( FALSE );

    } // End if Non-Indexed
```

We have now covered the rendering section that deals with both pipeline skinning methods. The final skinning method we have to deal with is pure software skinning. In many ways, software skinning is much simpler since we do not have the bone combination relationship to deal with and we do not have to worry about tweaking the vertex blend and software vertex processing render states.

In software skinning the basic process is:

1) Combine every bone matrix with every bone offset matrix used by the mesh and store in our temporary array.

2) Lock the vertex buffer's of the model space source mesh and the destination mesh.

3) Pass the combined matrices and both vertex buffers into the ID3DXSkinInfo::UpdateSkinnedMesh function. On function return, the destination mesh will contain world space geometry that has been calculated using multi-matrix transformations.

4) Set the device world matrix to an identity matrix (destination mesh already in world space)

5) Render the mesh

The basic process is not new to us but is listed above for your review. There are one or two things in the code that we have added for efficiency and convenience which will be discussed before we look at the code.

First, we really do not want to regenerate the skinned mesh (as described above) every time we wish to render it. If none of its bones have changed position since the last build then this is unnecessary since the world space geometry in the destination mesh will still be current. As mentioned earlier, we have added a boolean variable called Invalidated to our mesh container structure. The mesh only gets rebuilt if this value is set to true; otherwise we simply render the current destination mesh. The Invalidated boolean is set to true (for every mesh container) when the hierarchy is updated (e.g., after animation has been applied) inside the CActor::UpdateFrameMatrices function. If it is set to true when it comes time to render this mesh, we will rebuild the world space destination mesh (prior to rendering it) and then reset this Boolean back to false after this has been done. This way, we only rebuild the destination mesh when a change occurs to the hierarchy.

Below you will see the code to all the steps outlined above. It fetches every bone matrix and bone offset matrix stored inside the mesh container and combines each pair into a resulting temporary matrix buffer (m_bSoftwareMatrices). It then locks the vertex buffers of the source and destination mesh prior to passing them into the ID3DXSkinInfo::UpdateSkinnedMesh method along with the combined matrix buffer. This is the code for the software skinning case:

```
        else if ( pContainer->SkinMethod == SKINMETHOD_SOFTWARE )
        {
            D3DXMATRIX  Identity;
            ULONG       BoneCount = pSkinInfo->GetNumBones();
            PBYTE       pbVerticesSrc;
            PBYTE       pbVerticesDest;

            // Get the actual mesh
            LPD3DXBASEMESH pSrcMesh = pMesh->GetMesh();
            if ( !pSrcMesh ) pMesh->GetPMesh();

            // If the data has been invalidated since last rendering, update our SW mesh
            if ( pContainer->Invalidated )
            {
                // Loop through and setup the bone matrices
                for ( i = 0; i < BoneCount; ++i )
                {
                    // Concatenate matrices into our temporary storage
                    D3DXMatrixMultiply( &m_pSWMatrices[i],
                                        &pContainer->pBoneOffset[i],
                                        pContainer->ppBoneMatrices[i] );

                } // Next Bone

                // Lock the input and output vertex buffers
                pSrcMesh->LockVertexBuffer(D3DLOCK_READONLY, (LPVOID*)&pbVerticesSrc);
                pContainer->pSWMesh->LockVertexBuffer( 0, (LPVOID*)&pbVerticesDest);

                // Generate the newly skinned mesh data
                pSkinInfo->UpdateSkinnedMesh( m_pSWMatrices,
                                              NULL,
                                               pbVerticesSrc,
                                               pbVerticesDest );
```

```
                // Unlock the buffers
                pSrcMesh->UnlockVertexBuffer();
                pContainer->pSWMesh->UnlockVertexBuffer();

            } // End if update required
```

At this point, our destination ID3DXMesh (pSWMesh) contains the new position of our software skin's vertices in world space. Therefore, we must make sure when we render it that the world matrix is set to an identity matrix since we do not wish an additional world transform to be applied.

```
            // Already in world space so set world to identity
            D3DXMatrixIdentity( &Identity );
            m_pD3DDevice->SetTransform( D3DTS_WORLD, &Identity );
```

The next section of code requires some explanation. The world space vertices are currently stored in pSWMesh. This is just an ID3DXMesh and, as such, has no attribute data linked to it. We cannot use the CTriMesh interface to render this mesh because it is not attached to a CTriMesh. The CTriMesh stored in our mesh container is currently attached to the source mesh (pSrcMesh) which contains the model space skin data. Therefore, we will temporarily attach the destination mesh to the CTriMesh instead, so that we can use its draw functions and borrow is texture and material data (for managed mode meshes).

```
            // Attach the SW mesh to our CTriMesh (attach only) for rendering
            pMesh->Attach( pContainer->pSWMesh, NULL, true );
```

We are now ready to render the CTriMesh since it currently contains the world space mesh geometry. If the application specified a subset to be rendered, then we call CTriMesh::DrawSubset. If not, we assume this is a managed mode skin and call CTriMesh::Draw. We know from previous lab projects that this function simply calls DrawSubset for each subset in the mesh.

```
            // Render the mesh or subset
            if ( AttributeID >= 0 )
            {
                // Draw the subset
                pMesh->DrawSubset( (ULONG)AttributeID );

            } // End if attribute specified
            else
            {
                // Draw the mesh as a whole
                pMesh->Draw( );

            } // End if no attribute specified
```

With the mesh now rendered, we re-attach the CTriMesh's original ID3DXMesh which contains the geometry in its model space reference pose. We do this to be consistent for the application. In all other skinning techniques and in the case of regular meshes, the application could access the CTriMesh stored at each mesh container and would expect it to be in model space. By switching it back again, we make sure that this assumption is correct when software skinning is being used. After all, we never know what somebody will want to do with our actor. For example, they may be traversing our hierarchy in order to build model space bounding volumes for each of the meshes.

```
            // Re-attach our source mesh
            pMesh->Attach( pSrcMesh, NULL, true );

            // Now we can release our source mesh
            pSrcMesh->Release();

        } // End if software skinned
    } // End if skinned
```

That ends the software skinning conditional code block, and consequently, the entire skinned mesh rendering code block. The only conditional code block left to cover is executed if the mesh stored here is not a skin, but a standard mesh. When this is the case, we render the mesh as we always have.

```
    else
    {
        // Render the mesh
        if ( AttributeID >= 0 )
        {
            // Draw the subset
            pMesh->DrawSubset( (ULONG)AttributeID );

        } // End if attribute specified
        else
        {
            pMesh->Draw( );
        } // End if no attribute specified
    } // End if not skinned
}
```

While that was quite a big rendering, each rendering method taken individually is fairly small.

Conclusion

We have now covered all of the modifications to CActor that provide automatic support for the loading, transformation, and rendering of skinned meshes. There is no need to look at modified code in the CScene or CGameApp class because, there is none. We have managed to expand our actor functionality without the application needing to know about the changes. Thus, we have accomplished all of our design goals that we stated early in the lesson.

In the next lesson we will continue to explore what can be achieved with both animation and skinned meshes. In the workbook, we will add a data-driven animation layer to CActor, allowing us to select which combinations of animation sets we wish to play on the fly in response to use input or other game events. In our textbook, we will also gain a much deeper understanding of how the data for a skinned mesh is laid out when we create a new class derived from CActor that will be used to procedurally generate skinned tree meshes. The examination of the CTreeActor code will demonstrate how we can build skinned meshes and skeletal hierarchies procedurally as well as how to generate procedural animation for the skeletal structure. This should be a nice way to round off our current studies of skinned meshes and skeletal hierarchies. The combination of all of these concepts in our workbook will make for a good way to mark the halfway point in this course.