

Chapter Eleven

Skinned Meshes and Skeletal Animation I



Introduction

Newly emerging graphics technologies have allowed gamers to enjoy scene realism that would be unthinkable only a few years ago. With new cards now supporting some 256MB or more of on-board local video memory and features like real-time decompression of textures during rendering, we now have the ability to use greater quantities of higher resolution textures. We have also seen a dramatic increase in polygon budgets due to similar advances in vertex processing hardware. The combined effect is that the game artist experiences a larger degree of creative freedom coupled with the confidence that the game engine can handle the loads in real time. As a result, the virtual worlds we take for granted in modern computer games can often look so real as to be almost photo-realistic. But in the area of animation, and specifically with respect to game characters, until fairly recently the technology has not kept pace. Traditionally, as soon as you saw a character animating in the scene it was obvious that this was not a photo or a movie you were looking at.

In early games, comparatively low CPU speeds and the lack of dedicated 3D transformation and lighting hardware meant that per-vertex processing had to be kept to a minimum in order to maintain an acceptable frame rate. In-game characters were often constructed as segmented polygonal objects, where each limb of the character would be created as a separate mesh and attached to a frame hierarchy. The frames of the hierarchy represent the ‘bones’ of the character. Animations like walking or shooting could be applied to the hierarchy (as we examined in Chapter Ten), causing the meshes of the separate body parts attached to the individual frames to move. While this approach achieved its objective, it brought with it some undesirable artifacts. Because each body part was a separate mesh, when two neighboring body parts (such as an upper leg mesh and a lower leg mesh) were rotated at extreme angles with respect to each other, gaps between them became very obvious. Unless extreme care was taken on the modeling side to reduce this artifact – and it could not always be successfully done – the results were less than ideal.

As CPU speeds increased, a greater degree of per-vertex processing could be done in software and a technique called *vertex blending* emerged. Vertex blending is the process of transforming a model space vertex into world space using a weighted blending of multiple world space matrices. This is in contrast to the single world matrix concept we are familiar with. This technique allows programmers to use a single mesh for an entire character while retaining all of the benefits of a hierarchy for animation (much like the segmented character example mentioned previously). The single mesh behaves like a “skin” draped over the “bones” of the hierarchy such that when the bones (i.e. the frame hierarchy) are animated, rather than just rotating limbs constructed from separate meshes, the entire skin (i.e. mesh) is deformed -- bending and stretching to maintain the shape described by the underlying skeleton. Because the skin is a single mesh, no cracks will appear and the limbs of the character smoothly bend and stretch to provide much more realistic animation.

To make this skin/bone concept clearer, first consider a segmented object, where each mesh is assigned to a single frame in a multi-frame hierarchy. Forget about the fact that each mesh has its own vertices and forget about which meshes these vertices actually belong to. Based simply on the frame the parent mesh was originally assigned to, you can imagine that some vertices would be animated by one frame matrix, while other vertices belonging to another mesh assigned to another frame would be animated by a different frame matrix. If we were to go one step further and imagine that all of the vertices in the

hierarchy were actually a part of a single mesh and we add the ability to specify, on a *per-vertex* level, which matrix in the hierarchy should be used to transform each vertex, then we wind up with our skin concept. That is, we have a single mesh (a skin) animated by the frame hierarchy (a skeleton) such that each vertex in the mesh stores information about which matrix in the hierarchy transforms it to world space. If we were to play a walk animation on our hierarchy, then we would see that when the vertices that are assigned to the leg frames (bones) are animated, the legs of the character mesh can bend without causing cracks or visible seams.

Although assigning a single vertex in the mesh to a single frame in the hierarchy would cause the skin to deform, often this is too restrictive for producing realistic skin animation. This is where vertex blending comes in. To see where vertex blending can play a role, look at the skin around your elbow and imagine it as a mesh of vertices. If you bend your arm at the elbow, retracting the forearm up to meet your shoulder (imagine that your forearm is a bone in the hierarchy), you can clearly see that the skin at your elbow is affected by this bone movement. The skin changes to cover the new shape formed at the elbow joint. Thus we might say that the vertices of your elbow should be transformed by the forearm frame matrix in the hierarchy. That is, whenever the forearm frame is rotated, the elbow vertices that are children of that frame would be directly manipulated by that frame's matrix.

However, if you stretch out your arm in front of you and just rotate your wrist up and down, you can see that while not as severe as the previous example, the rotation of the wrist bone also has some influence on the skin near the elbow -- causing it to shift and change slightly. We could say then that the elbow vertices are directly affected by the rotation of the forearm bone and also by the rotation of the wrist bone (and theoretically other bones as well, such as the shoulder bone). Therefore, to correctly model the behavior of our skin, we would need to be able to specify not only one matrix per vertex, but in fact any number of matrices per vertex. While some vertices in the mesh may only be affected by a single bone, others may be affected by two or more different bones to varying degrees, sometimes simultaneously. Although the orientation of your wrist bone did affect the position of the skin around your elbow, it did not influence the skin position to the same degree that rotating your forearm did. Therefore, to effectively skin a mesh:

1. We need a **skeleton**. This is just a frame hierarchy where each frame is assumed to be a bone in the skeleton. Each frame matrix is referred to as a bone matrix.
2. We need a **skin**. This is just a standard mesh describing the character model in its reference (default) pose. The vertices are defined in model space just like any other standard mesh.
3. We need a **vertex/bone relationship mapping**. We want to know for each vertex, which bones (frames) in the skeleton (hierarchy) directly affect it. The bone(s) will ultimately be used to calculate the world space position of the vertex. As described, a single vertex may be directly connected to several bones, such that movement of any of those bones will influence its final world space position to some degree.
4. We need **weights** for each vertex describing how influential a particular bone will be in calculating its final world space position. If a vertex is influenced by N bones, then it should also contain N floating point weight values between 0.0 and 1.0. These describe the influence of that bone on the vertex as a percentage. For example, if a vertex stores a floating point weight value of 0.25 for matrix N, it means that the transformation described by bone N (the frame matrix) should be scaled to 25% of its strength and then applied to the vertex. So if a vertex is influenced by N matrices, then it should store N weights. The sum of the N weight values should add up to exactly 1.0.

So now we need the ability to assign individual vertices, rather than entire meshes, to frames in our hierarchy. We also need the ability to perform vertex blending. Item four in the previous list is where vertex blending comes in. We know that the first thing the DirectX transformation pipeline will typically do is transform vertices into world space using the currently set world matrix (a single matrix). Things are a little different now however because, for any skin vertex currently being transformed, a vertex might need to be transformed into world space using contributions from a number of bone matrices. This is how vertex blending works. If we were going to implement vertex blending ourselves (not using APIs like DirectX or OpenGL), then our code would have to include a vertex blending function. That function would be called for each model space source vertex and generate a world space destination vertex that had been transformed using the contributions of several matrices. Vertex blending is actually a remarkably easy thing to do as we will briefly describe below.

Note: The DirectX pipeline will perform vertex blending on our behalf, but an understanding of the blending process is essential to understanding skinning. This next section is intended to show you how vertex blending works and what is going on behind the scenes.

Imagine that we wish to perform vertex blending with three matrices for a given model space vertex. In this example we can think of the model space vertex as being a vertex in the mesh which is assigned to three bones in the skeleton. Remember that bones are ultimately just frame matrices and the skeleton is just a frame hierarchy.

First we take the model space vertex and, in three separate passes, transform it by each world space bone matrix to create three temporary vertices:

ModelVertex = Model Space vertex we wish to transform into world space

M1 = 1st World Space matrix to use in blending

M2 = 2nd World Space matrix to use in blending

M3 = 3rd World Space matrix to use in blending

TempVertex1 = ModelVertex * M1;

TempVertex2 = ModelVertex * M2;

TempVertex3 = ModelVertex * M3;

At this point we are midway through the process. When performing vertex blending, each vertex also stores some number of weights (sometimes referred to as *beta weights* or *skin weights*) with values between 0.0 and 1.0. These weights describe how influential a specific matrix will be when determining the final position of the world space vertex. For example, if a vertex is to be blended into world space using three matrices as above, the vertex would contain three weights describing how strongly the transformation in each matrix will contribute to the final vertex position. If we were to sum these weights, they should add up to 1.0. If the vertex used in the above example had weight values of 0.25, 0.5, and 0.25, then we know that the matrix M1 would contribute 25%, matrix M2 would contribute 50%, and matrix M3 would contribute 25% to the final vertex. Therefore, after we have generated the temporary vertex for each matrix, we can scale each temporary vertex position by the corresponding weights. The second part of our vertex blending procedure would do just that:

```

TempVertex1 = TempVertex1 * ModelVertex.Weight1; //( Weight1 = 0.25 )
TempVertex2 = TempVertex2 * ModelVertex.Weight2; //( Weight2 = 0.5 )
TempVertex3 = TempVertex3 * ModelVertex.Weight3; //( Weight3 = 0.25 )

```

Now that we have scaled each temporarily transformed vertex position by the corresponding weight, we sum the results to create the final blended world space vertex position.

Blended Result = TempVertex1 + TempVertex2 + TempVertex3;

That is all there is to vertex blending. Note that although it is not *required* that the weights add up to 1.0, there may be strange and unpredictable results when they do not. We generally want our fractional transformations to add up to one complete transformation.

The following code gives us a bit more insight into the vertex blending process. It takes a model space vertex, an array of matrices, and an array of vertex weights (one for each matrix in the array) and returns a final blended world space vertex.

```

D3DXVECTOR3 TransformBlended (D3DXVECTOR3 *pModelVertex ,
                             D3DXMATRIX *pBlendMatrices ,
                             float * pWeights , float NumOfMatrices )
{
    D3DXVECTOR3 TmpVertex;
    D3DXVECTOR3 FinalVertex ( 0.0f , 0.0f , 0.0f );

    for ( int i = 0 ; i < NumOfMatrices; i++ )
    {
        TmpVertex = ( *pModelVertex * pBlendMatrices[i] ) * pWeights[i];
        FinalVertex += TmpVertex;
    }

    return FinalVertex;
}

```

Vertex blending is the key process behind character skinning and skeletal animation. With vertex blending, we can state that some vertices in the mesh are connected to more than one bone in the hierarchy, like our elbow vertices as discussed previously. In such a case the vertices in this area might be attached to both the upper arm bone and the lower arm bone. Since each bone is just a frame in the hierarchy, we can transform the vertices in this region of the mesh into world space simply by performing a blend (as shown above) using the matrices for both the forearm bone and the upper arm bone. When either or both of these bones are animated, the vertices of the skin move and change shape to accommodate the new bone positions, much like our skin does in real life. Because all of the vertices in the skin are a single mesh, we do not suffer the crack problems between the joints of the character's limbs. We will see later that it is the job of the 3D artist to construct the bone hierarchy and assign vertices in the mesh to these different bones. The artist will also be able to alter the weights of each vertex to tweak the model and get the weights of all influential matrices for a given vertex just right.

From the programming perspective, one of our first jobs will be to load the skeleton. Fortunately, we already know how to do this because a skeleton is just a D3DXFRAME hierarchy loaded automatically

by the `D3DXLoadMeshHierarchyFromX` function. Each frame in the hierarchy will be a bone and its frame matrix is called a *bone matrix*.

NOTE: While we are now referring to the frame hierarchy as a *skeleton* and the frames as the *bones* of this skeleton, nothing has actually changed with respect to chapters 9 and 10. This is just a naming convention we are using because it is more appropriate under these circumstances.

Most gamers were quite impressed the first time they saw the skeletally animated skinned mesh characters in the original Half-Life™. Their smoothly deforming skins set a precedent that could not be ignored. Pretty soon, almost all new games were implementing character skinning and skeletal animation as the technique of choice, and the characters in computer games have become a lot more believable as a result. Keep in mind that skinning techniques are not reserved exclusively for character animation only. One of the lab projects in this chapter will use skinning and skeletal animation to create some very nice trees for our landscape scenes. Such a system works quite nicely for this purpose and you will soon be able to play wind animations that cause swaying branches and rustling leaves.

While increased CPU speeds meant that vertex blending could be performed in real time, things are even better today since vertex blending can now be performed by the GPU. This is true for most T&L graphics cards released since the days of the first GeForce™ cards. The DirectX 9.0 API exposes these features to our application so that we can perform hardware accelerated vertex blending on the skin and bones system that we implement.

There are a number of additional advantages to using a skin and bones system beyond the removal of cracks and seams discussed earlier. First, the artist has to do very little repetitive mesh building. The artist only has to create a single mesh in its ‘reference pose’ and then use the modeling package of his/her choice to fit a skeletal bone structure neatly inside the mesh. After assigning which vertices will be influenced by which bones and specifying the appropriate weights, the main work is done. From that point forward it is a matter of applying any animation desired to the skeleton to see the results.

Another great thing is that many of the more popular commercial modeling packages export the skin and bones data into X file format. The mesh is stored as a regular mesh and the skeleton is just stored as a vanilla frame hierarchy (like the ones we looked at in previous lessons). Later in the chapter we will examine all of the X file data in detail.

From our perspective as programmers, we benefit from the fact that animating the bones of a character is no different from animating a normal frame hierarchy. Thus our last chapter will serve us well as we work through our lab projects. We will still use the animation controller to choose the animation we wish to play and call `AdvanceTime` to update the frame matrices in the hierarchy. Once the matrices have been updated, we traverse the hierarchy and build the absolute world matrices for each frame (bone). Then all that is left to do is set the matrices and render our mesh.

Transforming and rendering the mesh is where things will change a fair bit. We must now be able to set multiple world matrices on the device so that the transformation pipeline can calculate the world space position of each vertex using all of the world space bone matrices to which it is connected. We will also see how the vertex structure of the mesh has to change so that it can store a number of weights (one for each bone matrix that affects it). Once the device has this information however, we can call `DrawPrimitive` as usual and let the pipeline render each subset. The device will use the multiple matrices

we have set from our bone hierarchy to perform the vertex blending. This is certainly going to be exciting material for sure, but we do have a fair bit to learn about before we can perform this process properly.

We have now had a rapid tour of what skinning and skeletal animation is and why it is the technique of choice for most game programmers. Very little code is needed to implement it, and it takes up very little memory compared to other techniques (such as tweening, as we will see next). It also gives us control over every aspect of the character's body. You can instruct the artist to create bones that can be moved to cause wrinkles in the skin, twitch the character's nose, or cause the characters hair to blow in the wind. These are all very cool animation effects that we are seeing in today's computer games and now we are going to learn the foundations for how to do achieve this. But first, we will take a very brief diversion...

11.1 Vertex Tweening

There are alternatives to skinning that can produce very good results. One popular choice is called **tweening** (or **morphing**). id Software® used this technique in the Quake II™ engine to produce smooth character animations without cracks or other visual artifacts.

Tweening is similar to the keyframe animation approach we discussed in our last lesson. In the case of Quake II™ for example, the file (.md2 file in this instance) stores a number of keyframes, but these keyframes do not store a timestamp and transform (SRT) data. Instead they store a timestamp and an entire mesh in a given pose. Each mesh represents exactly what the vertex data of the mesh should look like at the specified time stamp. The file will therefore contain a large list of pre-posed meshes.

Let us imagine an .md2 file containing a 'walk' animation cycle. The file will store a number of meshes, where each mesh can be thought of as a snapshot of the character at a certain time throughout the life of the walk animation.

To render the mesh for a given time (say $T = N + 0.5$ -- where N is an arbitrary time through the duration of the animation) the first thing we would do is find the two keyframes in the mesh list that bound this time. In this example it would be keyframe N and keyframe $N+1$. Our two keyframes would now be two separate meshes of the character in different poses as shown in Fig 11.1.

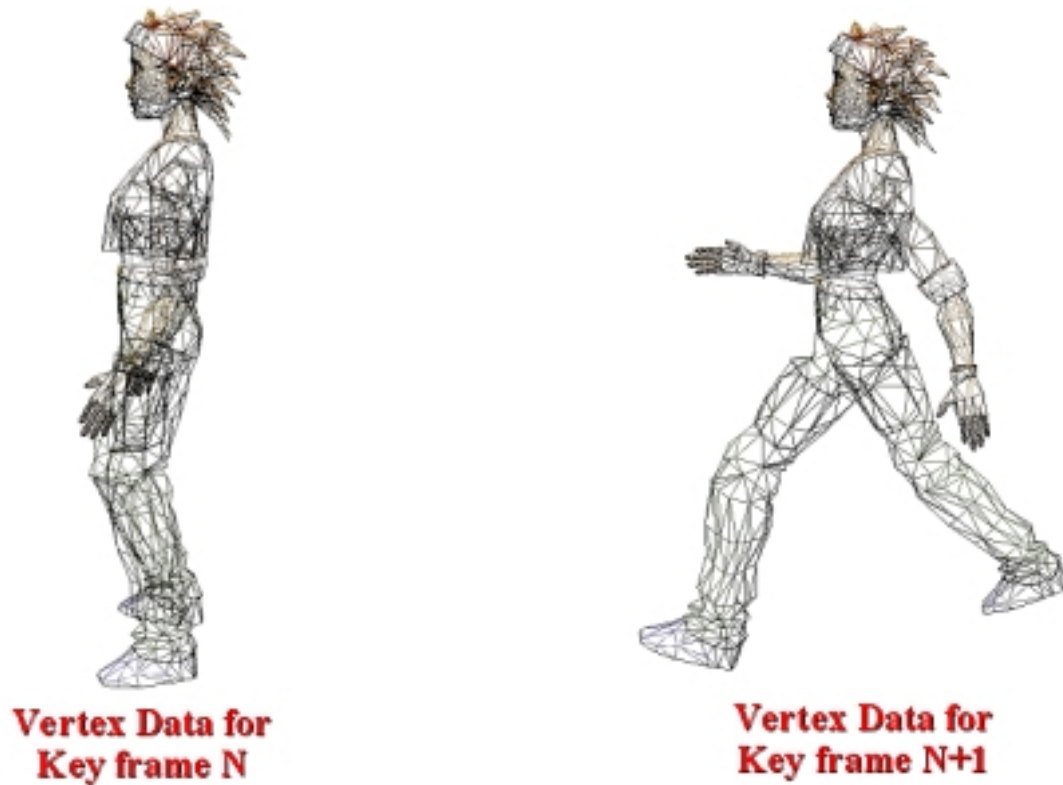


Figure 11.1

To calculate the final position of each vertex in the mesh we will interpolate between matched vertices in both sets. We will use the amount that the requested time is offset between the time stamps of both keyframes as the interpolation weight factor. So we are actually building a new mesh on the fly by interpolating between the vertex sets of two input meshes. The interpolation would look something like the following pseudo code:

```
t = AnimTime - KeyFrame[N].Time;

for( I = 0 to NumVertices )
{
    Vertex V1 = KeyFrame[N].Vertex[I];
    Vertex V2 = KeyFrame[N+1].Vertex[I];

    FinalMesh.Vertex[I] = V1 * (1.0 - t) + V2 * t
}
```

In the current example, AnimTime is the time we request to render the mesh and N and N+1 are assumed to be the indices of keyframes that bound the requested AnimTime. Since each mesh is simply the same object in different poses, we know that there will be the same number of vertices in both of the source meshes. We also know that there will be exactly this many vertices in the dynamically generated output mesh.

If we imagine that we would like the mesh generated for an animation time of $N+0.5$ (midway between the two keyframe poses above) we would have a temporary vertex buffer containing the interpolated vertex data. The mesh would look like the results in Fig 11.2.



**Temporarily generated
'Tweened' Vertex Data**

Figure 11.2

We could now transform and render this mesh just as we would any other standard mesh.

So as you can see, during each frame we generate a new model space mesh by interpolating between keyframe meshes. Once we have the vertices for both of the source meshes, we interpolate between them to generate the in-*between* pose, which is where the term *tweening* originates.

While this is a simple technique to implement in software and it does allow us to use single skin meshes, it does have its drawbacks. The main problem is that it can consume a good deal of memory because we have to store multiple copies of the mesh vertex buffer (one for each key pose).

DirectX 9 exposes hardware support for tweening and many graphics cards support the acceleration of tweening on the GPU. However, we will not cover tweening in this chapter for a number of important reasons. First, we cannot implement tweening without using declarator-style vertex formats (non-FVF) and these will not be discussed until Module III in this series. Second, many programmers have now abandoned tweening/morphing as a means of generically animating characters in their games in favor of skin and bone systems (mostly due to the memory costs). However, that is not to say that vertex tweening is without value -- far from it. Tweening is still a very popular technique in the area of facial animation. With the release of Half-Life 2™, facial animation is certainly something that is on the minds of game developers everywhere. An artist can build multiple facial expressions that can be easily morphed from one to the other to add lots of additional personality to a game character. However, it is worth noting that skin/bones can also be used for this purpose and is rapidly gaining popularity. Tweening will be covered in a little later in this course series, so do not worry too much about it for now.

11.2 Segmented Character Model Animation

For completeness, before we work our way up to skinned meshes and skeletal animation we will first look at how we might perform character animation the old fashioned way -- by using what we learned in the previous chapter on hierarchical animation. This is the way characters used to be implemented in the games of a few years ago. This was because per-vertex processes (such as vertex blending) were too expensive to be performed in real-time due to the low CPU speeds and the lack of 3D hardware acceleration in most end user systems. Taking a look at this older approach however should help us understand the origins of skinning as well as why skeletal animation and skinning are such useful techniques to have in our tool chest.

Armed with only the animation knowledge from the previous chapter and instructed to create an animated character that can play a number of animations (such as walk, run, and jump for example), we might imagine that we could use the exact same system from Lab Project 10.1. We would perhaps instruct our game artist to build a character object using multiple mesh body parts and arrange them in a hierarchy inside the modeling program. This is no different from how we constructed the space scene in the previous lesson. We had multiple meshes (the space ship, the bay doors, and the radar dishes) each attached to frames in a hierarchy and it all worked nicely. The fact that we are now using the hierarchy to represent meshes of human body parts should theoretically work just as well.

For the sake of simplicity, let us work only with the bottom half of a character. This lower body might consist of five separate meshes: a pelvis, a left upper leg mesh, a left lower leg mesh, a right upper leg mesh and a right lower leg mesh. The five separate meshes might be arranged in the hierarchy as shown in Figure 11.3.

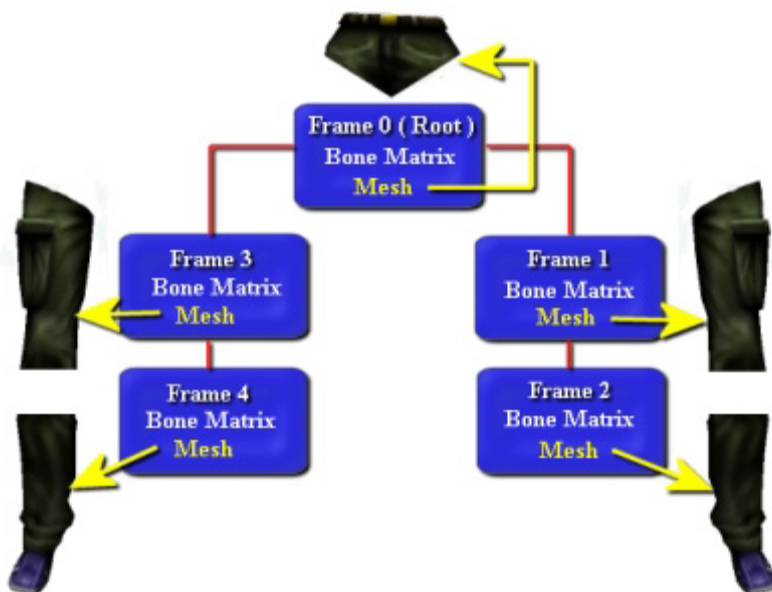


Figure 11.3

You should already have a good idea about how this frame hierarchy would be exported and stored in an X file. Certainly there would be little trouble loading it in using `D3DXLoadMeshHierarchyFromX` as we learned in previous lessons. Because we are now using the frame hierarchy to represent a character instead of a generic scene, we will refer to each frame in the hierarchy to which a body part is attached as a 'bone'. This works well because they will behave very much like a set of human bones. That is, when we rotate one of our bones in real life, our skin and clothes around that bone move also. Likewise, when we rotate a frame in our hierarchy, a single body part attached to that frame will move (along with any child frames and their respective meshes). So we will think of each frame as an individual bone and the frame hierarchy as a skeleton. In Fig 11.3, we can see that the pelvis is the root bone and that it has a pelvis mesh child. It also has two child bones which are the upper parts of each leg (along with their meshes). Each upper leg bone also has a lower leg child bone (each of which has a lower leg mesh).

After the artist has constructed the hierarchy of meshes in Fig 11.3 he could create a series of appropriate animation keyframes for walking, running, crouching, etc. and export everything to an X file. From our perspective, nothing has changed. We can still use `D3DXLoadMeshHierarchyFromX` to load this character model hierarchy using exactly the same code we used in the last chapter to load our scene (a scene of body parts now). We will get back a fully loaded hierarchy along with an `ID3DXAnimationController` so that we can start assigning the different animation sets (walk, run, etc.) to tracks on our animation mixer. To play a given animation requires only a call to `ID3DXAnimationController::AdvanceTime`, just as we saw in our last two lab projects.

So it seems as if we have achieved character animation free of charge simply by re-using the code we already have. However, as discussed in the introduction, this approach is not without its problems. The obvious problem with representing characters using the segmented character model is that in real life our various limbs are not separate detached entities. While it is true that our bones most definitely are individual segments of our skeleton and that they can move independently, our skeleton is covered with muscle and a continuous layer of skin which behaves very differently. Because this is not the case with our segmented character representation, certain artifacts which do not arise in real life become unavoidable.

Fig 11.4 shows one of the legs in our segmented character. We know already that the leg consists of two bones and two meshes -- the upper leg mesh is attached to the Frame 1 bone and the lower leg mesh is attached to the Frame 2 bone. These are the frame matrices in their default pose (commonly referred to as the *reference pose*) where the character meshes have been carefully arranged so that no cracks or spaces can be seen between the two separate leg meshes. The red dashed line in Fig 11.4 shows the boundary where the two leg meshes meet.

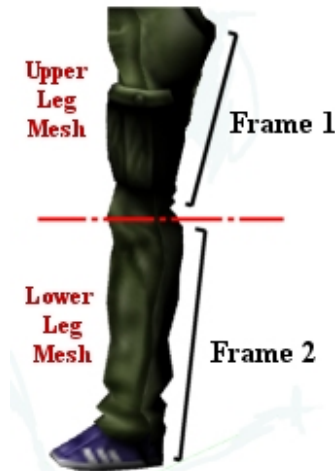


Figure 11.4

If we were to rotate Frame 1 (which we can think of as the joint at the top of the upper leg mesh where it meets the pelvis) relative to its pelvis parent, the leg would rotate without any artifacts. The lower leg is a child of the upper leg, so it too would be rotated. The entire leg would rotate much like the hand of a clock. However, if we were to rotate just the Frame 2 bone, (located in the knee area at the center of the leg) we would want the leg to bend in the middle (required if we want to play a realistic walking or running animation). But Fig 11.5 shows us what happens to the segmented character model system if the Frame 2 bone is rotated (thereby rotating its attached mesh).

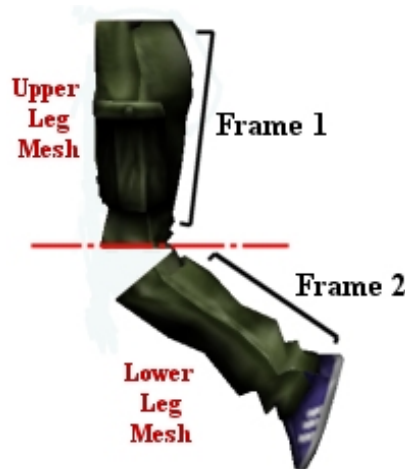


Figure 11.5

Fig 11.5 makes it instantly clear that the leg mesh is not one smoothly skinned entity, but is in fact two meshes. While this artifact can be minimized to some degree by careful modeling of the separate body part meshes (ex. tapering and overlapping them where they meet so that the crack is not so obvious) this is not something we can entirely avoid using the segmented character model. The problem is that each mesh is also being treated exactly like its associated bone. If we were to remove the skin from our own bodies, we would see that our skeletons would suffer the same artifacts (where each bone would be connected by a ball joint for example but would appear quite distinct).

So what are we to do to remedy this problem? Tweening would be one possible solution, but we have already discussed why it might not be the best choice for us. Beyond the memory concerns, consider as well that we spent a good deal of time learning how to animate hierarchies and use the D3DX animation system. We have already experienced the power and flexibility of the animation controller and it would be a shame to have it stripped away from us. Ideally we want a system where we can still store and play our animations using the same techniques we are already comfortable with but that addresses the concerns discussed thus far. This means that we want to represent our characters using a bone hierarchy (just like the previous example) but we do not want to use a segmented geometry model.

Note: One final point in favor of using a hierarchical bone structure over a tweening system that is worth thinking about is that tweening is quite restrictive. Essentially we are locked into having to interpolate frames provided by the artist. With a skeletal structure, while we will most often be playing back animation created by the artist in a modeling package, we still have the freedom to programmatically apply transformations to any bones in the hierarchy we see fit (even at run time). In that sense, there are an infinite number of ways that we can animate a skeletal structure.

11.3 Skinning

Mesh skinning, used in conjunction with a skeletal structure, provides the solution to the problems previously discussed (although it is not without its own issues). The skeleton remains the same – it is a frame hierarchy and we can generate and play back SRT keyframe animations exactly as we did before. But instead of assigning separate meshes to individual frames, we are now going to use a single mesh, referred to as a *skin*. That is, rather than assign different meshes to different bones we will instead assign different *vertices* within a single mesh to different bones.

In the simplest case, each vertex in the skin is assigned to only one bone. We could theoretically imagine the skin as being constructed from separate sub-meshes, where each sub-mesh is a group of vertices attached to the same bone. The difference is that all sub-meshes share a single vertex buffer because they are all part of the same mesh object. In the more complex (and probable) case however, the logical division of the mesh into imaginary subsets is not so clear because we can assign a single vertex to more than one bone in the hierarchy. When a vertex is to be transformed, several bone matrices are used (to varying degrees) to determine the final world space position of that vertex. When a bone is rotated, any vertices that are assigned to that bone will be rotated also. The degree to which they will rotate will depend on the weights specified for that bone on each vertex. For the moment, do not worry about how the mesh is stored in the hierarchy or how the vertices are attached to frames, we will get to that a little later in the chapter.

While the artist is responsible for generating this mesh and constructing the bone hierarchy such that it correctly describes the mesh in its reference pose, the artist will also be responsible for determining which bones in the skeleton will influence each vertex, and by how much (using the weight value). This is not as hard as it might seem and we will discuss creation of skin and bone data from the artist's perspective in just a moment.

Fig 11.6 depicts the leg section of our single skin as it might look in our skinning application. The bones still exist in the same places since the skeleton is unchanged from our previous segmented example (Fig

11.6). The leg mesh depicted here however is now part of a single unified skin. In this example we are focusing on the knee joint (the Frame 2 bone). Most of the vertices in the lower leg section of the skin might be assigned to just the lower leg bone (Frame 2) and most of the vertices in the upper section of the leg might be mapped to just the upper leg bone (Frame 1). However, the vertices about the knee joint (within some limited radius) are probably going to need to be influenced by both bones.

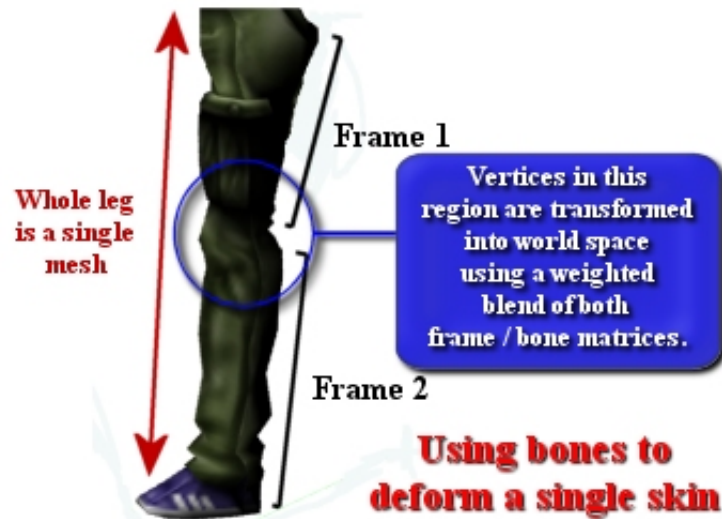


Figure 11.6

Because this is a single mesh, when the knee bone is rotated (Frame 2) all of the vertices attached to that bone will also move by some amount determined by the weight stored in the vertex for that bone. The vertices outside the blue circle in the lower part of the leg will be 100% influenced by this bone because they are attached to only the Frame 2 bone. However, the vertices in and around the knee joint are also influenced to some degree by both bones, so their positions will change also but not quite to the same extent. As the vertices in this area move, they stretch the skin polygons of which they are a part so that they occupy the space that would have caused a gap in the segmented character model. We see in Fig 11.7 that this rotation causes no cracks to appear -- the section of the skin around the knee joint smoothly deforms to give the impression of a single skin covering our skeleton.

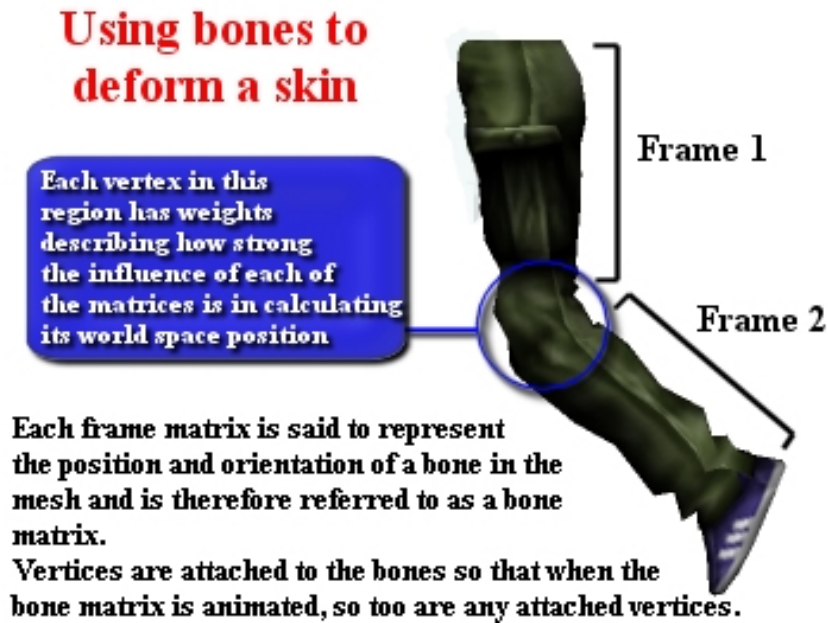


Figure 11.7

The result in Fig 11.7 is much more realistic than our earlier segmented character results.

As mentioned earlier, DirectX 9.0 supports hardware acceleration of the vertex blending process used to transform each vertex from model space into world space. There is support for using up to four different bone matrix influences per vertex, which is generally more than enough. Since many of the more recent graphics cards provide acceleration of vertex blending in hardware, skinning becomes an exciting prospect. Further, even if hardware support for skinning is not available on the user's machine, we can instruct DirectX to use software vertex processing to transform our skins. In this case the vertex blending will be carried out in the software module of the pipeline. Unlike some of the software processing alternatives supported by the DirectX pipeline, the software vertex blending module is very efficient and can be considered extremely useful even in a commercial project. You may even find that on some systems (depending on the video cards and CPU speeds) there is very little performance difference between hardware accelerated blending and software vertex blending performed by the DirectX transformation pipeline (although to be clear, this is not typically the case). The bottom line is that with or without hardware acceleration, we can still perform real-time skinning.

11.3.1 Generating Skinned Mesh Data

We already know how to animate a skeletal hierarchy but we still have no idea how the skin will be represented in memory. Nor do we know how to let the transformation pipeline know which vertices in the skin should be influenced by which matrices in the hierarchy and what weight values should be used. Rather than dive straight into API specific function calls, let us start with an analysis of the data first. It will be helpful to start at the beginning of the process and briefly discuss the steps the artist must take to assemble the skin and skeleton. It will also be helpful to know how that data will be exported into X file format. This means we can examine how we load the bone hierarchy and the accompanying skin from

the X file so that we are familiar with all of the data structures and where they come from. This knowledge will make it very easy to understand how to work with skinned meshes in the DirectX pipeline. So let us start our discussions at the art pipeline stage.

While we cannot possibly hope to cover the implementation details of generating skinned meshes in the modeling application (there are entire books devoted to the subject) we will take a look at the basic steps an artist must take in order to create the skinned mesh data for the programmer. This provides extra insight into how the data is generated and stored and thus will aid us in understanding the key concepts of character skinning in our game engine. This section will be kept deliberately brief and generic as the exact process for creating a skinned character and a skeletal hierarchy will vary between modeling applications.

1. Creating the Skin

The first thing the artist will usually do when generating the data for a skinned character system is to build the skin. The skin is just a mesh of the character in its default pose. There is no hard and fast rule about what the default pose should be, but modelers will usually prefer to start off with a pose that allows the skeletal structure to be more easily placed inside the character mesh. As it pertains to building the skin vertex data, this is all the artist typically has to do. Unlike a tweening system where the artist would have to create several copies of the same character mesh in many different poses, in the character skinning system, the artist just has to make this one mesh in its reference pose. Fig 11.8 is an example of a character skin mesh. As you can see, it is just a normal geometric character model and there is nothing special or unique about it. This mesh can be created in whatever modeling application is preferred so long as it can be imported into the application that will be used for building and applying the skeletal structure.

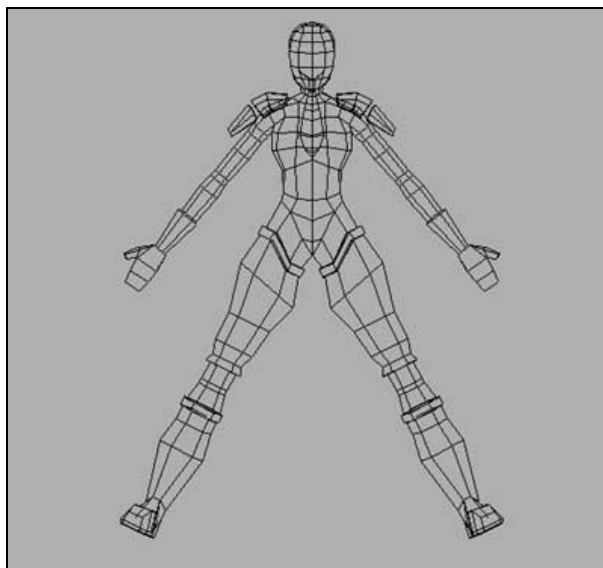


Figure 11.8

Note: The process of creating a set of individual meshes for use in a tweening system can actually be aided by the skeletal animation system most 3D modeling programs provide. For example, an artist can create a skeletal character and animate it within the package, but rather than export the skin and skeleton (which we would do for use in a skinned mesh system), he/she could just save out the transformed meshes at individual keyframes throughout the animation cycle. This essentially means saving the mesh data only and discarding the skeleton data as it will no longer be needed. Whether or not you wish to do this when the skeleton is available depends on the needs of your project, but it is important not to assume that each individual keyframe pose used in tweening must be a unique mesh constructed by hand every time.

2. Applying a skeleton to the skin

Once the mesh is imported into the application that we are going to use to apply the skeletal structure, the next step is for the artist to map this skin to a hierarchy of bones. The process of building and applying a skeleton will vary between 3D modeling packages, but the basic concepts are generally the same. In this discussion we will look at some screenshots of various processes and describe those processes with relation to Character Studio™. Character Studio™ started its life as a powerful animation plug-in tool for 3D Studio MAX™, but in the latest versions of 3D Studio MAX™ has now been fully integrated as part of the MAX package. If you have an earlier version of 3D Studio MAX™ (pre version 5.0) then you will need to purchase the Character Studio™ plug-in separately.

Character Studio™ allows you to visually insert skeletal structures inside of your skin mesh. Although you can build your own skeletal structures (linking one bone at a time together into a hierarchy), Character Studio™ ships with several pre-packaged skeletal structures which you can use directly. The package also ships with numerous test-bed animation scripts (more can be downloaded, purchased or created) that can be assigned directly to the skeletal structure. The test animations run the gamut from smooth motion captured walking scripts to slipping on a banana peel. Many of them are actually quite fun to watch, even using just the default skeleton.



The skeletal structure is referred to as a **Biped** (for obvious reasons) and the animation files are generally stored with a .bip extension. Below you can see the Biped rendered as a simple skeleton in Character Studio™.

While it is not much to look at (perhaps a bit like the Battle Droids from Star Wars™) it is in fact the graphical representation of what will eventually become our bone (frame) hierarchy.

The pre-packaged (or downloadable) test animation scripts are specifically designed to animate this skeletal structure. Therefore, all we have to do is fit this skeletal structure inside our mesh, attach it, and our mesh will benefit from those same animations.

To fit the Biped to the mesh, the artist will scale and orient its limbs to match the reference pose of the skin (this is why it is helpful to keep the arms and legs splayed out away from the torso as shown in Figure 11.8).

Fig 11.9 shows what the current biped might look like after being fitted to the skin shown in Fig 11.8. Notice how the joints of the biped have been rotated and scaled so that they accurately represent what the skeleton of our mesh would look like in its reference pose. The rotation and position of each bone joint will be recorded and will be described by a bone matrix in our hierarchy when the skeletal structure is exported to X file format. We will learn more about that a little later in the lesson.

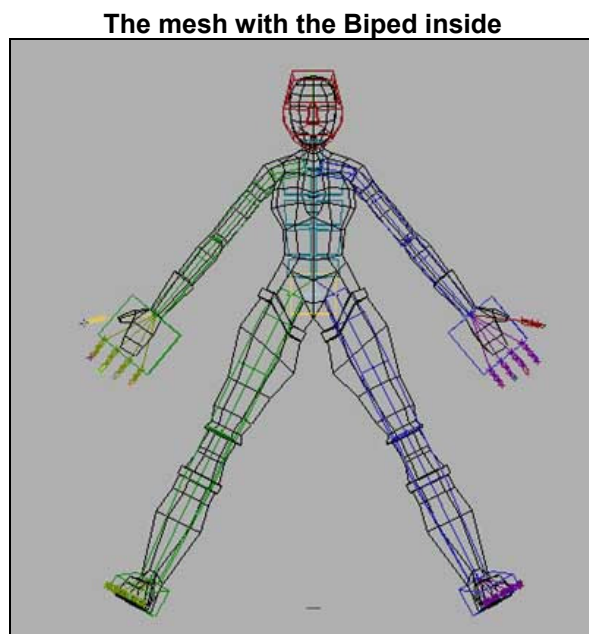


Figure 11.9

Once the skeletal structure has been properly aligned with the skin, the artist will use the 3D Studio MAX™ Physique Modifier to auto-calculate which vertices in the skin are influenced by which bones in the skeleton. It is probable that some (or even many) vertices in the skin will be influenced and thus attached to more than one bone in the skeleton. We saw this earlier with our knee joint.

Furthermore, the weight of each matrix's contribution to each vertex is also calculated for all influential bones for a given vertex. The default calculation that 3D Studio will do for us will involve testing each bone matrix for each vertex to see if that vertex is in a region of influence (also referred to as an *envelope*) for a given matrix. If so, then that vertex will be attached to that bone and the weight of the matrix for that vertex will be calculated as a product of the distance from the vertex to the bone. To calculate the region of influence for a bone, an ellipsoidal bounding volume is used around each bone. Any vertices that fall within this ellipsoidal region will be attached to the bone and influenced by the bone to some degree.

3. Tweaking the bone influences and skin weights

Although the Physique modifier is a completely automated process, after it has been applied, it is not uncommon to find that the mapping of the skin to the skeleton requires some manual tweaking. For example, the default envelope size used to calculate a region of influence for each bone may well

include vertices that we would not like to be influenced by the movement of that bone. Alternatively, we might want the vertex influenced by a bone, but perhaps not to such a large degree. In some cases, we may even find vertices that were not attached to the skeleton at all. By playing the test-bed animation scripts we can easily identify problem areas and correct them. We simply look for places where the skin is not animating in as natural way as we would like given the default matrix contributions and skin weights automatically calculated for us. We will often find that by adjusting the region of influence for a given bone we can address these inaccuracies. The process involves a good deal of trial and error, with us running the animation scripts again and again between tweaks to verify the adjustments just made. Again, this is all in the domain of the project artist, but is helpful to understand how it works (especially if you intend to do your own art and animation).

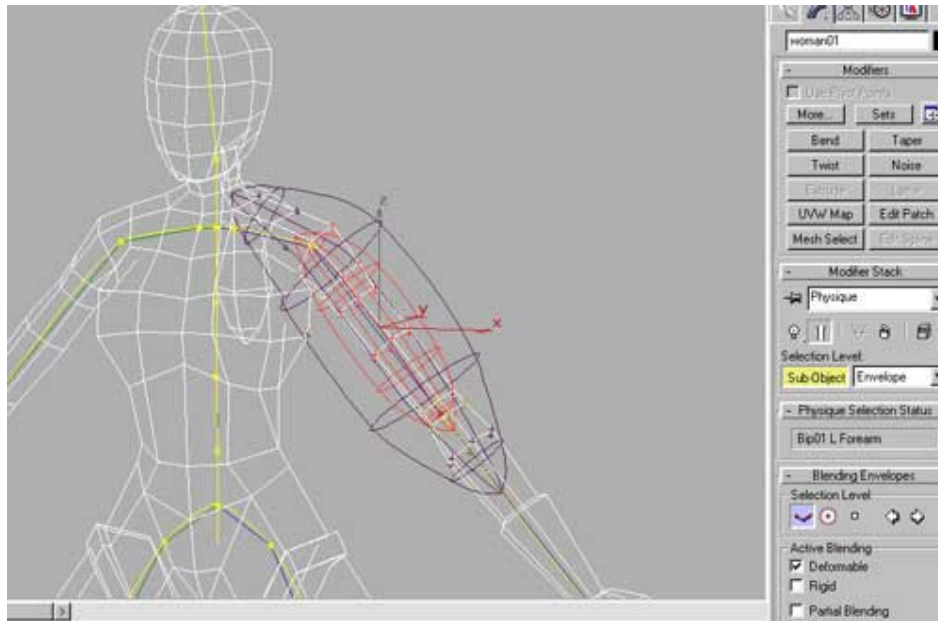


Figure 11.10

Figure 11.10 is a screenshot of an interface for adjusting the envelope (region of influence) for the upper arm bone using the Physique modifier. The size of the ellipsoidal region of influence for that bone can be adjusted to consume more or less of the surrounding vertices of the skin.

4. Applying Animations

Once the skeleton and skin system has been tuned, any number of pre-canned animations can be selected to animate the skeletal structure. This automatically animates the attached skin. There are many animation scripts available for Character Studio™ (usually .bip or .mot files) that can be purchased or downloaded, but most game projects generally entail the artist coming up with original data. Whether this is done by hand in the animation program or by spending time in a motion capture studio, most games involve their own proprietary animation sequences. Ultimately we need keyframes for the animations timeline one way or the other.

5. Exporting the character skin and skeletal structure

Once the artist has the skin and bone system animating correctly, the next step is to get that data into a format that our application can easily load and process. While the DirectX SDK ships with X file exporters for both Maya™ and 3D Studio Max™, there are third party exporters available as well. One of the better free X file exporters available for 3D Studio MAX™ is the Panda™ DX Exporter plug-in. This is a great exporter that converts all the data we need into X file format, including not only the skin and bone data, but the animation data as well. The skeletal structure will be exported as a vanilla X file frame hierarchy and the animation data will be exported as standard keyframe animation data. The only thing we will need to come to grips with is the extra data stored in the X file containing the mapping information which describes which bones in the hierarchy influence which vertices in the mesh. We will discuss this in a moment.

Note: In many file exporters that work with 3D Studio MAX™, you have to be very careful not to mix MAX bones with Character Studio bones (they are different). Keep this in mind when you are designing your skeletons.

If you have 3D Studio MAX™ and wish to use the Panda™ DX Exporter you can visit their website and download it for free at <http://www.pandasoft.demon.co.uk/directxmax4.htm>.

There are other plug-in exporters available for 3D Studio MAX™ which do not export the data in X file format, but rather in some other easy to read format. The excellent Flexporter™ plug-in, designed by our colleague and friend Pierre Terdiman, exports to a proprietary file format called .ZCB. While .ZCB files are easily read, the real power of Flexporter™ comes from the fact that it is actually a plug-in SDK that lets you define your own file formats and write your own plug-ins. Flexporter™ will simply provide your plug-in with the data it reads from MAX and you can store it however you see fit. This saves you the trouble of having to learn the entire MAX Plug-In SDK. You can download the Flexporter™ plug-in SDK at <http://www.codercorner.com/Flexporter.htm>.

Finally, we would be remiss if we did not mention some of the non-free export tools available to you as well. One of the most popular workhorse applications available for dealing with export of scene data and animation is Okino Software's PolyTrans™. It is a very powerful tool that supports data import and export from almost all of the popular 3D modeling programs (MAX, Maya, Lightwave, etc.) and it has an X file exporter built in. This means that you can support art and animation from just about every modeling tool on the market and get it all into X format with a few clicks of a button. A similar tool that has most of the same bells and whistles as PolyTrans™ is DeepExploration™. Neither of these tools are free (unlike the other exporters just mentioned), although they are both reasonably priced given their power and flexibility. If you have the budget, either one is a worthwhile investment for any professional project.

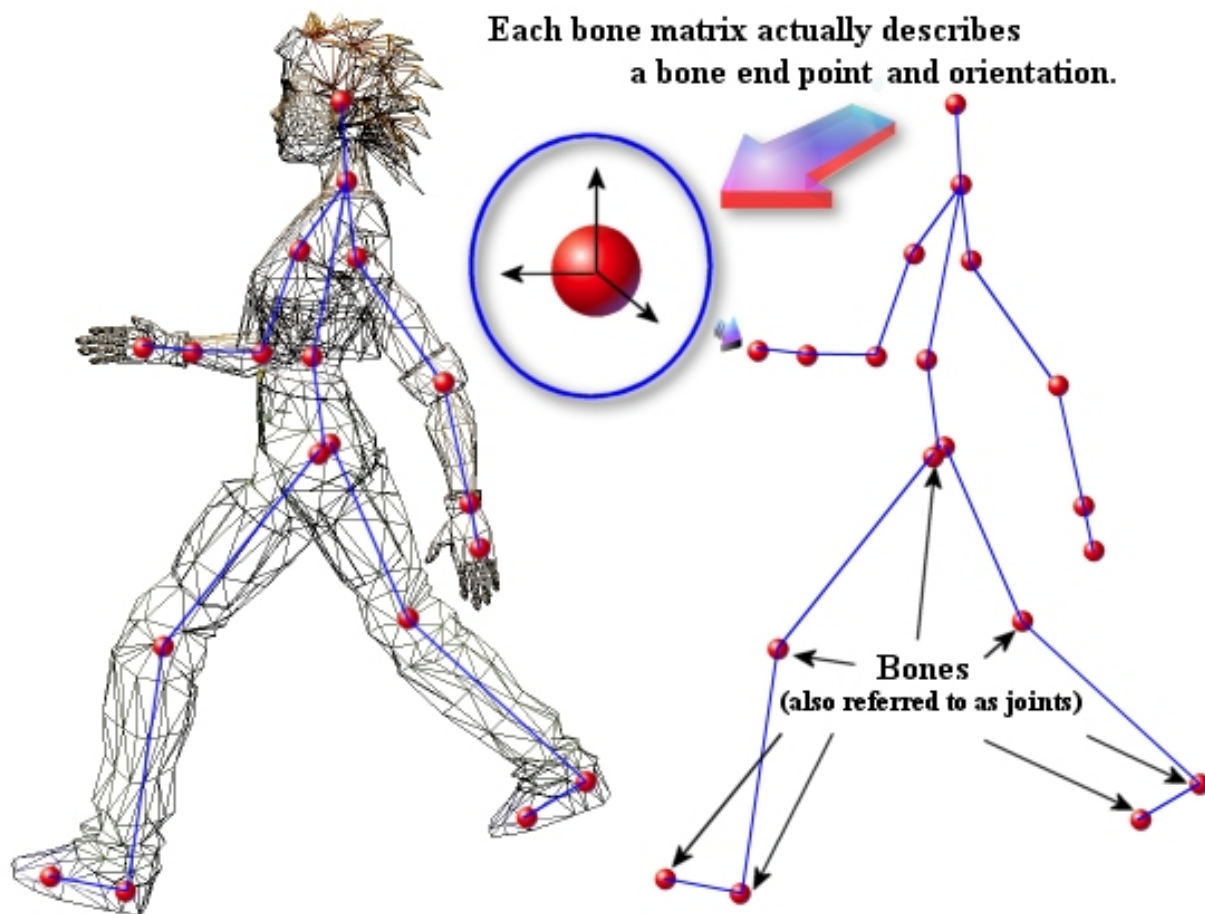
So even if none of the X file exporters fit your needs, you could always use a system like Flexporter™ and write some code to build your own plug-in file format. If you intend to use the D3DX animation system then you will need to write the code that also uses this data to populate the animation controller, its animation sets, and its underlying keyframe interpolators. Fortunately, we learned how to do this in the last chapter, so you have some knowledge of how to approach such a system.

We now understand skin mesh creation at a high level. Although there will be differences between various modeling applications, the underlying system we are creating still consists of two separate entities: the skin and a skeletal structure. When saved out to an X file, the skeleton is saved as a frame hierarchy that `D3DXLoadMeshHierarchyFromX` will load automatically. Each frame in the X file will be a bone in the skeleton and the matrices stored in each frame will contain the parent-relative transformation of that bone. Each bone in the hierarchy will have also been assigned a name so that the animation controller can manipulate the bone matrix. This also allows us to identify the matrices which affect each vertex. This will all be dealt with at load time and, as we will see later, is something that we will need to know in order to make sure that the Direct3D device has the correct bone matrices available when rendering specific subsets of the mesh.

For the remainder of this chapter we will be assuming that you have exported your skinned character and skeletal animation structure into X file format. You should have a single X file which contains at least one skin, one skeletal hierarchy, and a collection of animation data. You do not have to export keyframe animation data with the skinned character since you may choose to apply the animations to the bone matrices programmatically instead. However, by storing the various animations in separate animations sets inside the X file, you have the benefit of using `D3DXLoadMeshHierarchyFromX` to load the various animation sets and register them with the animation controller. This way you can just select the pre-canned animation you wish to play, assign it to a track on the animation mixer, and start animating your character immediately. If you do not have a character X file handy to practice with, you can use the samples accompanying this course, or those (like `tiny.x`) that ship with the SDK.

11.3.2 Bones and Joints

In Fig 11.11 we can see that the animation data manipulates the bones in the skeleton and that because the skin vertices are attached to those bones, the entire mesh is animated as a result. Although in 3D Studio MAX™ (and in Fig 11.11) we have rendered the skeleton as a physical entity existing inside the skin, it will not exist in our applications as a visual object. The bones of the skeleton are ultimately just matrices describing a position and orientation relative to a parent bone in the hierarchy. We render them just to get a feel for what is happening behind the scenes.



It is common to represent the hierarchy in diagrams by connecting up the bone matrix positions with lines to form a skeleton. Whilst this is useful for visualizing the underlying skeleton that the mesh is using, we must remember that in such diagrams the lines themselves are not what our bone matrices represent, each bone matrix actually represents a bone end point.

Figure 11.11

As Fig 11.11 mentions, it is very common for people to think of the bones themselves as being the links between joints. While of course this is technically true in real life, it is important to bear in mind that a bone in our hierarchy behaves much more like a joint. That is, In the context of a skeletal structure in computer graphics, it is perhaps more intuitive to think of a bone in the hierarchy as being just the bone end-point (or a *joint*).

Recall that the 'bone' is a frame of reference in our system. Therefore, in the diagram we can see that the bones in our hierarchy will actually be the red spheres shown in the skeleton and technically not the blue lines connecting them. It can be a bit confusing that the word 'bone' is the more common way to refer to these frames instead of the (perhaps more applicable) term 'joint' which better describes what is being animated. Technically, it is really both (the joint plus the bone that emerges from it – much like the Z axis emerging from a coordinate system origin) but the actions that we apply to a given bone really take place at the joint itself (i.e. the system origin).

Note: Many modeling applications and texts on the subject of 3D skeletal animation still refer to bones as 'joints'. We will use the 'bone' terminology in this course in keeping with the DirectX naming convention for such entities but just try to keep this concept in the back of your mind.

To really understand the bone/joint equivalence concept, take another look at Fig 11.3. Because we are describing Frame 1 in this example as being the 'upper right leg bone', we might envisage this bone as being a long white thing that spans the length of the upper leg mesh. While this is understandable, and not entirely false, just keep in mind that a bone is described by a matrix that contains a position and an orientation. The important point to note is that the position in that frame matrix describes the *end-point* of the bone that starts at the top of the leg. Keep in mind that like any line, the term 'end-point' can apply to both the terminal point (the 'end') of the bone as well as its origin (the 'beginning'), but that we are referring to the origin. The bone extends down through the length of the thigh to the kneecap (the next bone in the hierarchy), but rotations are governed by what takes place at the origin (the hip).

In this discussion we have changed nothing from the previous chapter; we have simply started referring to things using different names. Because the frame hierarchy is being used to represent a character, we are referring to it as a skeleton, with each frame as a bone and each frame matrix as a bone matrix. The bone matrix still describes the position and orientation of the body part mesh that is attached to that bone relative to the parent bone.

In the next section we will examine how the skinned mesh(es) will be stored in an X file. This will prove to be a very short discussion since most of the standard templates we have already covered in the previous chapter will be used again. While it is not strictly necessary for you to understand how the skinned mesh and skeletal structures are stored in an X file in order to be able to load and render them, examining how the data is stored (especially the way that the vertices are mapped to bones in the hierarchy along with weights) will help greatly in understanding how to store and render that data once it is loaded.

11.4 X File Skins and Skeletons

When a skinned mesh and a skeletal structure are exported to an X file, the skeletal structure is exported as a generic frame hierarchy, where each frame is a bone. The name of the frame will be the name that the artist assigned to the bone in the modeling package. This frame hierarchy is represented using the standard Frame template that we looked at earlier in the course. If we use our example of a pair of legs constructed from a skeletal structure consisting of five bones (a pelvis, two upper leg bones and two lower leg bones) we can see that the skeleton is represented in the X file as shown in the file snippet that follows.

Recall that each frame contains a transformation matrix (which we will now refer to as the parent-relative bone matrix) and any number of child frames. As we can see in this example, the pelvis frame is the root frame, which contains two immediate child frames -- the left upper leg bone and the right upper leg bone. Each one of these two child bones themselves have a child bone -- the left upper leg bone has the left lower leg bone as a child and the right upper leg bone has the right lower leg bone as a child. Notice that each bone in the skeletal structure that will be animated and have vertices assigned to it is given a name. In this example we have just set the matrix data of each bone to identity, but in a real world situation this would contain the position and rotation information of the bone.

```
Frame Pelvis    // Root Frame
{
    FrameTransformMatrix // Pelvis Bone Matrix
    {
        1.000000,0.000000,0.000000,0.000000,
        0.000000,1.000000,0.000000,0.000000,
        0.000000,0.000000,1.000000,0.000000,
        0.000000,0.000000,0.000000,1.000000;;
    }

    Frame LeftUpperLeg        // Child Bone of Pelvis
    {
        FrameTransformMatrix // LeftUpperLeg Bone Matrix
        {
            1.000000,0.000000,0.000000,0.000000,
            0.000000,1.000000,0.000000,0.000000,
            0.000000,0.000000,1.000000,0.000000,
            0.000000,0.000000,0.000000,1.000000;;
        }

        Frame LeftLowerLeg    // Child of LeftUpperLeg Bone
        {
            FrameTransformMatrix // LeftLowerLeg Bone Matrix
            {
                1.000000,0.000000,0.000000,0.000000,
                0.000000,1.000000,0.000000,0.000000,
                0.000000,0.000000,1.000000,0.000000,
                0.000000,0.000000,0.000000,1.000000;;
            }
        }
    }
}
```

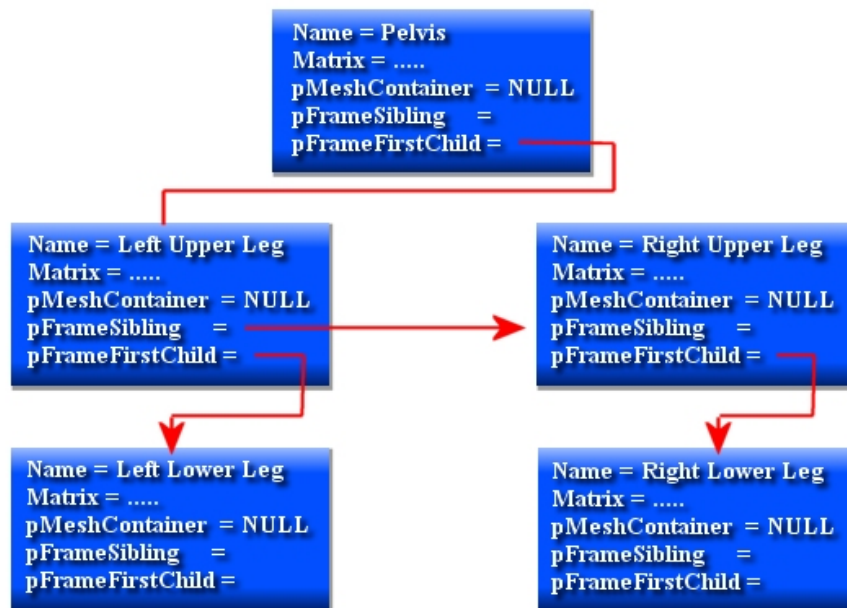
```

Frame RightUpperLeg          // Child Bone of Pelvis
{
    FrameTransformMatrix      // RightUpperLeg Bone Matrix
    {
        1.000000,0.000000,0.000000,0.000000,
        0.000000,1.000000,0.000000,0.000000,
        0.000000,0.000000,1.000000,0.000000,
        0.000000,0.000000,0.000000,1.000000;;
    }

    Frame RightLowerLeg        // Child Bone of RightUpperLeg
    {
        FrameTransformMatrix    // RightLowerLeg Bone Matrix
        {
            1.000000,0.000000,0.000000,0.000000,
            0.000000,1.000000,0.000000,0.000000,
            0.000000,0.000000,1.000000,0.000000,
            0.000000,0.000000,0.000000,1.000000;;
        }
    }
}
}

```

Forgetting about how the character skin is stored for now, we know that if we were to use the D3DXLoadMeshHierarchyFromX function to load this hierarchy it would create a D3DXFrame hierarchy for our skeleton as shown in Fig 11.12.



The D3DXFrame Hierarchy that would be created by the D3DXLoadMeshHierarchyFromX Function

Figure 11.12

Our application would be returned a pointer to the root frame (the pelvis bone). The pelvis bone would have its child pointer pointing at the LeftUpperLeg bone, which would be connected to the RightUpperLeg bone via its sibling pointer. So, both of these frames are considered to be child bones of the pelvis bone. The LeftUpperLeg bone would have its child pointer pointing at the LeftLowerLeg bone and the RightUpperLeg bone would have its child pointer pointing at the RightLowerLeg bone.

However, while the frame hierarchy representation in the X file is unchanged when representing bones, the big change in a skin and bones representation is that there is no longer a mesh attached to the various frames in the hierarchy. So, where exactly is this single skin (mesh) data stored?

As it turns out, there is no hard and fast rule about where the skin needs to be stored. What we do know is that it will no longer be assigned to a single frame in the hierarchy and that it will have the appropriate information contained inside the Mesh data object describing which vertices map to which bones. Therefore, the mesh might be embedded as a child mesh of the root frame or even some arbitrary child frame. Because the position of the mesh in the hierarchy no longer directly dictates which frames animate it (this will be described by a bone table inside the mesh object) or the order in which rendering takes place, the skin can pretty much be defined anywhere in the hierarchy (although it is generally defined in the root frame). When D3DXLoadMeshHierarchyFromX detects that the Mesh data object has 'skinning information' inside its data block, it knows to treat this mesh differently and supply our application with bone-to-vertex mapping information. Note that we no longer need to render the mesh hierarchically because it is a single mesh (although we still need to update the hierarchy matrices recursively). So if the Mesh data object is defined inside the root frame, then the root frame's pMeshContainer will point to the mesh container that stores the skin. This is the pointer we will use to access and render the subsets of the mesh.

You will be pleased to know that the definition of a skin mesh is no different from the regular Mesh X file definition. All of the same standard templates are used to define the Mesh data object itself as well as the child data objects such as the material list, the texture coordinates, and vertex normals. The only difference with a mesh that will be used for skinning is that it will contain additional child data objects. So we might imagine an arrangement as shown in the next example, where the Mesh object is defined as a child of the root frame. In this example we have not shown *all* of the mesh data and child objects, but you will get the general idea.

```
Frame Pelvis    // Root Frame
{
    FrameTransformMatrix  // Pelvis Bone Matrix
    {
        1.000000,0.000000,0.000000,0.000000,
        0.000000,1.000000,0.000000,0.000000,
        0.000000,0.000000,1.000000,0.000000,
        0.000000,0.000000,0.000000,1.000000;;
    }

    Mesh LegsSkin
    {
        4432;                                // Number Of Vertices
        -34.720058;-12.484819;48.088928;,    // Vertex List
        -25.565304;-9.924385;26.239328;,
        -34.612186;-1.674418;34.789925;,
```

```

    0.141491;7.622670;25.743210;;
    -34.612175;17.843525;39.827816;;
    -9.608727;27.597115;38.148296;;
    ... Remaining Vertex Data Goes Here...

    6841;                // Number of Faces
    3;28,62,1;; // Face List
    3;3,16,3420;;
    3;11,23,29;;
    3;104,69,7;;
    3;0,13,70;;
    3;9,97,96;
    ... Remaining Face Definitions Go Here ...

MeshNormals
{
    4432;
    -0.989571;-0.011953;-0.143551;;
    -0.433214;-0.193876;-0.880192;;
    -0.984781;0.061328;-0.162622;;
    -0.000005;0.123093;-0.992395;;
    ... Remaining Mesh Normals List Goes Here ....
}

.. Other child objects go here, Texture coordinates
   and a Mesh Materials List for example...

} // End mesh definition

Frame LeftUpperLeg           // Child Bone of Pelvis
{
    FrameTransformMatrix     // LeftUpperLeg Bone Matrix
    {
        1.000000,0.000000,0.000000,0.000000,
        0.000000,1.000000,0.000000,0.000000,
        0.000000,0.000000,1.000000,0.000000,
        0.000000,0.000000,0.000000,1.000000;;
    }

    Frame LeftLowerLeg       // Child of LeftUpperLeg Bone
    {
        FrameTransformMatrix // LeftLowerLeg Bone Matrix
        {
            1.000000,0.000000,0.000000,0.000000,
            0.000000,1.000000,0.000000,0.000000,
            0.000000,0.000000,1.000000,0.000000,
            0.000000,0.000000,0.000000,1.000000;;
        }
    }
}

Frame RightUpperLeg          // Child Bone of Pelvis
{
    FrameTransformMatrix      // RightUpperLeg Bone Matrix
    {

```

```

1.000000,0.000000,0.000000,0.000000,
0.000000,1.000000,0.000000,0.000000,
0.000000,0.000000,1.000000,0.000000,
0.000000,0.000000,0.000000,1.000000;;
}

Frame RightLowerLeg          // Child Bone of RightUpperLeg
{
    FrameTransformMatrix      // RightLowerLeg Bone Matrix
    {
        1.000000,0.000000,0.000000,0.000000,
        0.000000,1.000000,0.000000,0.000000,
        0.000000,0.000000,1.000000,0.000000,
        0.000000,0.000000,0.000000,1.000000;;
    }
}
}

```

Fig 11.13 shows how the loaded hierarchy and mesh container would be arranged in memory after the D3DXLoadMeshHierarchyFromX function has finished executing. There is a single mesh attached to the root frame although, as mentioned, the mesh could have been attached to another frame in the hierarchy. Your application will probably want to store this mesh pointer in some other more easily accessible application variable. Fetching from the root frame structure every time is also fine.

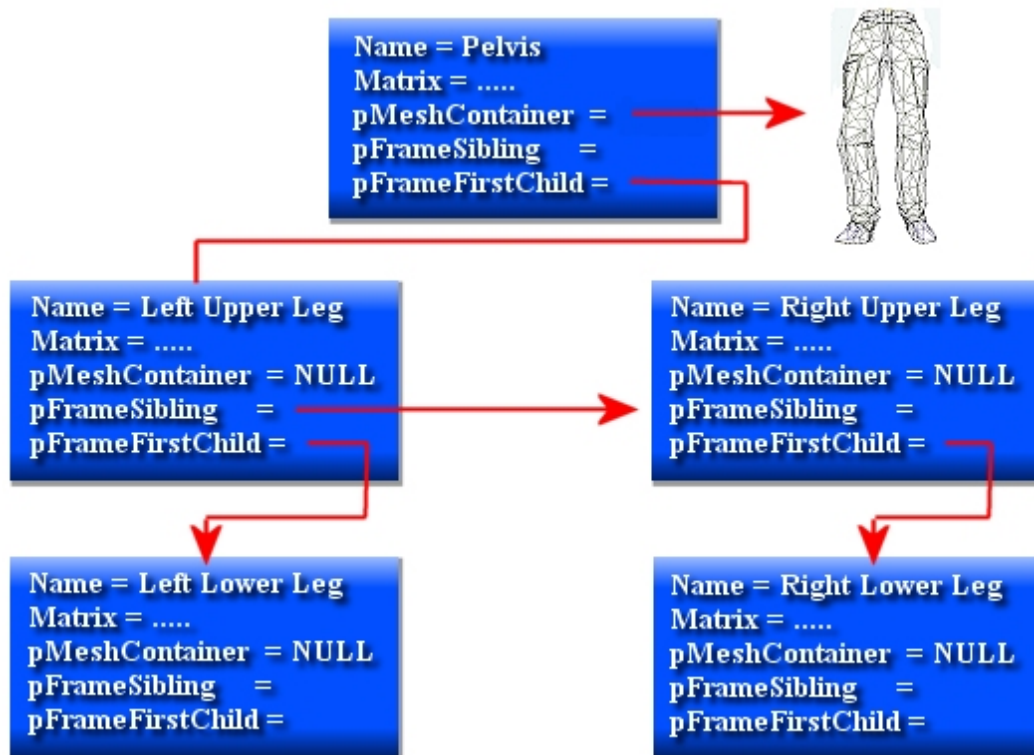


Figure 11.13

While things have not changed much from our previous chapters, we still need to address the bone/vertex relationship. As it happens, a skin mesh data object will be like a normal mesh data object with the exception that it will now have two additional types of child data objects embedded within its definition (not shown in Fig 11.13). These new object types are both standard X file templates. Together they will contain the information that describes which vertices a bone affects and by how much. The existence of these standard templates in the file is how D3DXLoadMeshHierarchyFromX identifies the mesh as a skin and thus takes additional action to load the bone-to-vertex mapping table. We will examine these templates in the next two sections.

11.4.1 The XSkinMeshHeader Template

The first standard template we will discuss is the XSkinMeshHeader template. There will be exactly one of these data objects defined as a child of the mesh data object. This template stores information about the maximum number of bone matrices that affect a single vertex in the mesh and the maximum number of bone matrices that are needed to render a given face. A ‘face’ is a triangle consisting of three vertices, so if (for example) each of a triangle’s vertices were influenced by four unique bone matrices, then we know that in order to transform and render this triangle we would need to have access to the $(3 * 4)$ 12 matrices that influence this single face.

It may not yet be obvious why we would need to know how many matrices collectively affect a face. After all, we are just transforming each vertex into world space, one at a time, using the matrices that affect that vertex. It would seem that all we would need to know is how many matrices affect the specific vertex being transformed. As it turns out, this is true when we perform software skinning (discussed later), where we basically just perform vertex blending one vertex at a time. However, if the hardware supports vertex blending, then things will change a bit. We learned early in this course series that when calling the DrawPrimitive family of functions, the pipeline transforms our vertices in triangle order (i.e. we pass in lists of triangles to be rendered and they are processed one at a time). Thus, in order for the pipeline to transform a triangle’s vertices using vertex blending, the device must have access to all of the matrices that influence all of the triangle vertices before we issue the DrawPrimitive call. All of this will make a lot more sense later on in this chapter when we start looking at API specific skinning.

Let us first have a look at the XSkinMeshHeader type. It is a very simple template with just three members. Again, there will be exactly one of these child data objects inside each skinned mesh data object.

```
template XSkinMeshHeader
{
    <3cf169ce-ff7c-44ab-93c0-f78f62d172e2>
    WORD nMaxSkinWeightsPerVertex;
    WORD nMaxSkinWeightsPerFace;
    WORD nBones;
}
```

WORD nMaxSkinWeightsPerVertex

This will describe the maximum number of bones (matrices) that influence a vertex in the mesh. For example, some vertices might only be influenced by a single bone, others by two. Perhaps there are also a few vertices that are affected by three bone matrices. In this example, this value would be set to 3 to let DirectX know that at most, a vertex in this mesh will only be influenced by three matrices. This is actually very useful information. We will see later that when we create a skinned mesh for hardware skinning, the vertex format used will also contain a number of floating point weight values per vertex. These weights describe the influence for each bone that affects it. Since all of the vertices in a mesh's vertex buffer will need to be the same format, we need to make sure we specify the correct FVF flags when creating the mesh. We want to ensure that enough storage space is allocated per-vertex for the maximum number of weights needed by any given vertex in the mesh.

WORD nMaxSkinWeights

This value will describe the maximum number of bones that influence a single triangle in the mesh. If a triangle in the mesh used four unique bones for *each of its vertices* but other triangles in the mesh used fewer, this value would still be set to $4 \times 3 = 12$. This will tell our application how many matrices we must make available (at most) to the device in order for a given triangle to be transformed by the pipeline using vertex blending. Not all triangles will need access to this many matrices to be transformed and rendered, but the system must be setup such that we can cope with those that do.

WORD nBones

This value describes how many bones (frames) in the hierarchy influence this skinned mesh. At first you might assume that all bones must affect the mesh, but that would not be correct. This is because the X file might contain multiple skeletal structures all linked in one big hierarchy and it might also contain multiple skins for those skeletons. In that case we could say that the hierarchy contains multiple skinned meshes where each mesh is mapped only to the bones for its own skeleton in the hierarchy. The X file might also contain an entire scene hierarchy of which a skinned mesh's bone hierarchy is just a subset. Suffice to say, all of the frames defined in the X file might not belong to a single skeletal structure or might not belong to the skeletal structure attached to a given skin. Quite often however, a skinned mesh and its skeletal structure will be stored in its own separate X file.

11.4.2 The SkinWeights Template

The SkinWeights template stores information about which vertices in the parent mesh data object are affected by a single bone in the hierarchy. It also stores the weights by which those vertices are influenced by the bone. Therefore, the nBones member of the XSkinMeshHeader data object describes how many SkinWeights child data objects will follow it in the mesh definition. If a skin's skeleton consists of five bones, there will be one XSkinMeshHeader child data object followed by five SkinWeights data objects (one SkinWeights child data object defined for each bone in the skeleton for this mesh).

```

template SkinWeights
{
    <6f0d123b-bad2-4167-a0d0-80224f25fabb>
    STRING transformNodeName;
    DWORD nWeights;
    array DWORD vertexIndices[nWeights];
    array FLOAT weights[nWeights];
    Matrix4x4 matrixOffset;
}

```

STRING transformNodeName

This member is a string that will store the name of the bone for which this information will apply. For example, if this string contained the name `RightUpperLeg`, then the data that follows (the vertices attached to this bone and their weights) will describe the vertices attached to the `RightUpperLeg` bone in the skeletal hierarchy.

DWORD nWeights

This value describes how many vertices in the skin will be influenced by this bone. As a result, it will tell us the size of the two arrays that follow this member which contain vertex indices and vertex weights for this bone.

array DWORD vertexIndices [nWeights]

Following the `nWeights` member is an array of vertex indices. The number of vertices in this array is given by the `nWeights` member. For example, if `nWeights = 5` and the `vertexIndices` array held the five values `{4, 7, 20, 45, 100}` then it means that the fourth, seventh, twentieth, forty-fifth and one hundredth vertex defined in the parent mesh's vertex list are mapped to this bone. Therefore, when these vertices are being transformed into world space, this bone's matrix should be used to some extent. We deduce then that if a vertex is influenced by n bones, then its index will be included in the `vertexIndices` array of n `SkinWeights` data objects.

array DWORD weights[nWeights]

We looked at some pseudo-code earlier in the chapter that performed vertex blending using multiple matrices. Recall that we multiplied the model space vertex with each bone matrix (one at a time) to create n temporary vertex positions (where n was the number of matrices being used in the blend). Each temporary vertex position was then scaled by the weight assigned to the matrix that transformed it, which scaled the contribution of the matrix. Recall as well that the combined weights are equal to 1.0 and that the sum of these scaled temporary vectors was the final vertex position in world space. So we know that it is not enough to define for each bone the vertices that are influenced by it; we must also define weights describing the strength of the bone's influence on each of those vertices. Therefore, following the `vertexIndices` array is another array of equal size defining the weight of this bone on the corresponding vertex in the `vertexIndices` array. Each of these weights will usually be a value between 0.0 and 1.0.

Matrix4x4 matrixOffset

The final member of the `SkinWeights` data object is the bone offset matrix. The bone offset matrix is often a cause for much confusion for students who are new to skeletal animation. We will examine the bone offset matrix in detail very shortly.

Before we finish our X file discussion, let us look at an example of how the skin/bone data is stored in the file. To save space, we will not show our leg hierarchy again, we will just concentrate on the mesh data object (which we already know will exist as a child of one of the frame data objects). We will also pare down the amount of mesh data and standard child objects shown so that we can concentrate on the skin information. We really just want to focus on how the XSkinHeader data object and the multiple SkinWeights data objects will be embedded as child data objects of the mesh. We will not include the SkinWeights data object for each bone (to save space) and instead will simply show two of them along with their vertex mappings. Since our skeleton actually consists of five bones, we know that in reality there would be five SkinWeights data objects defined for this mesh.

```
Mesh LegsSkin
{
    300;                // Number Of Vertices
    -34.720058;-12.484819;48.088928;;    // Vertex List
    -25.565304;-9.924385;26.239328;;
    -34.612186;-1.674418;34.789925;;
    0.141491;7.622670;25.743210;;
    -34.612175;17.843525;39.827816;;
    -9.608727;27.597115;38.148296;;
    ... Remaining Vertex Data Goes Here...

    200;                // Number of Faces
    3;28,62,1;;    // Face List
    3;3,16,3420;;
    3;11,23,29;;
    3;104,69,7;;
    3;0,13,70;;
    3;9,97,96;
    ... Remaining Face Definitions Go Here ...

    MeshNormals
    {
        4432;
        -0.989571;-0.011953;-0.143551;;
        -0.433214;-0.193876;-0.880192;;
        -0.984781;0.061328;-0.162622;;
        -0.000005;0.123093;-0.992395;;
        ... Remaining Mesh Normals List Goes Here ....
    }
    XSkinMeshHeader
    {
        2;        // Each vertex is assigned to no more than two bones
        4;        // The maximum bones used by a single triangle is 4
        5;        // This skin uses a five bone skeleton
    }

    SkinWeights        // Mapping information for the 'RightUpperLeg' bone.
    {
        "RightUpperLeg";
        2;                // Two vertices are mapped to this bone.
        150,              // Vertex 150 is one of them
        300;              // And vertex 300 is the other

        0.500000,         // Vertex 150 has a weight of 0.5 for this bone
```

```

        0.500000;          // Vertex 300 also has a weight of 0.5 for this

        //bone offset matrix for the RightUpperLeg bone
        1.253361,   -0.000002,   0.254069,   0.000000,
        -0.218659,   0.223923,   1.078679,   0.000000,
        0.058231,   -1.440720,   -0.287275,   0.000000,
        -8.131670,   62.204407,   -2.611076,   1.000000;;
    }

    SkinWeights    // Mapping information for the 'LeftUpperLeg' bone.
    {
        "LeftUpperLeg";
        3;          // Three vertices mapped to this bone
        1,          // Vertex 1 is mapped to this bone
        20,         // Vertex 2 is another
        25;         // Vertex 3 is another

        0.333333,   // Vertex 1 weight for this bone
        0.333333,   // Vertex 2 weight for this bone
        0.333333;   // Vertex 3 weight for this bone

        // Bone offset matrix for left upper leg bone
        1.253361,   -0.000002,   0.254069,   0.000000,
        -0.218659,   0.223923,   1.078679,   0.000000,
        0.058231,   -1.440720,   -0.287275,   0.000000,
        -8.131670,   62.204407,   -2.611076,   1.000000;;
    }

    ... Remaining SkinWeights objects defined here. There should be 5 in total, one for each bone...
} // End mesh definition

```

So we can see that the data is stored in the X file in a simple fashion. There will be a hierarchy of bones and a single mesh. That mesh will contain SkinWeights data objects (one for each bone) describing which vertices in the mesh are attached to each bone in the hierarchy, along with the weight describing the influence of that bone matrix on its attached vertices. Furthermore, each SkinWeights data object also contains a bone offset matrix for the bone being described. Before we talk about loading the hierarchy and configuring the API for skinned mesh rendering, let us first examine the bone offset matrix.

11.5 The Bone Offset Matrix

The bone offset matrix stores the inverse transformation of the bone's absolute (not relative) position and orientation in the hierarchy with the skeleton in its reference pose. Although bone offset matrices are provided for each bone in the X file, you could create your own bone offset matrix for each frame by traversing the hierarchy (arranged in its reference pose) starting at the root, and for each frame calculate the absolute matrix for that frame (remember that they are stored as relative matrices in the hierarchy initially). Once you have the absolute matrix for a given frame, just invert it and you will have the bone offset matrix for that bone. The following snippet of code should make this clear.

Note: The code sample that follows is for illustration only. This information is already contained in the X file and loaded automatically by the D3DXLoadMeshHierarchyFromX function.

In the next code example, we have derived our own structure from D3DXFrame just as we did in the last chapter. Recall that we added an additional member called the combined matrix (i.e. the world space matrix of the frame). We will do the same thing again this time, but also add a new member to store the bone offset matrix that we will generate.

Just as a reminder, the standard D3DXFRAME structure is defined as:

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

We will derive our own structure from it with one matrix to contain the absolute transform for each frame when we update the hierarchy and another to contain the bone offset matrix for the given bone.

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

The following function would then be called when the bone hierarchy is first constructed and is still in its reference pose (i.e. before any animations have been applied). *pRoot* is assumed to be the root bone returned to the application by D3DXLoadMeshHierarchyFromX. Thus it would be called as:

```
CalculateBoneOffsetMatrices( pRoot, NULL );
```

The function implementation is almost identical to the UpdateFrameHierarchy function we wrote in the previous chapters. It steps through the bone hierarchy using recursion, combining relative matrices along the way to calculate the absolute matrix for each frame. We can think of the combined matrix as being the absolute bone matrix. It describes the actual position of that bone in the representation. Once we have the bone matrix, we invert it to get the bone offset matrix. This offset matrix describes an inverse transformation that provides access to the local space of the bone (as it was defined in its reference pose). You should refer back to our discussion in Chapter Four about the relationship between local space and inverse transformations if this does not sound familiar to you.

```
void CalculateBoneOffsetMatrices (D3DXFrame *pFrame , D3DXMATRIX *pParentMatrix)
{
    D3DXFRAME_BONE * pMtxFrame = (D3DXFRAME_BONE*) pFrame;

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

```

else
    pMtxFrame->mtxCombined = pMtxFrame->TransformationMatrix;

    // Calculate the bone offset matrix for this frame
    D3DXMatrixInverse (pMtxBoneOffset, NULL, pMtxCombined);

    if(pMtxFrame->pFrameSibling)
        UpdateFrameMatrices( pMtxFrame->pFrameSibling, pParentMatrix );
    if(pMtxFrame->pFrameFirstChild)
        UpdateFrameMatrices(pMtxFrame->pFrameFirstChild,
                            &pMtxFrame->mtxCombined );
}

```

It is very important to note that while the bone matrices in the hierarchy change when animations are applied to them, this is not true of the bone offset matrices. They will not change throughout the life of the character (otherwise there would be little point storing the information in the X file). Using code similar to the above example, they would be calculated once when the mesh is first loaded and stored away for later use. Remember, we never need to calculate them as shown above; this code was provided as a means of describing how the bone matrices stored in the X file were initially calculated by the modeling application that created it.

11.5.1 The Necessity of the Bone Offset Matrix

Early on in this chapter we looked at some vertex blending code that showed a vertex in the skin being transformed from model space to world space using a blend of multiple world matrices. If we were to use this function to transform the vertices of our skin, the matrices we would use to transform a given vertex would be the absolute bone matrices calculated when we updated the frame hierarchy. We would certainly not apply the relative transformations because we wish to render the vertex and thus require that it exist in world space. So before we transform the skin vertices from model space to world space, we would first need to apply any relevant animations to the frame hierarchy (using the `ID3DXAnimationController::AdvanceTime`) method. These animations result in a new set of parent-relative frame matrices being stored (as described in the previous chapter). Then we would traverse the hierarchy and concatenate all of the appropriate matrices together (as we did in the previous chapter also) to update the absolute matrices stored in each frame. These combined matrices in each bone store the world space transform for the bone and thus should be used to transform any attached vertices during vertex blending.

Without the bone offset matrix, this final transformation presents a problem. Imagine for the sake of simplicity that we are transforming a single vertex in the skin from model space to world space. Also imagine that the vertex is only attached to one bone (e.g., `RightUpperLeg`) and that the vertex has a weight of 1.0 stored for that bone. We might assume that we could transform the vertex into world space by doing the following (given that `mtxCombined` has just been updated to contain the correct position and orientation of the bone in world space):

World Space Vertex = Model Space Vertex * `RightUpperLeg->mtxCombined`.

This looks like it should work, until we remember that the vertex is defined in model space. That is, the vertex is defined relative to the origin of the model space coordinate system instead of being defined relative to the bone via which it is being transformed. Why is this necessary? Well let us think for a moment about what we would like to have happen. If an animation is applied to a bone, we want that motion to propagate out to all of the skin vertices that are attached to that bone so that they move along with it. Where have we seen this sort of behavior before? In Chapter Ten, of course. When we animated a given frame in our hierarchy, that animation was propagated down to all of its children and they experienced the same animation themselves. This is exactly what we want to have happen in our bone/vertex case. But recall that our frame hierarchy is set up in a relative fashion. That is, each frame is defined in the local space of its parent. Unfortunately this is not the case for our vertices. We know that the vertices of our skin are defined relative to the origin of the model (their ‘parent’, if you will). They exist in model space. So if we want them to move along with the bone, they will need to be defined relative to the bone (i.e. exist in bone local space). Or more to the point, we want to attach our vertices to the hierarchy such that they behave as children of the bone.

So then why not just store the vertex as a position relative to the bone instead of the model space origin? That is not possible either because the vertex may need to be transformed by multiple bones, and in each case the vertex would need to be defined relative to that bone. That is clearly not going to work unless we wish to store multiple copies of our vertices (which is problematic and obviously not very appealing).

Just for clarity, let us see what happens anyway if we simply transform the vertex by the bone matrix. To simplify this example we will forget about orientation for now and just concentrate on the position of a bone in its reference pose.

Let us imagine that we have a bone that has a combined matrix such that, in its reference pose it is at position $\mathbf{b}(50,50,50)$. Let us also assume that we have a vertex that will be influenced by this bone and it is positioned at position $\mathbf{v}(60,60,60)$ with respect to the model space origin (in its reference pose). We can see right away that the vertex is offset (10,10,10) from the bone when both the bone hierarchy and the skin are in the reference pose. So what we are going to do next is simply assume that the skeleton is still in its reference pose and transform our vertex by our bone matrix.

We know for starters that the vertex should be at position (60,60,60) in its default pose. So if we used the above transformation approach, this is where the vertex should be after being multiplied by the bone matrix (since the bone has also not moved from the default pose). You might see already that this is clearly not going to work. If we think about the case of simply transforming and rendering the mesh in its reference pose, the vertex is already in the correct position (with respect to the bone) even before we transform it using the bone matrix. If we were to transform vertex \mathbf{v} by bone \mathbf{b} , we know that with the bone at (50,50,50) the vertex should be at (60,60,60). But look at what happens next:

$$\begin{aligned}\textit{World Space Vertex} &= \mathbf{v} * \mathbf{b} \\ &= (60, 60, 60) + (50, 50, 50) \\ &= (110, 110, 110)\end{aligned}$$

Note: We are dealing with only the translation portion of the bone matrix to make the example easier. We are assuming that the bone is not rotated in its default pose. Therefore $\mathbf{v} * \mathbf{b} = \mathbf{v} + \mathbf{b}$.

Of course, if the vertex was originally defined in bone space instead of model space then the position of the vertex would have been stored as $v(10,10,10)$. This vertex position which is defined relative to the position of its influential bone would indeed create a transformed vertex position of $(60,60,60)$ as expected.

So given that our vertices are not defined in bone space, but in model space, what we need to do before we transform any vertex by a given bone matrix is first transform the vertex into that bone's local space. This way, the vertex is defined in bone space *prior to* being transformed by the bone matrix. Using our simple example again, this would mean performing an inverse translation of the bone's position for the vertex to convert it from a model space vertex position to a bone space vertex position.

First we transform model space vertex \mathbf{v} into the space of bone \mathbf{b} by subtracting the bone's offset (i.e. position) from the model space vertex.

$$\begin{aligned}\textbf{Bone Space Vertex} &= \mathbf{v} * -\mathbf{b} = (60, 60, 60) + (-50, -50, -50) \\ &= (10, 10, 10)\end{aligned}$$

We now have the vertex in bone space, so we can just apply the combined bone matrix to transform the bone space vertex into world space:

$$\begin{aligned}\textbf{World Space Vertex} &= \textbf{Bone Space Vertex} * \mathbf{b} = (10, 10, 10) + (50, 50, 50) \\ &= (60, 60, 60)\end{aligned}$$

Now we can see that the vertex has been transformed correctly by the matrix into world space. This was made possible by first transforming (in this case translating) the vertex into bone space and then transforming it into world space. We can think of the transformation into bone space as attaching the vertex to the bone as a child. The vertex becomes defined as bone-relative, just as the frames in a hierarchy are defined relative to their own parents. Thus if the parent moves or rotates, so do the children. This is exactly the behavior we want from our vertex-bone relationship. Remember that in our X file, the skin mesh was not actually defined as a child in the local space of any frame (which was why it would have been fine to store it in any frame node – it was just for storage purposes). But in our last chapter of course, our non-skin scene mesh vertices were defined relative to the frames that they were in. So what we are doing with this transformation of the vertex to bone local space is an extra step to get us back to where we were in our last demo – where mesh vertices lived in frame local space.

Again, as we discussed in Chapter Four, to get object A into the local space of object B, assuming both objects currently exist in the same space (i.e. same frame of reference) we need to transform A by the inverse matrix of B. Note that in the case of our reference pose mesh, both the bone frame (once it has been concatenated) and the vertex are both defined in model space. Remember that the reference pose mesh has not been moved out into the world. That is, the root node frame matrix is identity.

If this all makes perfect sense to you, then feel free to skip the next section. However, if you still need a bit more insight into how this all works, please read on. We will go through this process a bit at a time and use some diagrams to try to make everything as clear as possible.

11.5.2 The Bone Offset Matrix: A Closer Examination

The mathematics of 3D transformations and all of the different ‘spaces’ can sometimes get a little confusing for new students. After all, just look at our last example of the vertex/bone transformation from the previous section:

$$\begin{aligned}\text{Bone Space Vertex} &= \mathbf{v} * -\mathbf{b} = (60, 60, 60) + (-50, -50, -50) \\ &= (10, 10, 10)\end{aligned}$$

$$\begin{aligned}\text{World Space Vertex} &= \text{Bone Space Vertex} * \mathbf{b} = (10, 10, 10) + (50, 50, 50) \\ &= (60, 60, 60)\end{aligned}$$

It looks like all we have done is subtracted the bone’s position from the vertex and then added it right back on again! But when you really think about it, it starts to make some sense. Since we were transforming the vertex using the skeleton in its reference pose, we expect that the resulting vertex we get back from the transformation is the vertex position we already have stored. After all, nothing has actually moved, so the relationships should be the same as they were before the transformation.

So let us go one step further. This time let us assume that our animation controller has assigned some transformation to bone **b** translating it from position (50,50,50) to (100,100,100) when all is said and done. We certainly want the vertex to correctly move when the bone that influences it moves. The system above does exactly that because we first subtract the bone’s reference position from the model space vertex to transform the vertex into a bone local space position (i.e. the vertex is defined relative to the bone origin). We then use the newly modified combined bone matrix to transform the vertex into world space, thus maintaining the relationship between the vertex and the bone.

In the following formula, **b** is the bone’s reference position (how it is stored in the X file’s bone offset matrix) which never changes, **v** is the model space vertex, which also never changes and **c** is the current world transform (the combined matrix) of the bone that changes when animations are applied or when the character is moved about the world.

First we transform the vertex into bone space by subtracting the inverse *reference* position of the bone from the model space vertex.

$$\text{Bone Space Vertex} = \mathbf{v} * -\mathbf{b} = (60, 60, 60) + (-50, -50, -50) = (10, 10, 10)$$

Now we transform the bone space vertex using the *current* world transform of the bone so that the vertex is correctly positioned in the world. In this example we are assuming the current world space position of the bone is (100,100,100) after animation and hierarchy matrix concatenation has occurred.

$$\begin{aligned}\text{World Space Vertex} &= \text{Bone Space Vertex} * \mathbf{c} = (10, 10, 10) + (100, 100, 100) \\ \text{World Space Vertex} &= (110, 110, 110)\end{aligned}$$

As we can clearly see, the vertex has moved along with the bone. Although the bone has moved to position (100,100,100) the vertex has still maintained its relationship of a (10,10,10) offset from the bone, which placed it at world space position (110, 110, 110).

Now in these simple examples, we are assuming that the vertex is influenced by only one matrix. If the vertex is influenced by multiple matrices then the above steps would need to be performed for each matrix. The resulting temporary world space vertex positions would each be scaled by their respective weights before being added together to create the final world space vertex position, just as we saw earlier in our vertex blending code. It is also worthy of note that in the above example we are only showing the translation component of the process; the bone can also be rotated. So the fact that we have access to the bone offset matrix means that we always have the ability to transform the vertex back into bone space for a given bone and then apply the current world matrix of the bone. If the bone has been rotated, then the vertex will also be rotated about the bone space origin as expected rather than the model space origin.

So now we know that the X file will contain a bone offset matrix for every bone. We also know that when this matrix is applied to a model space vertex, it will transform the vertex from model space to bone space. This process essentially attaches the vertex to our bone as a child, just as we saw with our meshes in the last two chapters, which solves all of our problems. Remember that the bone offset matrices never change, while the world space bone positions do (when animated). With this in mind, we will rewrite our vertex blending function discussed earlier to first transform the vertex into each bone's space before applying the world transform for that bone.

Note: You will not have to write your own vertex blending function when you are using the DirectX pipeline. You will just need to make sure that it has access to both the world matrix and the bone offset matrix for each bone for the proper vertex transformations.

Our function parameter list has now been modified from our earlier example to include a pointer to an array of bone offset matrices. We pass in each world space (combined) bone matrix in the pBoneMatrices array and we will also pass in each bone's corresponding bone offset matrix (loaded from the X file) in the pOffsetMatrices array. Notice that in this function, rather than applying the inverse transform matrix to the vertex and then applying the world bone matrix to the bone space vertex in two separate steps, we can combine the bone's world matrix with its offset matrix first to get a resulting matrix that will perform both steps with a single vertex/matrix multiply. So we will transform the vertex from model space straight into world space with this one combined matrix.

```
D3DXVECTOR3 TransformBlended ( D3DXVECTOR3* pModelVertex ,
                               D3DXMATRIX * pBoneMatrices ,
                               D3DXMATRIX *pOffsetMatrices ,
                               float *pWeights , float NumOfMatrices )
{
    D3DXVECTOR3 TmpVertex;
    D3DXVECTOR3 FinalVertex ( 0.0f , 0.0f , 0.0f );
    D3DXMATRIX mtzTempMatrix;

    for ( int I = 0 ; I < NumOfMatrices; I++ )
    {
        D3DXMatrixMultiply ( &mtzTempMatrix , pOffsetMatrices[I],
                             pBoneMatrices[I]);
```

```

    TmpVertex = ( *pModelVertex * mtxTempMatrix ) * pWeights[I];
    FinalVertex += TmpVertex;
}

return FinalVertex;
}

```

Remember that the bone offset matrices never need to be recalculated. They are loaded (or created) during initialization and are re-used during each iteration of the game loop. When combined with the bone's world transform, we get a matrix that can correctly be used in blending operations to transform any vertices that are influenced by that bone.

As promised, before moving off the topic of the bone offset matrix, we will look at some additional examples with the aid of some diagrams to make sure that you fully understand the concept of the bone offset matrix.

In the following example we will see why the bone offset matrix is needed to transform a vertex that is influenced by two different bones in the hierarchy.

In Fig 11.14 we see two bones that influence a single vertex. For simplicity, we are working with 2D vectors. The absolute position of bone 1 in the skeleton reference pose is (5, 5) and the absolute position of bone 2 in the reference pose is (3, 8). The model space vertex in the reference pose is positioned at (5, 8). Although the bone matrices are initially stored as relative matrices in the X file, we are assuming in this example that the hierarchy has already been traversed and that the absolute position of each bone in the reference pose has been determined. We could say at this point that both of the bones and the vertex are now defined in the model space of the skin (assuming we have not moved the skeleton by updating the position of the root bone).

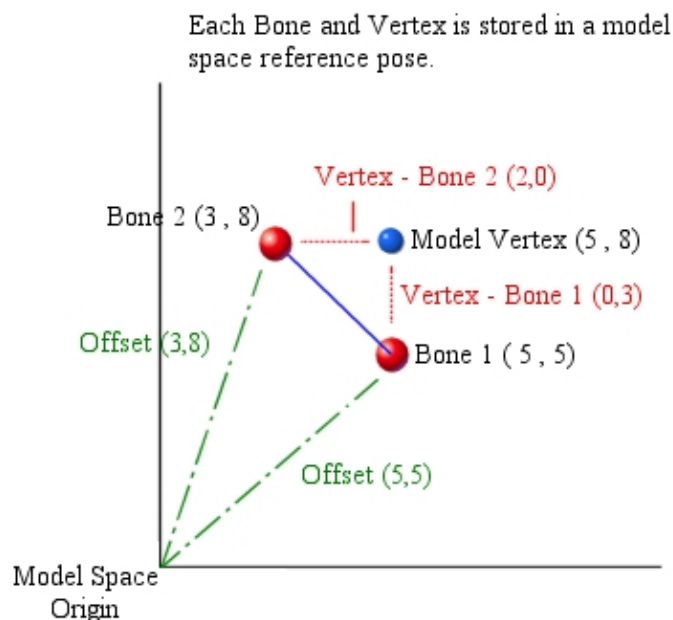


Figure 11.14

We note that the vertex is offset (0, 3) from bone 1 and (2, 0) from bone 2. We will assume that the weight of influence of each (bone) matrix on this vertex will be 0.5. That is, they will both be used in equal measure to transform the vertex into world space.

The two green lines in the diagram (labeled ‘Offset’) tell us how far from the model space origin each bone is offset. We can think of the bone offset matrix as being the reverse of these two lines. In other words, for bone 1, the offset matrix would have a translation vector of (-5, -5) and for bone 2 the bone offset matrix would have a translation vector of (-3, -8).

We know that in order to transform this model space vertex into world space we must transform it by each bone matrix, scale the resulting vectors by the respective weights, and then sum the results. If you recall our vertex blending function, you will remember that this is done one transformation at a time. We would first need to transform the vertex by bone 1 and then scale the result by bone 1’s weight to create a temporary world space vertex for that matrix. We would then do the same for bone 2 and combine the results. So let us start our example with how the vertex would be transformed by bone 1.

For this example, we will discuss how to transform the vertex into world space *after* the two bones to which the vertex is attached are moved into a new world space positions.

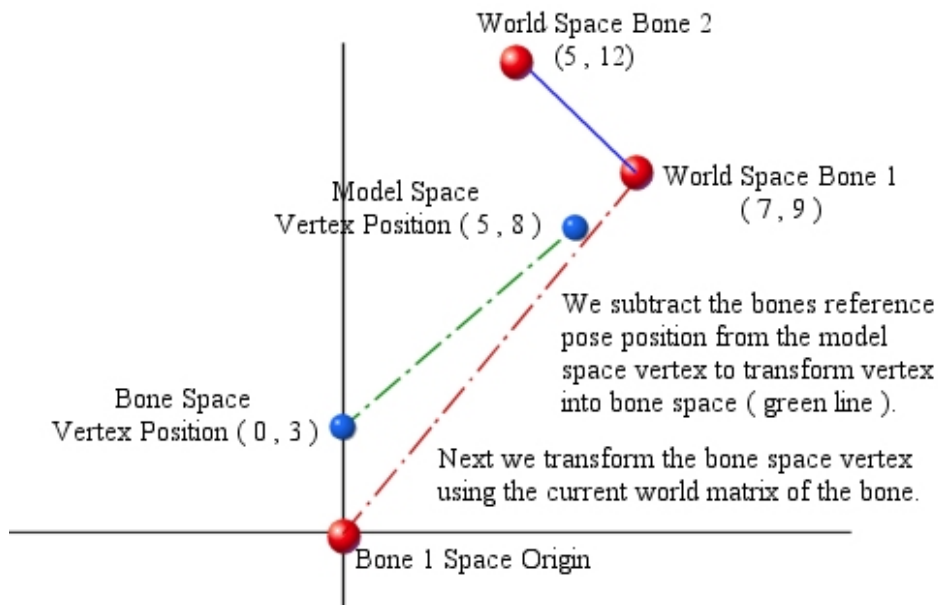


Figure 11.15

In Fig 10.15, both bone 1 and bone 2 are moved into world space positions (7, 9) and (5, 12) respectively. Beginning with bone 1 we apply the bone offset matrix to the model space vertex. This slides the blue vertex in the diagram from its model space position of (5, 8) into a bone 1 space position of (0, 3). As you can see, we have slid the vertex back along bone 1’s offset vector. In bone 1 space we can imagine that bone 1 is now at the origin of the coordinate system and that the vertex is offset (0, 3) from the bone, as was the case in the reference pose.

Since the world transform for bone 1 is now (7, 9), we apply this transformation to the bone space vertex (0, 3) as shown in Fig 11.16.

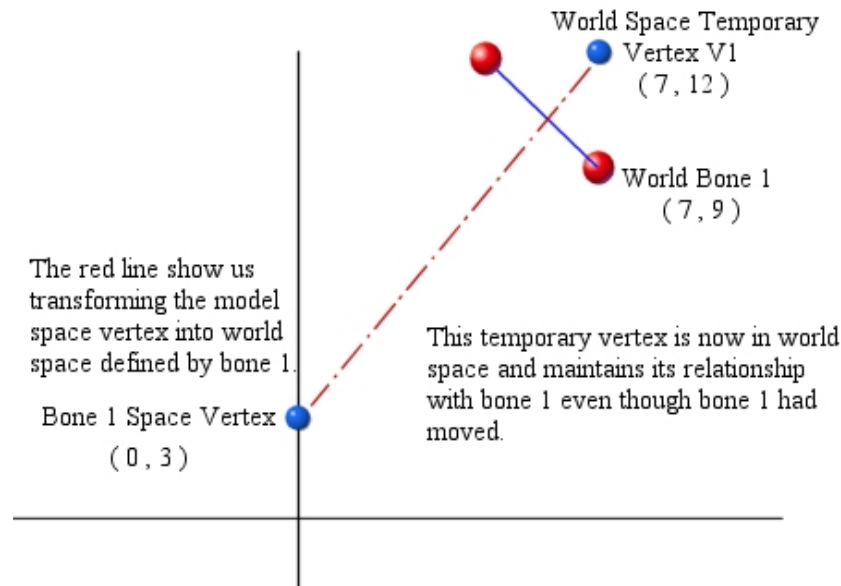


Figure 11.16

We can see in Fig 11.16 that we have now slid the vertex into position (7, 12) and it remains correctly offset (0, 3) from bone 1. So the relationship between the bone and the vertex in the reference pose has been correctly maintained even with the bone in its new world space position.

The next step is to scale the temporary world space vertex by the weight of bone 1 (0.5) for that vertex. This will effectively halve the length of the vector. In Fig 11.17 we can see that the final position of the temporary vertex, when multiplied by the weight of 0.5 (3.5, 6) is halfway along the red dashed diagonal line.

We now repeat the process all over again for bone 2. First we subtract the offset of bone 2 from the model space vector to get the vertex into bone space. We then multiply this vertex by the world matrix of bone 2 to create another temporary vertex of (7, 12) which is scaled by the weight of bone 2 (also 0.5) and we wind up with a temporary vertex position of (3.5, 6). Because both matrices influence the vertex equally, this means that we end up with two temporary vertex positions of (3.5, 6). When these two temporary vectors are added together we have the final world space position of the vertex (7, 12). Obviously, if the weights of the matrices for this vertex were altered so that one bone influenced the vertex more than the other and/or if the vertex was not positioned equidistant from each bone, the final world position of the vertex would be different.

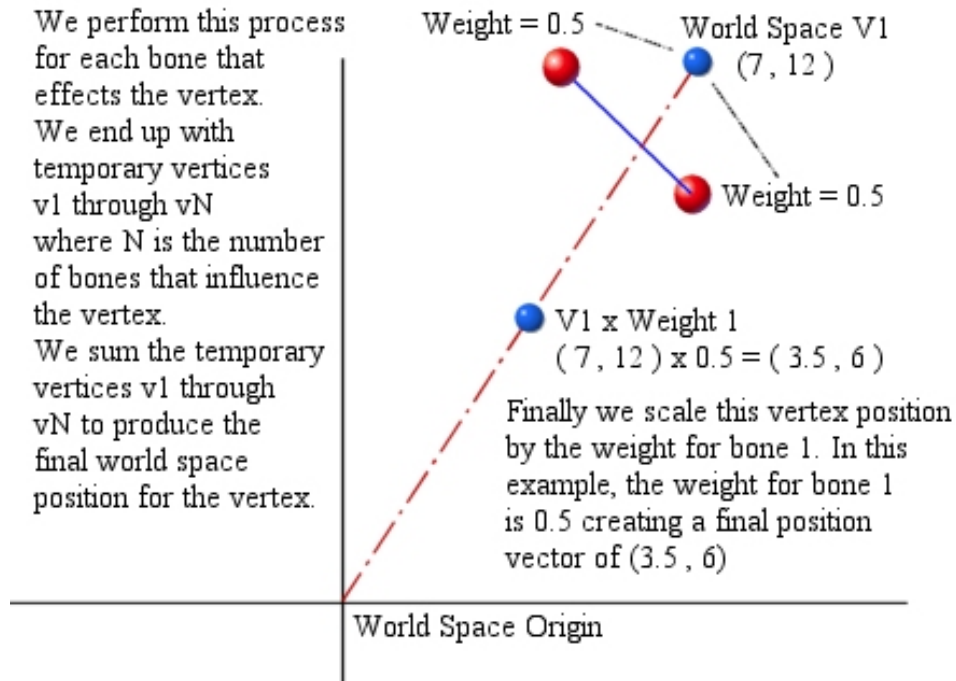


Figure 11.17

Bear in mind that we are showing a simple example here based on only translations. If any of the bones world matrices contained rotations, then the vertex would be rotated about the bone space origin before being translated into position.

Let us now tackle a slightly more complex example. In Fig 11.18 we see a model space vertex that has a position of (30, 20, 0) and is influenced by three bones. The diagram shows the position of each of the three bones in the reference pose and also shows the influence of each bone on the vertex. As you can see, the weights for this vertex are 0.25, 0.5, and 0.25 for bones 1 through 3, respectively. Therefore, we could say that bone 1 and bone 3 contribute 25% of the vertex world space position each, while bone 2 will contribute half with its influence at 50%. Once again, we are looking at the absolute bone positions for the reference pose calculated by traversing the hierarchy and combining the relative matrices stored therein.

Vertex Weights

Bone 1 = 0.25

Bone 2 = 0.5

Bone 3 = 0.25

Model Space = 30,20,0

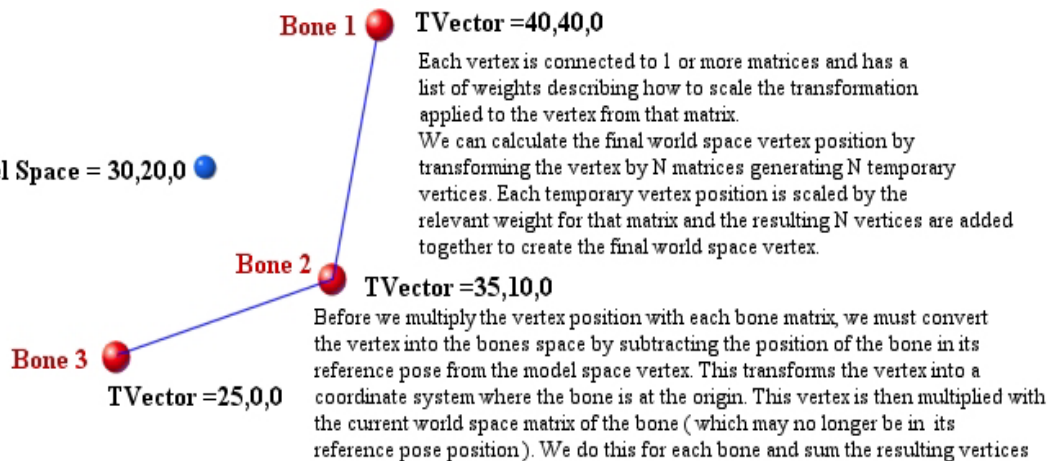


Fig 11.18

If we imagine that these three bones define the arm of a skeleton which is bent in its default pose, let us see what happens to the model space vertex when we move bone 3 to straighten the arm (Fig 11.19).

Vertex Weights

Bone 1 = 0.25

Bone 2 = 0.5

Bone 3 = 0.25

Model Space = 30,20,0

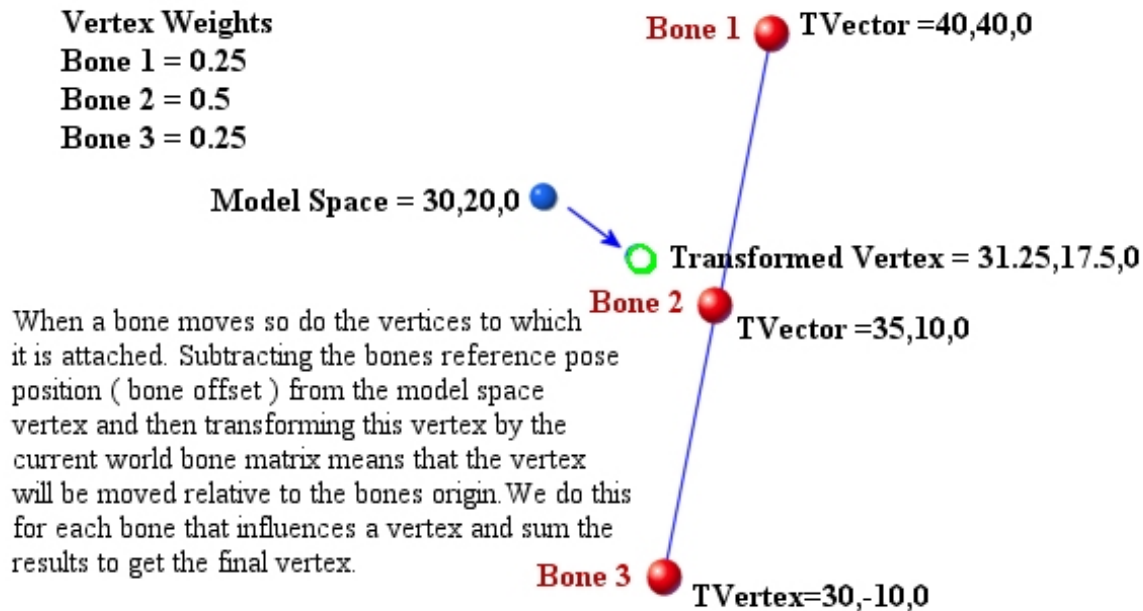


Fig 11.19

We already know that in order to calculate the final position of the vertex we must calculate (for each bone) a temporary vertex position by subtracting the bone's offset in the reference pose from the model space vertex. (Do not forget that technically this step is a bone offset matrix multiplication that accounts for rotation as well. We are focusing on translation only because the math is easier to grasp.) Then we have to transform the bone space vertex using the bone's combined world matrix and scale the result using the weight value. This will be done in three separate steps (once for each bone matrix) before summing the three temporary vertex positions to create the final world space vertex position.

The following calculations walk through the process. They demonstrate exactly what our vertex blending function would do if given the model space vertex, the three bone matrices, and their respective bone offset matrices from Figure 11.19.

Notice that for each of the three bones we transform the model space vertex into bone space using the current bone's inverse transformation (the bone offset matrix). We then transform the bone space vertex position with the current bone transformation and scale the resulting vector by the weight for that bone. Notice that in Fig 11.19, only bone 3 has been moved. Therefore, we expect that the transforms for bones 1 and 2 should just output the same model space vertex because the relationship between the vertex and those bones are the same as they were in the reference pose. However, when transforming the vertex by bone 3 (which has moved), the bone position is no longer equal to the reverse of the transformation described by the bone's offset matrix. Therefore, the vertex position in step three will not be the same model space vertex, and thus the sum of these vertices produces a different world space vertex position as well. The new world space vertex position is shown in Fig 11.19 as a green circle.

Transform Vertex V with Bone 1

$$\begin{aligned}
 \text{Bone Space } V &= (30, 20, 0) + (-40, -40, 0) = (-10, -20, 0) \\
 \text{Un-Scaled World } V1 &= (-10, -20, 0) + (40, 40, 0) = (30, 20, 0) \\
 \text{Scaled World } V1 &= (30, 20, 0) * 0.25 = (7.5, 5, 0)
 \end{aligned}$$

Transform Vertex V with Bone 2

$$\begin{aligned}
 \text{Bone Space } V &= (30, 20, 0) + (-35, -10, 0) = (-5, 10, 0) \\
 \text{Un-Scaled World } V2 &= (-5, 10, 0) + (35, 10, 0) = (30, 20, 0) \\
 \text{Scaled World } V2 &= (30, 20, 0) * 0.5 = (15, 10, 0)
 \end{aligned}$$

Transform Vertex V with Bone 3

$$\begin{aligned}
 \text{Bone Space } V &= (30, 20, 0) + (-25, 0, 0) = (5, 20, 0) \\
 \text{Un-Scaled World } V3 &= (5, 20, 0) + (30, -10, 0) = (35, 10, 0) \\
 \text{Scaled World } V3 &= (35, 10, 0) * 0.25 = (8.75, 2.5, 0)
 \end{aligned}$$

Sum the 3 temporary vectors to create the final vertex position

$$\begin{aligned}
 \text{Final World Space Vertex} &= \text{Scaled } V1 + \text{Scaled } V2 + \text{Scaled } V3 \\
 &= (7.5, 5, 0) \\
 &+ (15, 10, 0) \\
 &+ (8.75, 2.5, 0) \\
 &= (31.25, 17.5, 0)
 \end{aligned}$$

Hopefully you now understand the purpose of the bone offset matrix. At its most basic level, it is used to temporarily attach the model space vertex to the bone hierarchy such that the vertex exists in bone local space before it is transformed into world space. This allows animation of the hierarchy to filter down to the level of each individual vertex in the skin because the vertex becomes a temporary child of the bone. Thus any movement of the bone will move the vertex, just as we saw in our last chapter when we noted that the transformation of a parent node affected the children of that node.

11.6 D3DXLoadMeshHierarchyFromX Revisited

We now know how all of the data for our skin and bone entities are stored in the X file. The next logical step then is to take a look at how we can load all of this information into our application. In this chapter we will use `D3DXLoadMeshHierarchyFromX`, but if for some reason you do not want to use this function, you should be in good position, given the information covered in this chapter and the last, to write code that parses the X file manually.

Because a skinned mesh is really just a mesh that has its vertices attached to various frames/bones in the hierarchy, we are not limited to storing only a single skinned mesh or skeleton per X file. It is entirely possible that the X file might contain a vast number of frames representing an entire scene, and embedded in that hierarchy might be several branches of the frame hierarchy that represent skeletal structures for several skinned meshes contained within that scene. After all, as far as the frame hierarchy and the animation controller are concerned, bones are just frames. The fact that a mesh may be contained within the file that uses some section of the frame hierarchy as a skeleton is insignificant in the grand scheme of things as far as the loading function is concerned. Of course, it is not insignificant to us since we would like to be able to separate the components in such a file according to the logical design requirements of our game engine. So when we load the various skinned meshes from the file, we will need to be able to identify the frames in the hierarchy that are being used as the mesh skeletons and make sure that we set everything up so that the appropriate meshes are associated with their skeletons and that their vertices are properly deformed during the frame animation step.

The `D3DXLoadMeshHierarchyFromX` function was covered in detail in the last chapter, but here is its definition again for convenience:

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

We know already how this function works; it returns the root frame in the hierarchy, which may be the root bone of a single skinned mesh skeleton or, if the file contains multiple skinned meshes or an entire scene, might just be the root frame of the scene itself and not mapped to any mesh vertices contained within. Either way, this is our handle to the entire scene and our means to traverse the frames/bones being used.

It is quite common (although not required) for each skinned mesh to be stored in a separate X file so that each call to `D3DXLoadMeshHierarchyFromX` creates a new hierarchy for each skinned mesh, with its associated animation data and its own animation controller. When this is the case, each frame hierarchy will contain the skeletal structure for a single mesh and its accompanying animation data. This is

generally the cleanest way to deal with skinned mesh characters and will be the method we prefer in this course.

Recall that if any animation data is stored in the X file, then an interface to a `D3DXAnimationController` object will also be returned. This will allow us to play back those animations to update the bones of our skeleton (and consequently, the skin). As discussed, the `pAlloc` parameter is only used when parsing custom data objects in the X file. Since we will not be requiring custom data in this chapter, we can assume that this value will be set to `NULL`.

So one of our first questions needs to be, how do we know when the loading function detects a skinned mesh in the file? A follow up question is, once identified, how do we setup and store that data for later use? The answers, as you may have already guessed, are to be found in the implementation of our `ID3DXAllocateHierarchy` derived class, and in particular our implementation of its `CreateMeshContainer` function. You should recall from the previous chapter that this function is called by the loading function whenever a mesh is encountered in the file. The mesh data is passed to this function allowing us to store it in the mesh container in the hierarchy as well as make any optimizations to the mesh data before we do so. It is in this function that we are informed that the mesh data we are being passed is intended to be used as a skin. We will be passed additional information about the mapping between the vertices of the mesh and the bones in the hierarchy in the form of an interface to an `ID3DXSkinInfo` object.

The `ID3DXAllocateHierarchy::CreateMeshContainer` definition is shown next for our convenience. It should already be quite familiar to you since we covered this function in detail in the previous chapter.

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

This function is called for each mesh found in the X file. Our derived implementation is responsible for allocating a new `D3DXMESHCONTAINER` structure, taking the data passed into the function and storing it in this new mesh container, and then returning the newly populated mesh container back to the loading function. From there it will be attached to the relevant frame in the hierarchy. In Chapter Ten we saw that this was a relatively trivial task, but it was important because it allowed our application to own the memory for the meshes in the hierarchy. However, you might also remember that we ignored the `LPD3DXSKININFO` parameter – this will no longer be the case.

The `pSkinInfo` parameter is set to `NULL` when we are passed a normal (non-skin) mesh. For normal meshes, we simply store them in the mesh container and return as we did previously. If however, the `pSkinInfo` parameter is not `NULL`, then it means that the mesh we have been passed is to be used as a skin and it should have its vertices mapped to different bones in the hierarchy. When this is the case, we

have a fair bit of work to do. We have to make sure that we setup our mesh container so that we know which matrices have to be used to render a given subset in the mesh.

As it turns out, setting up a skinned mesh differs quite significantly depending on the type of skinning we are performing and whether or not hardware support is available. Since the different techniques can sometimes confuse matters when discussed collectively, what we will do in this chapter is talk about each skinning technique in isolation. On a case by case basis we will examine the skinned mesh creation process, the data structures needed to store it, and how to animate and render the skinned mesh.

In our lab project code we will implement all of the skinning techniques using a single class, but for now, we will consider each one separately for a more constructive analysis. This will make implementing any of the (three) methods we will study quick and easy. We will begin with the most basic skinning method -- software skinning. Essentially, this is a technique where the vertex blending stage is done in software (on the CPU) and not by the DirectX transformation pipeline on the GPU.

While examining each skinning technique we will look at how the `CreateMeshContainer` method should be written to correctly generate the skinned mesh for use with that technique.

11.7 DirectX Skinning I: Software Skinning

Software skinning is the first skinning technique that we will study in this chapter. As the name implies, it is not dependant on hardware support for vertex blending. As a result, it is not bound by the same limitations as hardware skinning, although it is generally not as fast. For example, when we use a hardware skinning technique, we are often limited to only a few bones being allowed to influence a single vertex. In the software case however we are not limited at all -- we can calculate our world space vertex positions using as many bone matrices as we desire. Therefore, while hardware skinning is preferable in most situations given the performance advantage, if you ever need to perform very complex skinning which requires large numbers of influential bones per vertex (more than would be supported by the DirectX transformation pipeline in hardware), software skinning would be your solution.

Note: Even though software skinning is not as fast as hardware skinning, it still performs very respectably.

When we use software skinning, the skin vertex data will be stored in a standard `ID3DXMesh`. There is nothing special or unique about this mesh -- it will be passed to our `CreateMeshContainer` function and stored in our mesh container just as we saw in the previous chapter. During iteration of our game loop, the frame hierarchy is animated and we traverse it to build our absolute bone matrices. Our next task is to take the model space mesh vertex data we have stored in our mesh container and transform it into world space. We certainly do not want to overwrite the skinned mesh vertex data because this is the model space reference pose of the skin that we will need to transform against each time. Thus, an additional mesh will be needed to store the transformed results. We call this the *destination mesh*.

To render the skinned mesh, we will take the vertices in the original (*source*) mesh and transform them into world space (using multi-matrix blending) and store the resulting world space vertices in the vertex

buffer of the destination mesh. It is this destination mesh that we will render each frame. It is worth noting that because this destination mesh has its vertices already stored in world space, we should set the world matrix of the device to an identity matrix before we render the destination mesh.

So how do we transform the mesh from model space to world space when performing software skinning? Well, when a skinned mesh is found in the X file, our CreateMeshContainer function will be passed an ID3DXSkinInfo interface as well as the mesh for the skin. ID3DXSkinInfo contains lots of information, such as the bone offset matrix for each bone in the hierarchy and the mapping table which describes which vertices in the skin are mapped to which bones in the hierarchy and with what weight. We will store this ID3DXSkinInfo interface in our mesh container, so that we can access it in our game loop when we need to transform and render the mesh.

It just so happens that when performing software skinning, things could not be easier for us as programmers. Indeed, the ID3DXSkinInfo interface contains a helper function called UpdateSkinnedMesh which performs the entire transformation process for us. We will take a detailed look at this interface in a moment along with the many helper functions it provides, but first let us examine this vitally important method that is used to perform software skinning.

```
HRESULT UpdateSkinnedMesh
(
    CONST D3DXMATRIX *pBoneTransforms,
    CONST D3DXMATRIX *pBoneInvTransposeTransforms,
    PVOID pVerticesSrc,
    PVOID pVerticesDst
);
```

We pass this function the source mesh vertex buffer, our array of absolute bone matrices, and our array of bone offset matrices for each bone that influences the mesh. We also pass in the vertex buffer of a destination mesh where the transformed vertices will be stored. Because the ID3DXSkinInfo object contains all of the bone-to-vertex mapping information and the bone weight values, this function can quickly step through each vertex in the source mesh vertex buffer and use vertex blending (similar to the vertex blending code we introduced earlier) to transform each vertex using the relevant bones in the passed matrix arrays.

The resulting vertices will be copied to the destination vertex buffer (in a separate destination mesh that we will store in the mesh container structure). On function return, the destination mesh can then be used for drawing. Note that the source vertex buffer is read-only and is never modified by this function. We always maintain the original model space mesh. Every time the hierarchy is updated or animated, we must perform this step to calculate new values for the destination mesh. Again, because the destination vertex buffer that we render from always contains the mesh vertices in world space, we need to set the device world matrix to an identity matrix before rendering.

The way this function is designed does cause a small, but easily solved, problem for us. We need to pass in an array of absolute bone matrices as the first parameter, but our absolute bone matrices are not stored in array format – they are stored throughout the hierarchy in their respective frames. Therefore, in the CreateMeshContainer function, we will need to do two things. First, we will need to extract all of the bone offset matrices from the ID3DXSkinInfo and store them in an array inside the mesh container.

Second, we will need to fetch the corresponding absolute bone matrices in the hierarchy and store pointers in the mesh container as well. This means that our mesh container will store an array of pointers to absolute bones matrices and a corresponding array of bone offset matrices. Finding out which bone offset matrix belongs with which frame matrix in the hierarchy is the only tricky part, and we will examine how that is done momentarily, when we look at some code for an example CreateMeshContainer function.

Note: While we could write our own vertex blending code, it is generally a better idea to use the ID3DXSkinInfo::UpdateSkinnedMesh function. On the whole, it performs vertex blending at a very respectable speed.

The basic steps for software skinning are shown next along with a detailed account of how we can implement this process.

1. Load and store the mesh as a standard mesh in the mesh container and store the passed ID3DXSkinInfo interface in the mesh container as well. This is all done inside the ID3DXAllocateHierarchy derived object's CreateMeshContainer method. We will also fetch the name of each bone stored in the ID3DXSkinInfo object and store pointers to those bones' absolute matrices in the mesh container. This gives us easy access to that information when transforming the mesh without having to traverse the hierarchy. We also retrieve the bone offset matrix for each bone from the ID3DXSkinInfo object and store them in the mesh container as well. In the end, we will have an array of bone matrix pointers and an array of their corresponding bone offset matrices ready to be sent to the UpdateSkinnedMesh method.
2. In our game loop, we animate the skeleton as usual using the animation controller (i.e. activate an animation set and call the AdvanceTime method). This will apply animation changes to the parent-relative matrices stored in each bone.
3. After calling AdvanceTime, we build the absolute (world) bone matrix for each frame by concatenating matrices as we step through the hierarchy. At the end of this stage, each bone in the hierarchy will contain an absolute world space matrix for that bone. It is these matrices that have pointers stored in the mesh container.
4. Now we call the ID3DXSkinInfo::UpdateSkinnedMesh function to transform our source mesh vertex data into a destination vertex buffer in a destination mesh. The destination vertex buffer will contain the updated world space positions of the skin's vertices. We pass this function the array of absolute bone matrices we have just updated and an array containing each bone's corresponding bone offset matrix. This function will combine each bone matrix with its corresponding bone offset matrix and use the resulting matrix to transform each vertex that is attached to that bone. The ID3DXSkinInfo object already stores the weight values, so it will manage the vertex blending calculations with little trouble.
5. We now have a destination mesh with the skin's world space vertices stored in its vertex buffer. We set the device world matrix to identity and then draw the mesh as usual -- loop through each subset, set the required textures and materials, and call DrawSubset. Remember that the destination mesh is just a regular ID3DXMesh.

Now that we know the basic steps involved in software skinning, let us look at some implementation details.

11.7.1 Storing the Skinned Mesh

Before we can render a skinned mesh we must be able to load it, and before we can do that we will need to setup some data structures to store the required information. This is what we will discuss in this section.

First, we will use the same D3DXFRAME derived structure that we used in the previous chapter. You will recall that we derived a structure from D3DXFRAME that contained an additional 4x4 matrix that to store the absolute world transform of each frame when we do our hierarchy update. We will now think of each frame as being a bone, and the frame hierarchy as the skeleton of our skinned mesh, but the core ideas remain the same.

```
struct D3DXFRAME_MATRIX: public D3DXFRAME
{
    D3DXMATRIX    mtxCombined;
};
```

The next structure that we will need to add new members to is the D3DXMESHCONTAINER. Let us first remind ourselves what the standard version of this structure looks like:

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

We will use this structure to store all of the necessary data for our skin. Unlike the previous chapter, we will now be using the pSkinInfo member to store the ID3DXSkinInfo interface that is passed to the CreateMeshContainer callback during the loading of the hierarchy. The ID3DXSkinInfo object will contain the names of all the bones used by this skin, the bone offset matrix for each of these bones, and the indices and weights describing which bones influence which vertices and to what extent. We do not have to manually create the ID3DXSkinInfo object -- it will be automatically created and populated by D3DXLoadMeshHierarchyFromX and passed to the CreateMeshContainer callback. All we have to do is store it and eventually use it to transform the mesh in our render loop. The rest of the members of this structure are used to store the ID3DXMesh (inside the MeshData member) and additional information such as materials and textures that are used by its various subsets.

Now we need to derive from this structure and add a few members of our own. For starters, we know that when we transform the model space source mesh into world space using `ID3DXSkinInfo::UpdateSkinnedMesh`, we will need a place to store the resulting vertices. Since we wish to maintain the original mesh data, we will add an `ID3DXMesh` to the mesh container called `OriginalMesh`. It will always store the mesh that was originally loaded and passed to the `CreateMeshContainer` function. Thus, we will use the mesh that is stored in the `MeshData` member of the mesh container as our destination mesh. `MeshData` will always contain the current world space transformation of the vertices and will be the mesh that we render each frame. Therefore, every time we call `ID3DXSkinInfo::UpdateSkinnedMesh`, the vertices from `OriginalMesh` will be transformed and stored in `MeshData`. This first new data member that we will add to our derived mesh container is shown below.

`ID3DXMesh *pOriginalMesh`

The next thing we have to consider is ease of access to the world matrices for each frame in the hierarchy. Recall that we need to pass this into `UpdateSkinnedMesh` as its first parameter. Theoretically, we could just copy the absolute bone matrices each time the hierarchy is traversed and updated, but that is cumbersome and time consuming. Instead, when we first create the mesh we will traverse the hierarchy and find all the bones used by the skin and store pointers to each of their absolute bone matrices in a bone matrix array. This array will be stored inside the mesh container so that when animation has been applied and the matrix for each bone has been updated, the mesh will have an array of pointers to these matrices. With these pointers on hand, we can build an array of bone matrices quickly that can be passed into the `UpdateSkinnedMesh` method. So the next new mesh container member is an array of matrix pointers:

`D3DXMATRIX **pBoneMatrixPtrs`

Note that this new member is an array of matrix pointers and not an array of matrices. This is very important, since we want each element in the array to point to the world matrix of each bone in the hierarchy used by this mesh. When the matrices are updated, these pointers will still point to them, so we will always have direct and efficient access to the absolute bone matrices used by this skin. They will not be invalidated when the bone matrices are recalculated.

We also know that before we use each absolute bone matrix to transform a vertex, it must be combined with the bone offset matrix first, so that transformations are performed in bone space. So in addition to our bone matrices, we must pass an array of corresponding bone offset matrices to the `UpdateSkinnedMesh` function. The function will automatically combine the matrices in these two arrays and use the resulting matrices to transform our skin vertices. Therefore, we will need to add another matrix array to our mesh container for the corresponding bone offset matrices. We will populate this array inside the `CreateMeshContainer` function (a one time process) with the bone offset matrices that we extract from the `ID3DXSkinInfo` object we are passed.

`D3DXMATRIX *pBoneOffsetMatrices`

Both the `ppBoneMatrixPtrs` array and the `pBoneOffsetMatrices` array should each have `N` elements, where `N` is the number of bones in the hierarchy used by the skin. We will see in a moment how the matrices will be stored such that `pBoneOffsetMatrix[N]` is the bone offset matrix for the bone stored in `*pBoneMatrixPtrs[N]`. This gives us a nice one-to-one mapping when it comes time to pass this data to the `UpdateSkinnedMesh` function.

We mentioned that the first parameter to `UpdateSkinnedMesh` should be an array of absolute matrices for each bone. The second parameter of course is an array of corresponding bone offset matrices. Perhaps you have already noticed the problem? Passing in the bone offset matrices is not a problem because we will already have these stored in the mesh container as an array (just as the function expects). But we do seem to have a problem with the absolute bone matrices. Right now, all we have is an array of matrix pointers, but the function expects an array of matrices.

So we will have to loop through all of the bones in the mesh container's bone matrix pointer array and copy the actual matrix data into a temporary array so that it can be passed to `UpdateSkinnedMesh`. Once we call `UpdateSkinnedMesh`, the original mesh will be transformed into the destination mesh and this combined matrix array will not be needed again until the next time we need to transform the mesh. Therefore, we will make this a global array that can be used as a temporary storage area for any skinned mesh that we are about to transform and render. Remember, these next two variables will not be part of the mesh container; they will be global variables. It is this temporary matrix array that we will pass to the `UpdateSkinnedMesh` function:

D3DXMATRIX *pgbl_CombinedBoneMatrices
DWORD gblMaxNumBones

Now this is where we will change the rules a bit. Since we have to loop through each element in our bone matrix pointer array and copy it into this temporary array before sending it into the `UpdateSkinnedMesh` function, we might as well combine each bone matrix with its corresponding bone offset matrix as we do so. Otherwise, `UpdateSkinnedMesh` will end up having to loop through each matrix array we pass and combine them anyway. So before calling `UpdateSkinnedMesh`, we will loop through each bone in our bone pointer array and combine each one with its corresponding bone offset array, storing the resulting matrices in this temporary array. As the matrices in this array are already combined, we can pass this array as the first parameter to `UpdateSkinnedMesh` and set the second parameter (the bone offset array parameter) to `NULL`. Thus, the matrix copy that we have to do is essentially free.

Our approach will be to allocate one global array that has enough elements to store the maximum number of bones used by any skinned mesh in our scene. When we are loading our skinned meshes from their X files, we will use a global variable called `gblMaxNumBones` to record the highest bone count so far so that we can make sure this global temporary array is large enough to accommodate any of our skinned meshes. Although this might sound complicated, it really is not -- this matrix array is just like a scratch pad where we combine each mesh's bone matrix with its corresponding bone offset matrix prior to passing it into the `UpdateSkinnedMesh` function. We will see it in use when we look at the code momentarily.

But first, let us have a look at our new derived mesh container:

```
struct D3DXMESHCONTAINER_DERIVED : public D3DXMESHCONTAINER
{
    ID3DXMESH      * OriginalMesh;
    D3DXMATRIX    ** pBoneMatrixPtrs;
    D3DXMATRIX     * pBoneOffsetMatrices;
}
```

We now have proprietary data structures with everything we will need to store for software skinning. Populating these structures is done inside the `CreateMeshContainer` method of our `ID3DXAllocateHierarchy` derived object, called by `D3DXLoadMeshHierarchyFromX` for every mesh encountered in the X file. Let us next have a look at what this interface looks like, the information contained within the underlying object, and how we can use its member functions to access this information and store it in our mesh container structure. After that, we will take a look at one possible implementation of a `CreateMeshContainer` function to see how the mesh data is loaded and stored for software skinning.

11.7.2 The ID3DXSkinInfo Interface

For every mesh found in the X file that contains bone information, `D3DXLoadMeshHierarchyFromX` will create a new `ID3DXSkinInfo` object and pass its interface to our `CreateMeshContainer` callback. We will usually store this interface in the mesh container, especially in the case of software skinning where we will wish to use some of its member functions (such as `UpdateSkinnedMesh`). Besides having useful helper functions, it also contains vital information that we will need to retrieve. For starters, it will contain an array of all of the names of the bones in our skeleton to which vertices in the mesh are assigned. It also contains an array of bone offset matrices for each of those bone matrices. This is very important because the bone matrices themselves are stored in the hierarchy with their respective names, so this is our only means of identifying which bone offset matrix in the `ID3DXSkinInfo` object belongs with (and should therefore be combined with) each bone matrix in the hierarchy. Therefore, one of the first things we will do with this interfaces is pair the bone offset matrices with their corresponding bones in the hierarchy. We will do this using the new matrix arrays that we added to our derived mesh container structure.

To accomplish this, we will first call the `ID3DXSkinInfo::GetNumBones` method to find out how many bones in the hierarchy are mapped to this mesh. Then we will implement a loop that iterates over this bone count and call `ID3DXSkinInfo::GetBoneName` to retrieve the name for each bone index. Once we have the name of the current bone, we will traverse our hierarchy and search for the frame with the same name. Once located, we will store a pointer to this frame's absolute matrix in our mesh container at `ppBoneMatrixPtr[N]` (`N` is the current loop counter). We will follow this with a call to the `ID3DXSkinInfo::GetBoneOffsetMatrix` function to return the bone offset matrix for bone `N` also. This matrix will be stored in `pBoneOffsetMatrix[N]` in our mesh container. At the end of the loop we will have a bone matrix pointer array and a bone offset matrix array in our mesh container that has a one-to-one mapping.

Before we transform the mesh we must loop through each bone in the mesh container's bone pointer array, and for the current iteration (`N`) we multiply `*ppBoneMatrixPtr[N]` by `pBoneOffsetMatrix[N]` to create a combined bone matrix that we store in our global array `pCombinedBoneMatrix[N]`. Once we have done this for each matrix, we can pass the `pCombinedBoneMatrix` array to `ID3DXSkinInfo::UpdateSkinnedMesh` function to transform our skin mesh into world space so that it is ready to be rendered.

This might sound complicated, but it can actually be done using only a few lines of code. This will be clear enough in a moment when we look at a simple example of a CreateMeshContainer implementation. But before we do that, let us first look at the member functions of the ID3DXSkinInfo interface that are applicable to software skinning. There are a few functions in this interface that are applicable only to hardware skinning, so we will postpone their coverage until later in the chapter. Since the D3DXSkinInfo object is really just an object that encapsulates all of the SkinWeights data objects found in the X file for a given mesh, most of the functions are used to simply retrieve or modify this vertex/weight/bone data.

ID3DXSkinInfo Member Functions

```
HRESULT Clone( LPD3DXSKININFO *ppSkinInfo );
```

Given a pre-populated ID3DXSkinInfo interface, we can use this function to clone the object. We pass in the address of an ID3DXSkinInfo interface pointer and on function return it will point to the newly created object. All of the bone name, bone offset matrix, and vertex weight data contained in the original skin info object will be copied into the clone.

```
DWORD GetNumBones(VOID);
```

This function returns the number of bones in the hierarchy referenced by this skin. The bone matrices themselves are not stored in this object (they are stored in the frame hierarchy) but this object does contain a list of bone names and their respective bone offset matrices. We can use the bone names to pair the bone offset matrices with their matching bone matrices in the hierarchy. In our lab projects, the value returned by this function will be used to allocate the matrix arrays we have added to our D3DXMeshContainer structure.

```
LPCSTR GetBoneName( DWORD Bone );
```

This function returns the name of the bone stored at the index specified by the input parameter. We will use this function when setting up our mesh container inside the CreateMeshContainer function to get the name of each bone and match it to a bone in the hierarchy. We can then store this bone matrix and its accompanying bone offset matrix in the mesh container.

```
LPD3DXMATRIX GetBoneOffsetMatrix( DWORD Bone );
```

This is the function we will use to retrieve the bone offset matrix for each bone in the hierarchy. Our code will first use the GetNumBones member function to find out how many bones influence this skin. This count value also defines the number of bone offset matrices (and bone names) stored in the skin info object's internal arrays. We can loop through each of these bone indices and call GetBoneName to fetch the current bone name and GetBoneOffsetMatrix to fetch the 4x4 bone offset matrix that belongs with that bone name. Once we have the bone name and bone offset matrix, we can traverse the hierarchy

to find the matching frame in the hierarchy to access its bone matrix. After we have both a pointer to the bone matrix in the hierarchy and the bone offset matrix, we can then store them in the mesh container in their corresponding data arrays (ppBoneMatrixPtrs and pBoneOffsetMatrices).

Note: Writing code to find a bone in the hierarchy would be simple enough to do. But it is even easier if we use the D3DXFrameFind function. We will see this in a moment.

```
HRESULT GetBoneInfluence(DWORD Bone, DWORD *vertices, FLOAT *weights);
```

This function allows us to retrieve the indices of the vertices in the mesh that are attached to the bone specified in the first parameter, along with the weights for each of those vertices. When using software skinning, we will not have to worry about the weight data for each vertex/bone combination because this information is stored inside the ID3DXSkinInfo object. The ID3DXSkinInfo::UpdateSkinnedMesh function will automatically use the stored vertex weights to perform the vertex blending. This function would be useful if you are performing hardware vertex blending and you need to extract the weight information for each vertex and store it in the vertex structure (but there is even a helper function that does that for us as we will see later in the chapter). This function might also be useful diagnostically if you need to output or log this information.

The first parameter is the index of the bone that we would like the vertex/weight data returned for. The second and third parameters are pointers to pre-allocated arrays that should be large enough to hold the vertex index and vertex weight data. Each element in the DWORD array describes the index of a vertex in the skin that this bone is attached to. There is a corresponding weight defined in the float array describing how much that vertex is influenced by the bone matrix. The amount of memory that we must allocate for these arrays can be determined by calling the next function we will examine (GetNumBoneInfluences).

```
DWORD GetNumBoneInfluences( DWORD bone );
```

This function is used to find out how many vertices are attached to a bone. We pass in the index of the bone that we would like to enquire about and we are returned the attached vertex count. So if the bone in question influenced four vertices, the return value would be 4. We can use this value to allocate the vertex index and weight arrays passed in as the second and third parameters to GetBoneInfluence. We will not need to use this function during software skinning but it might be useful for diagnostic purposes.

```
HRESULT GetMaxVertexInfluences( DWORD *maxVertexInfluences );
```

This function can be used if you wish to enquire about the maximum number of vertices attached to a given bone in the hierarchy. We pass in the address of a DWORD which, on function return, will be filled with this value. For example, if this value was 3 on function return, then it means that at least one of our bones (but perhaps many) has three vertices attached to it and that there are no other bones that have more than this number of vertices attached to them.

```
Float GetMinBoneInfluence(Void) ;
HResult SetMinBoneInfluence( Float minInfluence ) ;
```

These functions allow us to get and set the minimum weight thresholds used to determine how much a bone matrix will contribute towards the transformation of one of its attached vertices. For example, if we set this value to 0.2 and then called `UpdateSkinnedMesh`, any matrices that influence a vertex with a weight less than 0.2 will not be used in the vertex blend. This might be useful if you know that you have some bones mapped to vertices with very small weights which do not influence the vertex position to any noticeable degree. You could set this threshold value so that these matrices will be ignored when transforming that vertex, speeding up the software transformation process. You will only rarely need to use this functionality (we will not use it at all in our lab projects).

```
HResult GetMaxFaceInfluences(LPDIRECT3DINDEXBUFFER9 pIB,
                             DWORD NumFaces, DWORD *maxFaceInfluences ) ;
```

This function returns the maximum number of bones that affect a single triangle in the mesh. We pass in the index buffer of the mesh we are testing as the first parameter and the number of triangles in the buffer as the second parameter. The third parameter is the address of a `DWORD` which on function return will store the value of the maximum number of matrices that affect a single triangle in the mesh.

When performing software skinning, this information is of little significance since we are not limited to only using a few matrices per face. But when we discuss hardware skinning later in the chapter we will see that things are very different. For example, a given graphics card might only allow the setting of 6 bone matrices on the device at any given time, even though it is possible that each vertex in a triangle may be influenced by as many as 4 bones. This means that a single triangle could be influenced by 12 unique matrices and to be correctly transformed and rendered by the 3D pipeline we would need the ability to set those 12 matrices. This cannot be done on the hardware in our example so we would have to place the device in software vertex processing mode where the limitation is removed (we must have a software or mixed mode device to be able to do this). Note that the pipeline will still perform the vertex blending for us, but it will be done by the DirectX software transformation module. So we would lose the advantages of hardware T&L for that mesh. Please note that this is not the same as software skinning (which we are currently discussing) where the pipeline will not have the vertex blending render state enabled at all. In other words, when performing software skinning, the pipeline does not multi-matrix transform (i.e., vertex blend) our vertices. We are instead using a function (`UpdateSkinnedMesh`) to do our vertex blending outside of the pipeline. We will discuss pipeline vertex blending capabilities and hardware skinning later in the lesson.

```
HResult UpdateSkinnedMesh
(
    CONST D3DXMATRIX *pBoneTransforms,
    CONST D3DXMATRIX *pBoneInvTransposeTransforms,
    PVOID pVerticesSrc,
    PVOID pVerticesDst
) ;
```

This is the function that performs vertex blending when using software skinning. It transforms the model space vertices in the source mesh into vertex blended world space vertices in the destination mesh.

This function is called during the render loop to update the skinned mesh vertices whenever any of its bones have been animated or changed. The first parameter is an array of bone matrices (the absolute/world space frame matrices) and the second parameter is an array containing the corresponding bone offset matrices. The third parameter is the vertex buffer from the source mesh which contains the model space vertices of the skin in its reference pose (as passed into the CreateMeshContainer function by D3DXLoadMeshHierarchyFromX). The fourth parameter is the vertex buffer of the destination mesh that will receive the world space vertices and will be used to render the mesh in its current position and orientation. This function first combines each bone matrix in the pBoneTransforms array with the corresponding bone offset matrix in the pBoneInvTransposeTransforms array to generate the final world space matrix for each bone. It then multiplies each vertex with the bone matrices that influence that vertex to generate the blended vertex for the final vertex buffer. If the vertex format contains a normal, then the normals will also be blended in the same way, but without the translation portion of the matrix being used.

Recall that in our code we will allocate a third (global) matrix array called pCombinedBoneMatrices to store the combined bone matrices prior to passing them to this function. We do this because our mesh container stores an array of pointers to the bone matrices and not an array of the bone matrices themselves. In order to pass the bone matrices into this function, we would first need to copy them into a temporary array. Since we are looping through each bone anyway for copying purposes, we might as well just do the multiplication ourselves and store the result in the pCombinedBoneMatrices array. Since this global array will now hold the combined bone matrices, we can simply pass this array in as the first parameter to UpdateSkinnedMesh and set the second parameter to NULL. The function will still work perfectly; it will simply skip the matrix concatenation step that we just performed (thus making our matrix copy a cost-free operation).

11.7.3 Manual Skin Creation

It is possible to create an empty `ID3DXSkinInfo` and populate it programmatically or with skinning information stored in a custom file format. For example, you might have a mesh object that was not originally intended as a skin (or perhaps it was, but it was not provided in skin format). You can programmatically build your own frame hierarchy to represent the bones of the skeleton and then write some code that maps that skeleton to the mesh. Alternatively you might start with a skeleton and then procedurally build a skin to fit around it. Once done, you could create an empty `ID3DXSkinInfo` and populate it with the information for each bone, such as its name in the hierarchy, which vertices are influenced by that bone, and the bone weights. You would also need to calculate the bone offset matrix for each bone (which we learned how to do earlier) and store those as well.

The `ID3DXSkinInfo` interface has three ‘Set..’ functions which can be used to populate an empty `D3DXSkinInfo` object. Before we look at these, let us first talk about the global `D3DX` function that can be used to create an empty `D3DXSkinInfo` object. Although you will not need to use these functions when you are loading your data from X files, they are extremely useful if you are building your skinned meshes by hand. While we will not spend time in this chapter doing this, it is an important technique to learn. So in the next chapter, we will spend a good deal of time examining how to construct a skinned mesh manually to create some interesting animating trees for our outdoor environments. For now, we will just get an overview of the functionality involved.

```
HRESULT D3DXCreateSkinInfoFVF(DWORD numVertices, DWORD FVF,  
                               DWORD numBones,  
                               LPD3DXSKININFO* ppSkinInfo );
```

Although `ID3DXSkinInfo` does not store the actual vertex data (this is stored in a separate `ID3DXMesh`), the first parameter should contain a value describing the number of vertices in the mesh that we intend to use as the skin and the second parameter should be the FVF flags for the vertex structure we intend to use. This is important, because the `UpdateSkinnedMesh` function will need to know if the vertex contains a normal so that the normal is also blended when the vertex is transformed. The third parameter is the number of bones that this skin will need to be mapped to (i.e. the number of bones in the hierarchy for which we intend to attach a vertex). The `D3DXCreateSkinInfoFVF` function will use this number to allocate an internal bone name array and a bone offset matrix array of the correct size. The fourth parameter is the address of an `ID3DXSkinInfo` interface pointer which on successful function return will point to a valid interface to an empty `D3DXSkinInfo` object.

Once we have created an empty `ID3DXSkinInfo` object we need to populate it with data -- filling in the name of each bone, the vertex indices and weights for that bone and the bone offset matrix for that bone.

```
HRESULT SetBoneName( DWORD BoneNumber, LPCSTR BoneName );
```

This function is used to set the name of a bone. The first parameter is the index of the bone we wish to set the name for. For a newly created object, this will start off at 0 and increase as we add each bone's information. This index should never be greater than or equal to the `numBones` parameter passed into

the `D3DXCreateSkinInfoFVF` function. The `BoneName` we pass in should also match the name of one of the bones in the hierarchy.

```
HRESULT SetBoneInfluence(DWORD Bone, DWORD numInfluences,  
                        CONST DWORD *vertices, CONST FLOAT *weights);
```

Once you have set the bone name, you have to tell `ID3DXSkinInfo` which vertices in the skin mesh are attached to this bone and the weight at which this bone should be used to influence each of them. The first parameter is the index of the bone we are setting the vertex mapping for. This should be followed by a `DWORD` parameter describing how many vertices in total will be influenced by this bone. As the third parameter we pass in an array of `DWORDs` where each element in the array describes the index of a vertex in the skin that will be influenced by this bone to some degree. The fourth parameter is where we store the matching weights for each vertex index specified in the previous parameter.

```
RESULT SetBoneOffsetMatrix( DWORD Bone, LPD3DXMATRIX pBoneTransform );
```

This function allows us to set the bone offset matrix for each bone. This matrix can be calculated by traversing the reference pose hierarchy (concatenating as we go) to the bone in question and then taking the inverse of the concatenated matrix.

At this point we have a through enough understanding of what the `ID3DXSkinInfo` interface contains and how to extract that data to store it in the mesh container to move forward. We also know how to manually create and populate an `ID3DXSkinInfo` if we need to fill it with some other non-X file data. Finally, it is worth remembering that, in the software skinning case at least, once we have extracted this information we do not just want to release the `ID3DXSkinInfo` interface. Instead we want to store it in the mesh container because we will use its `UpdateSkinnedMesh` function to transform the vertices of the skin into world space.

Let us now see how all of this theory can be put into practice by looking at a simple `CreateMeshContainer` implementation.

11.7.4 The `CreateMeshContainer` Function

In Chapter Nine we learned how to derive a class from the `ID3DXAllocateHierarchy` interface and implement its `CreateMeshContainer` function. This function will be called for each mesh contained in the X file. If the mesh is a skinned mesh, then the function will be passed an `ID3DXMesh` containing the skin (this is just a regular mesh) and an `ID3DXSkinInfo` interface that will contain the bone to vertex mapping information. Our function will need to store the mesh and the `ID3DXSkinInfo` interface pointer in the mesh container and will need to extract the bone offset matrices and store them as well. Our other responsibility is to find the matching bone matrices in the hierarchy and store pointers to them in our mesh container. Let us now have a look at the code to a typical function that would do all of this.

In this next example we will not show the processing of the material or texture data since we already know how to do this. The more complete function is of course fully implemented in the source code that accompanies this chapter. It is important to note that this implementation of the CreateMeshContainer function is for the software skinning case only. We will look at how this function will need to be implemented for the hardware skinning cases later (in our lab projects we will write a single function that handles all cases).

The function starts by allocating a new mesh container structure and getting a pointer to the ID3DXMesh passed into the function. This mesh will contain the skin in its model space reference pose.

```
HRESULT CAllocateHierarchy::CreateMeshContainer
( LPCTSTR Name, LPD3DXMESHDATA pMeshData, LPD3DXMATERIAL pMaterials,
  LPD3DXEFFECTINSTANCE pEffectInstances, DWORD NumMaterials,
  DWORD *pAdjacency,
  LPD3DXSKININFO pSkinInfo, LPD3DXMESHCONTAINER *ppNewMeshContainer )
{
    D3DXMESHCONTAINER_DERIVED *pMeshContainer = NULL;
    UINT iBone, cBones;
    LPDIRECT3DDEVICE9 pd3dDevice = NULL;
    LPD3DXMESH pDestinationMesh = NULL;
    *ppNewMeshContainer = NULL;

    // Get a pointer to the mesh of the skin we have been passed
    pDestinationMesh = pMeshData->pMesh;

    // Allocate a mesh container to hold the passed data.
    // This will be returned from the function
    // where it will be attached to the hierarchy by D3DX.
    pMeshContainer = new D3DXMESHCONTAINER_DERIVED;
    memset(pMeshContainer, 0, sizeof( D3DXMESHCONTAINER_DERIVED ));
```

Next we copy over the name of the mesh passed into the function and store it in the mesh container. We also get a pointer to the device object to which the mesh belongs as we will need it in a moment.

```
// If this mesh has a name then copy that into the mesh container too.
if ( Name ) pMeshContainer->Name = _tcsdup( Name );

pDestinationMesh->GetDevice(&pd3dDevice);
```

The next section of code is optional; it is not specific to skinned meshes and applies to meshes of any kind. It is shown here for completeness. It tests to see if the input mesh has vertex normals defined. If not, then in this example we will assume that they are desired and clone the mesh to create a new skin that has room for vertex normals. We store the cloned result in the mesh container's MeshData member. After cloning, we create normals using the very handy D3DXComputeNormals helper function.

```
// if no normals exist in the mesh, add them ( we might need to use lighting )
if ( !(pDestinationMesh->GetFVF() & D3DFVF_NORMAL) )
{
    pMeshContainer->MeshData.Type = D3DXMESHTYPE_MESH;

    // Clone the mesh to make room for the normals
    pDestinationMesh->CloneMeshFVF( pMesh->GetOptions(),
```

```

        pMesh->GetFVF() | D3DFVF_NORMAL,
        pd3dDevice,
        &pMeshContainer->MeshData.pMesh );

    pDestinationMesh = pMeshContainer->MeshData.pMesh;

    // Now generate the normals for the pmesh
    D3DXComputeNormals( pDestinationMesh, NULL );
}

```

In the above code, we clone the mesh and divert the `pDestinationMesh` pointer to point at the newly created mesh interface. Remember, this originally pointed at the mesh passed into the function by D3DX which will no longer be used (because it did not have normals). We did not increment the reference count of the mesh when we assigned this pointer and thus do not need to release it here (it was a temporary pointer assignment). When this function returns, D3DX will release its hold on the original mesh interface causing it to be unloaded from memory. This is exactly what we want since we will prefer to use the newly cloned mesh instead.

If the mesh already contained normals, then we will use this mesh as is and simply store the passed mesh in the mesh container's `MeshData` member. We are careful to increment the mesh's reference count so that it is not unloaded from memory on function return when D3DX releases its claim on the object.

```

else // if normals already exist, just add a reference
{
    pMeshContainer->MeshData.pMesh = pDestinationMesh;
    pMeshContainer->MeshData.Type = D3DXMESHTYPE_MESH;
    pDestinationMesh->AddRef();
}

```

At this point in the function the mesh has been copied into the mesh container and contains vertex normals -- so far we have done nothing that we would not do for a standard mesh.

The next section of code is where we identify whether or not this is a skinned mesh. If the `pSkinInfo` parameter is not `NULL`, then this is indeed a skinned mesh and we will need to fill out the new members of our mesh container structure. Before we do this, we would probably store the textures and materials as shown in previous lessons, but we will skip that step for this example to keep things simple.

If this is a skinned mesh then we will copy the passed `ID3DXSkinInfo` interface pointer into the mesh container and make sure that we increment the reference count (we do not want the object prematurely destroyed when the function returns).

```

// If there is skinning information, save off the required data
// and then set up for skinning because this is a skinned mesh.
if (pSkinInfo != NULL)
{
    // first save off the SkinInfo and original mesh data
    pMeshContainer->pSkinInfo = pSkinInfo;
    pSkinInfo->AddRef();
}

```

The next step is to find out how many bones this skin is influenced by, using the ID3DXSkinInfo interface GetNumBones function. Once we have this value we know how large to make the mesh container's bone offset matrix array. After we allocate the bone offset matrix array, we loop through each bone and extract its bone offset matrix using the ID3DXSkinInfo::GetBoneOffsetMatrix member function and store the matrix in the mesh container's array.

```
// We now know how many bones this mesh has
// so we will allocate the mesh containers
// bone offset matrix array and populate it.
NumBones = pSkinInfo->GetNumBones();
pMeshContainer->pBoneOffsetMatrices = new D3DXMATRIX [NumBones];

// Get each of the bone offset matrices so that
// we don't need to get them later
for (iBone = 0; iBone < NumBones; iBone++)
{
    pMeshContainer->pBoneOffsetMatrices[iBone] =
        *(pMeshContainer->pSkinInfo->GetBoneOffsetMatrix(iBone));
}
```

You might assume that we should now set up the mesh container's ppBoneMatrixPtrs array so that we can store all of the pointers to the absolute bone matrices in the hierarchy. However, at this point the hierarchy has not yet been fully created, so this is something that we will have to do after we have loaded the entire hierarchy and the D3DXLoadMeshHierarchyFromX function has returned program flow back to our application.

What we can do now though is check how large our temporary global combined matrix array is going to be because it needs to be able to hold enough matrices to accommodate the skinned mesh that uses the largest number of bones. So we check the current size of this global array (which will initially be allocated to a small default size) and if it is too small to contain the number of bones used by the mesh we are loading, we have found a new largest bone count and we resize the array. After all skinned meshes have been loaded, this array will be large enough to hold the bones of any skinned mesh in the scene. Recall that this global array is used by our application as a scratch pad prior to transforming a skinned mesh. It will be used to hold an array of concatenated bone and bone offset matrices for the mesh which will be passed into the ID3DXSkinInfo::UpdateSkinned Mesh function and used to multi-matrix transform the vertices.

```
// Resize the global scratch array if this mesh
// has more bones than any others previously loaded
if ( gblMaxNumBones < NumBones)
{
    gblMaxNumBones = NumBones;

    // Allocate space for blend matrices
    delete [] pgl_CombinedBoneMatrices;
    pgl_CombinedBoneMatrices = new D3DXMATRIX [ NumBoneMatricesMax ];
}
```

At this point our mesh container stores the model space reference pose mesh, the ID3DXSkinInfo interface, and all of the bone offset matrices. As discussed earlier, our preference will be to use the

MeshData member of the mesh container to store our destination mesh (i.e., the render-ready world space mesh), but right now it currently contains the original model space vertex data (i.e., the source mesh). So we will clone the mesh and store a pointer to the clone in the pOriginalMesh member that we added to our mesh container. Unlike its MeshData sibling, this mesh will never be overwritten and will always contain the model space mesh reference pose. This is the mesh whose vertex buffer will be passed into UpdateSkinnedMesh as the source vertex buffer. The transformed vertices will be copied into the destination mesh stored in the MeshData member of the mesh container for later rendering.

```
// Finally clone the mesh so that we now
// have a copy of it stored in the 'OriginalMesh' member
// of the mesh container.
// This is the mesh that will be the source mesh in transformation
pDestinationMesh->CloneMeshFVF( D3DXMESH_MANAGED,
                                pMeshContainer->MeshData.pMesh->GetFVF(),
                                m_pd3dDevice,
                                &pMeshContainer->pOriginalMesh );
}
```

As it turns out, these are the only steps we will need to take in our CreateMeshContainer function when we are creating a mesh for software skinning.

Finally, we assign the passed mesh container pointer to point at our newly created mesh container so that when the function returns, D3DXLoadMeshHierarchyFromX will have access to it and will be able to attach it to the hierarchy.

```
// Return the transformation
*ppNewMeshContainer = (D3DXMESHCONTAINER *)pMeshContainer;
pMeshContainer = NULL;

pd3dDevice->release();
return D3D_OK;
}
```

11.7.5 Fetching the Bone Matrix Pointers

After we have loaded our scene using D3DXLoadMeshHierarchyFromX, any skinned meshes that were contained inside the X file are now in their respective mesh containers in the hierarchy. However, our mesh container also has a new member called ppBoneMatrixPtrs which stores pointers to each absolute bone matrix for easy access after hierarchy changes. After we have finished loading the hierarchy, our next task will be to allocate this array so that we can populate it with the relevant matrix pointers. Our scene initialization code will usually look something like the code shown next. Note that this code assumes that CAllocateHierarchy is the name of our ID3DXAllocateHierarchy derived class and that pDevice is a pointer to a valid Direct3D device object.

```
D3DXFRAME pFrameRoot;
ID3DXAnimationController *pAnimController;
CAllocateHierarchy Alloc;
D3DXLoadMeshHierarchyFromX("MyFile.x", D3DXMESH_MANAGED,
```

```
pDevice, &Alloc, NULL, &pFrameRoot,
&m_pAnimController);
```

```
SetupBoneMatrixPointers ( pFrameRoot , pFrameRoot );
```

The SetupBoneMatrixPointers function will be a simple recursive function that we write to finalize our mesh container data. In our actual code for this lesson we will store each loaded hierarchy inside a separate CActor class, for ease of data management. So if we loaded three X files, each one containing a separate skinned mesh, we would have three CActor objects in our application -- each representing a different character with its own root frame and its own animation controller. We will also make the SetupBoneMatrixPointers function we are about to discuss a member function, but for now we will forget about this encapsulation and just look at the techniques involved. We will call the following function for each skeletal hierarchy that we load:

```
HRESULT SetupBoneMatrixPointers(LPD3DXFRAME pFrame , LPD3DXFRAME pRoot)
{
    if (pFrame->pMeshContainer != NULL)
        SetupBoneMatrixPointersOnMesh ( pFrame->pMeshContainer, pRoot );

    if (pFrame->pFrameSibling != NULL)
        SetupBoneMatrixPointers ( pFrame->pFrameSibling , pRoot );

    if (pFrame->pFrameFirstChild != NULL)
        SetupBoneMatrixPointers ( pFrame->pFrameFirstChild , pRoot );

    return S_OK;
}
```

This function recursively traverses each frame (bone) in the hierarchy and tests to see if the frame has a mesh container attached to it. If it does, then it calls the SetupBoneMatrixPointersOnMesh function to set up the matrix pointers in the mesh container. We will look at that function in a moment. After SetupBoneMatrixPointers has returned control back to the caller, all of the mesh containers which store skins will have had their ppBoneMatrixPtrs array allocated and populated with pointers to each absolute bone matrix in the hierarchy. Notice that SetupBoneMatrixPointers passes the root frame pointer down the tree as well so that it is accessible in the SetupBoneMatrixPointersOnMesh function. This allows us to search the hierarchy starting at the root for a specific bone.

Let us now examine the SetupBoneMatrixPointersOnMesh function.

```
HRESULT SetupBoneMatrixPointersOnMesh( LPD3DXMESHCONTAINER pMeshContainerBase ,
                                         D3DXFRAME *pRootFrame )
{
    UINT iBone, NumBones;
    D3DXBone *pFrame;

    D3DXMESHCONTAINERDERIVED *pMeshContainer =
        (D3DXMESHCONTAINER_DERIVED*)pMeshContainerBase;

    // If this is a skinned mesh then setup its bone matrix pointers
```

```

if (pMeshContainer->pSkinInfo != NULL)
{
    NumBones = pMeshContainer->pSkinInfo->GetNumBones();
    pMeshContainer->ppBoneMatrixPtrs = new D3DXMATRIX*[cBones];

    for (iBone = 0; iBone < NumBones; iBone++)
    {
        pFrame = (D3DXBone*)D3DXFrameFind( pRootFrame,
                                            pMeshContainer->pSkinInfo->GetBoneName(iBone));

        pMeshContainer->ppBoneMatrixPtrs[iBone] = &pFrame->mtxCombined;
    }
}

return S_OK;
}

```

The first thing this function does after casting the mesh container to our derived `D3DXMESHCONTAINER_DERIVED` structure is test to see if the `ID3DXSkinInfo` pointer is set to `NULL` in the mesh container. If not, then this mesh container contains a skin and we retrieve the number of bones that the mesh uses (just as we did in `CreateMeshContainer`). This value is used to allocate the container's `ppBoneMatrixPtrs` array to hold that many matrix pointers. Next, we loop through each bone stored in the `ID3DXSkinInfo` object and retrieve its name so that we can use the `D3DXFrameFind` function to search the hierarchy (starting at the root) for a frame with a matching name. Once we have a pointer to this frame, we store a pointer to its absolute matrix in the corresponding element in the `ppBoneMatrixPtrs` array. Remember, these absolute matrices will be rebuilt every time the hierarchy is updated. So this array provides fast access to the matrices after modifications have been made without have to perform a traversal of the tree.

We now have the absolute bone matrix pointers stored in the mesh container in exactly the same order that the bone offset matrices were stored in `ID3DXSkinInfo`. This provides a one-to-one mapping between `pBoneOffsetMatrices` and `ppBoneMatrixPtrs` in the mesh container for easy access later on.

11.7.6 Animating the Skeletal Hierarchy

Skeletal hierarchy animation will be no different from the animation techniques we studied in Chapter Ten. We simply use the animation controller interface to set and activate mixer track animation sets and then call the `AdvanceTime` method to run the process. After the relative matrices of the hierarchy (i.e. the skeleton) have been updated by the controller, we will build the absolute matrices in the hierarchy by traversing through the structure, combining relative matrices as we go. When our traversal is complete, each bone in the skeleton will have its absolute world space position and orientation stored in its `mtxCombined` member. If the frame is being used as a bone for a skin, a pointer to this combined matrix will be stored in the mesh container's `ppBoneMatrixPtrs` array.

The following code shows how a function could be called to update the animation of the bone hierarchy, which is immediately followed by the call to `UpdateFrameHierarchy` (which traverses the hierarchy and calculates the combined frame matrices).

```

void UpdateAnimation ( D3DXMATRIX matWorld, float m_fElapsedTime )
{
    if ( m_pAnimController != NULL )
        m_pAnimController->AdvanceTime(m_fElapsedTime);

    UpdateFrameMatrices(m_pFrameRoot, &matWorld);
}

```

You can see that there is nothing new here at all. We pass in the world matrix that we will use to position and orient the bone hierarchy and the updated time in seconds that is used to update the animation controller's internal timer. We will actually have a function like this as part of our CActor class so that we can individually update each skinned mesh in our scene. Nevertheless, do not forget that it is possible for multiple skinned meshes to be stored in a single hierarchy. In that case, we would be updating the animation and position of the entire hierarchy stored in a specific actor using the above code.

11.7.7 Transforming and Rendering the Skin

After we have applied skeletal animation and updated the absolute matrix for every bone in the hierarchy, we are ready to traverse the hierarchy and render any meshes that it contains. The hierarchy traversal is no different from the previous lessons -- we simply traverse the hierarchy and render any mesh containers we find (visibility testing preferably included). However, before we render a mesh container, we will want to see if it is a skinned mesh or not. If it is not, then we will render the mesh just as we did before. However if the mesh is a skinned mesh, then we will need to call `ID3DXSkinInfo::UpdateSkinnedMesh` to transform and blend the vertices from the source mesh into world space and store them in the destination mesh for rendering.

We will not bother showing the code that traverses the hierarchy and searches for mesh containers to render since we covered this in earlier lessons and it remains unchanged. The function we will look at is the one that renders each mesh container, called during hierarchy traversal for every mesh container found in the hierarchy. The function in this example is passed a pointer to the mesh container to be rendered and a pointer to the frame which owns the mesh container. The owner frame is important if this is not a skinned mesh because this frame's combined matrix will be the matrix we wish to use as the world matrix to render the mesh.

The first thing this function should do is check the mesh container's `ID3DXSkinInfo` interface pointer to see if it is `NULL`, because this informs the function whether the stored mesh is to be used as a skin. If not, then the mesh can be rendered normally. If it is a skin then additional steps must be taken.

```

void DrawMeshContainer( LPD3DXMESHCONTAINER pMeshContainerBase ,
                      D3DXFRAME pFrameBase )
{
    D3DXSkinContainer* pMeshContainer = ( D3DXSkinContainer* ) pMeshContainerBase;
    D3DXBone *pFrame = (D3DXBone*)pFrameBase;

    // first check for skinning

```

```
if (pMeshContainer->pSkinInfo != NULL){
```

If we get into this code block then the mesh container stores a skin. When this is the case, the first thing we do will be to set the world matrix to an identity matrix, since we will not require the pipeline to transform our vertices into world space. We will do this manually using software vertex blending. We also get the number of bones used by the mesh.

```
D3DXMATRIX Identity;
DWORD NumBones = pMeshContainer->pSkinInfo->GetNumBones();
DWORD i;
PBYTE pbVerticesSrc;
PBYTE pbVerticesDest;

// set world transform
D3DXMatrixIdentity(&Identity);
m_pd3dDevice->SetTransform(D3DTS_WORLD, &Identity);
```

Now we need to loop and combine each absolute bone matrix used by this mesh with its corresponding bone offset matrix. We will store the concatenated result in our global temporary matrix array.

```
// set up bone transforms
for ( i = 0; i < NumBones; ++i)
{
    D3DXMatrixMultiply
    (
        &pgbl_CombinedBoneMatrices[i],
        &pMeshContainer->pBoneOffsetMatrices[i],
        pMeshContainer->ppBoneMatrixPtrs[i]
    );
}
```

We now have an array of matrices to use for transforming the vertices from the source mesh into the destination mesh. Thus, we lock the vertex buffers in both the source and destination meshes for access to the memory.

```
pMeshContainer->pOriginalMesh->LockVertexBuffer(D3DLOCK_READONLY,
                                                (LPVOID*) &pbVerticesSrc);
pMeshContainer->MeshData.pMesh->LockVertexBuffer(0,
                                                (LPVOID*) &pbVerticesDest);
```

We now pass the pointers for both vertex buffers into the UpdateSkinnedMesh function. We also pass in the combined Bone/BoneOffset matrices we just stored in the temporary global array. When the function returns, the destination mesh in the mesh container will store the new world space blended vertex data for the skin in its current position (based on its current skeleton).

```
// generate skinned mesh
pMeshContainer->pSkinInfo->UpdateSkinnedMesh( pgbl_CombinedBoneMatrices, NULL,
                                              pbVerticesSrc, pbVerticesDest);
```

Note that the second parameter to the above function is supposed to be a pointer to an array of bone offset matrices, but we passed in NULL. As discussed earlier, we decided to combine the matrices

ourselves in order to handle the matrix copy that would have been necessary if we wanted to pass the matrix arrays in separately (because our mesh container stores matrix pointers, not matrices). UpdateSkinnedMesh will recognize that the bone offset array parameter is NULL and will therefore skip the concatenation process that we just performed ourselves.

We can now unlock the two vertex buffers.

```
pMeshContainer->pOrigMesh->UnlockVertexBuffer();  
pMeshContainer->MeshData.pMesh->UnlockVertexBuffer();
```

At this point, the mesh container now has the updated world space vertex data for the skin stored in the MeshData member (our destination mesh) so we can proceed to render this mesh as we would any other mesh. Below we see a call to a function called DrawMesh, which we can imagine would just loop through each of the mesh subsets, set any relevant textures and materials, and then call the DrawSubset function until the mesh was rendered in its entirety. The destination mesh is just a regular mesh after all and can be rendered using all of the standard techniques.

```
// render the mesh  
DrawMesh ( pMeshContainer->MeshData.pMesh )  
}
```

If this mesh container did not store a skinned mesh, then we will just set the world matrix to the combined matrix of its owner frame and render it (as discussed in previous lessons).

```
else  
{  
    IDirect3DDevice9 * pDevice;  
    pMeshContainer->MeshData.pMesh->GetDevice(&pDevice);  
    pDevice->SetTransform(D3DTS_WORLD, &pFrame->mtxCombined);  
    DrawMesh(pMeshContainer->MeshData.pMesh)  
    pDevice->Release();  
  
} // end if this is not skinned mesh  
  
} // end function
```

11.7.8 Software Skinning Summary

When we break it down into its separate parts, software skinning is really quite simple. We have seen that we can load a skeleton from an X file using the familiar D3DXLoadMeshHierarchyFromX and we now know that the skeleton is simply a standard frame hierarchy, where each frame is considered to be a bone. We also know how to modify our CreateMeshContainer function to support the loading of skinned mesh data from an X file. We saw that this function is passed a standard mesh containing the model space vertices of the mesh in its reference pose and an ID3DXSkinInfo interface which tells us which frames in the hierarchy are used by the mesh as bones. Animation was also quickly taken care of since the skeleton is treated in exactly the same way as any normal frame hierarchy.

Finally, we saw that skinned mesh rendering requires a few additional steps, but is also straightforward enough. We use the `ID3DXSkinInfo::UpdateSkinnedMesh` function to convert the model space skin vertex data into world space vertex data using vertex blending. We pass this function an array of bone and bone offset matrices along with our source mesh vertices and a destination vertex buffer that will receive the transformation results. The `ID3DXSkinInfo` object will manage the entire transformation process using the bone weights stored internally. It can do this because the `ID3DXSkinInfo` object contains all the vertex weights for each influential bone that was stored in the X file. As discussed earlier, these weights are used to scale the contribution of bones on one of its assigned vertices.

Once we have the world space vertex data in the destination mesh, we can simply set the world matrix of the device to an identity matrix and draw the mesh subsets as we normally would.

In the workbook that accompanies this chapter you will see that code similar to the sample code shown in this textbook has been implemented inside our `CActor` class. We will implement both software skinning and two different types of hardware skinning techniques in the code, so be sure to focus on the software skinning technique for now. We will begin to tackle the hardware techniques in the next section.

Software Skinning Advantages

- Very easy to understand and implement.
- Requires no hardware support and will therefore work on any system.
- There is no limit to how many bones a single vertex can be attached to because all of the bone/weight information is stored in the `ID3DXSkinInfo` object. The only limits we face are more practical: the memory taken up by the bone/vertex mapping information and the speed at which the vertices can be transformed by the CPU.
- Because the transformation is done in software, software skinning does not affect the way we batch render our primitives. This allows us to continue to batch by states such as textures and materials for optimal performance. This is not true for hardware skinning; we will be forced to batch by bone contributions as the primary key (discussed later).

Software Skinning Disadvantages

- Often considerably slower than hardware skinning techniques because vertices are transformed in software inside the `UpdateSkinnedMesh` method of the `ID3DXSkinInfo` interface. With dedicated hardware skinning, the vertices are multi-matrix transformed in the DirectX pipeline (by the GPU on a T&L card). As we will see however, non-indexed hardware skinning (unfortunately, the more commonly supported hardware skinning technique) can often suffer a performance hit as well due to less than ideal batching requirements.

11.8 DirectX Skinning II: Non-Indexed Skinning

The DirectX fixed-function pipeline provides two skinning methods that can take advantage of available 3D hardware support for vertex blending. It is worthy of note that the software skinning technique we have just discussed is not one of these techniques, since it was not performed in the DirectX pipeline. Software skinning simply sent a regular mesh to be transformed and rendered by the pipeline in the usual way; the fact that this mesh was being used as a skin was of no significance to the pipeline. This was because we transformed the vertices into world space manually with the help of the `ID3DXSkinInfo` interface. When it came time to render the skin, we simply handed it off to the pipeline as a regular mesh (albeit containing world space vertices). The pipeline was not aware that the vertices were transformed according to some skeletal pose or how they ended up in the destination mesh. So it is very important to remember, as we move forward, that the software skinning technique previously discussed was *not* performed in the T&L pipeline.

There are two geometry blending techniques that are integral to the pipeline. They can be invoked as an alternative to software skinning to accelerate the skinning process in hardware. These techniques are referred to as **non-indexed skinning** and **indexed skinning**. The latter is the more efficient of the two, but support is not as widely available, especially on older T&L graphics hardware. However, even if hardware support is not available for either of these techniques, we can still use the pipeline's skinning techniques by rendering the mesh using software vertex processing. This will of course mean that the vertex blending is carried out in the software module of the vertex transformation pipeline and thus is not going to be as fast. But the important point is that both techniques will work on all systems even when no hardware support is available. The software vertex blending module of the DirectX pipeline is still quite efficient, so this is a good fallback for us even in commercial applications.

You should note the distinction between software skinning as discussed previously and the pipeline skinning techniques performed using software vertex processing. As mentioned, in software skinning the pipeline plays no part in the generation of the world space vertex transformations for the skin. But when using either indexed or non-indexed pipeline skinning, we do not have to transform the vertices of the skin from model to world space ourselves. We simply pass the skin to the pipeline as model space vertex data and the pipeline will multi-matrix blend the vertices into world space for us. If hardware support is available, then the GPU will be utilized to perform this geometry blending with great speed and efficiency. If hardware support is not available, the pipeline can still perform the blending transformations for us using its own software transformation module.

If we intend to utilize the pipeline's vertex blending module on a system that does not support hardware acceleration, we will need to place the device into software vertex processing mode prior to rendering the skin so that the software module can be utilized by the pipeline. This also means that the mesh that contains the skin will need to be created with the `D3DXMESH_SOFTWAREPROCESSING` flag so that the software module of the pipeline can be allowed to work with its vertices. This brings other considerations to the fore. We know from our discussions in Chapter Three that in order for the pipeline to be able to perform software vertex processing, we must either create a software device or a mixed-mode device. This contrasts with the kind of device we would usually be striving for -- either a hardware device or the even more efficient PURE device. We cannot use software vertex processing with the latter two device types, so it should be obvious that if we intend to use the pipeline skinning techniques

and we want our application to work on an end user system that does not have hardware support for vertex blending, ideally we will want to create a mixed-mode device. This will allow us to render of all our regular geometry with the device set to hardware processing mode (so that the vertex data is transformed and lit by the GPU) and then switch to software processing mode for our chosen pipeline skinning technique. Although the default state of a mixed-mode device is hardware vertex processing mode, if the device had previously been placed into software vertex processing mode, we would want to place it back into hardware mode prior to rendering our regular meshes for maximum speed. To do so we would use the following function call:

```
pDevice->SetSoftwareVertexProcessing(FALSE);
```

When the time comes to render our skinned meshes (and no hardware support exists), we can place the mixed mode device into software mode as follows:

```
pDevice->SetSoftwareVertexProcessing(TRUE);
```

Anything we render from this point on will use the DirectX software transformation pipeline. While performance will not be as good, we have given ourselves the option of using the pipeline skinning techniques even when hardware support for vertex blending is absent on the current system

Of course, in order to switch between hardware and software vertex processing techniques you will need to have created a mixed-mode device. If you had created a hardware device or a PURE device, then software vertex processing could not be invoked and the pipeline would fail to render the skin on a device that had no hardware support.

Note: Be careful not to set the vertex processing mode to software for all of your rendering (even on a mixed-mode device). Once software vertex processing is enabled, any T&L acceleration that the 3D card has to offer will be ignored and all vertices rendered from that point on (until software vertex processing is disabled) will be transformed and lit in software. This will incur a significant performance penalty if you use it to render your entire scene rather than just the meshes that require software emulation to account for the lack of hardware support.

11.8.1 Why use software skinning at all?

If the DirectX pipeline can perform the vertex blending in software when no hardware support is available, then why would we ever need to use the software skinning technique we discussed in the previous section? The answer will become clearer as we discuss the semantics of the DirectX transformation pipeline with respect to vertex blending, but for starters, we know that with software skinning, we are not limited by how many bones are allowed to influence a single vertex. This is because the weights for each vertex for a given bone are stored inside the ID3DXSkinInfo object and are used when we manually transform the mesh using UpdateSkinnedMesh. However, when using the pipeline to perform vertex blending, we do not have the luxury of storing elements such as vertex weights in external data objects like ID3DXSkinInfo. Once we call DrawPrimitive, the model space vertices of the skin are sent off into the pipeline and from that point forward things are pretty much out of our hands. When the pipeline is performing vertex blending, we have to pass the weights in the vertex

structure itself. This is how the pipeline (and hopefully the GPU, if hardware support is available) is informed as to how strongly each currently set world matrix is to influence the vertex being transformed.

In a few moments we will look at how to store the weight values in the mesh vertices as well as how to set multiple world matrices (bone matrices) on the device. The important point right now is that there is a limit on how many weights can be stored in a vertex. This is because vertices need to be compact packets of information that can be passed over the bus at a reliably decent speed and stored efficiently in either local or non-local video memory. Therefore, DirectX limits the number of weights that we can store and use in a vertex to 3 in the non-indexed skinning technique. This will mean that a single vertex can, at most, be influenced by four matrices. Why four? As it happens, the pipeline expects the combined weights stored in the vertices to always add up to 1.0, so if we have three weights stored in a vertex, the pipeline can calculate the fourth weight by summing the first three and subtracting from 1.0. This clever design decreases the memory footprint of vertices in precious video memory and lowers the amount of subsequent bus traffic.

Weight4 = 1.0 – (Weight1 + Weight2 + Weight3)

Although four weights per vertex might not sound like much, it is generally more than adequate given how most skinned meshes are constructed and animated. Unfortunately, the situation is quite a good deal worse when we use non-indexed vertex blending. You will see in a moment how we can set up to four world (i.e. bone) matrices on the device. That is, before a vertex is sent to the pipeline, we will need to set the bone matrices that influence it. The weights in the vertices match a corresponding matrix; the pipeline will use the first weight defined in the vertex to weight the contribution of the first currently set world matrix, the second weight will be used to weight the contribution of the second currently set world matrix, and so on.

However, we know that the vertex blending pipeline is invoked after we have called the DrawPrimitive function (or the DrawSubset wrapper function), so the smallest entity we will be rendering will be at least one triangle. We do not render one vertex at a time, but rather one triangle at a time, so it is only between DrawPrimitive calls that we can change states -- like setting different world matrices. As a result, when using non-indexed blending, not only are we limited to a maximum of four matrices per vertex, but consequently, we are limited to four matrices *per subset*.

This last point has a real impact on our ability to batch render effectively. Since we can only change matrices between draw calls, our ability to batch render triangles has been greatly reduced. We will now need to attribute sort our mesh not only by textures and materials, but primarily by bone (matrix) contributions. This implies re-ordering the mesh triangles such that all triangles that use the same four bones will belong to the same subset. This can greatly reduce the number of triangles that can exist in a single subset and therefore greatly reduce the number of triangles in the skinned mesh that we can render with a single call to DrawPrimitive. For example, a typical non-skin mesh might have 20 triangles that use the same texture and material and usually we would attribute sort the mesh such that all of these triangles would belong to a single subset. However, in a skin mesh, each one of these triangles might be attached to a different combination of 4 bones. While this is probably an extreme example, the point is that it would cause not one, but twenty subsets to be created, each with a single triangle and each requiring its own DrawPrimitive call. Again, we have to batch by bone because we will need to set the bone matrices on the device before we render a given subset.

As if things do not sound bad enough, three weights (four matrices) is the maximum that the DirectX pipeline supports, so we will be able to use all three weights if we are using software vertex processing. However, if we want the pipeline to take advantage of hardware support for vertex blending, then we may hit another wall -- the graphics hardware might not support all four matrix blends. In some cases only two or three matrix blends are supported. When this is the case, the mesh will have to be modified so that only the supported number of weights is used in each vertex. This would usually involve having to step through the weight/bone information, removing less influential bones from attached vertices. The result is that the skin might not animate as nicely as the artist had intended. If a given vertex was assigned to four bones by the artist, but the current hardware only supports two bones, two bones would need to be detached from this vertex and not used. The vertex would now store only one weight value to blend the two matrices we would use to transform it.

You may be thinking that the thought of loading a mesh and determining how many bones the hardware supports and having to convert that data into a hardware friendly mesh, batched into subsets based on bone contributions, sounds like a complete nightmare. However, you will be glad to know that support is available. When the mesh is first loaded inside the `CreateMeshContainer` function, we can use the `ID3DXSkinInfo::ConvertToBlendedMesh` function to perform all of these steps for us. We will look at this function in more detail later, but for now just know that we will pass this function the regular `ID3DXMesh` containing the skin that is passed into our `CreateMeshContainer` function. `ConvertToBlendedMesh` will automatically clone the mesh into a format that contains the correct number of vertex weights in each vertex and will attribute sort the mesh so that triangles are batched primarily by bone combinations. This means that all of the triangles that use one set of up to four matrices will belong in one subset; all the triangles that use another combination of four matrices will belong in another subset; and so on. Subsets are then further subdivided so that a single subset will contain only triangles that share the same material and texture. As you can imagine, this means we can end up with subsets that are quite small, even in typical cases.

For example, the skinned mesh contained in ‘Tiny.x’ which ships with the DirectX SDK uses only a single texture and material. This is something we try to target when working with skinned meshes, for obvious reasons. If we were to register Tiny as a regular mesh, it would have only one subset and we could render it with a single draw call. However, when Tiny is converted into a skin that is compatible with the non-indexed skinning technique, the new mesh will have been broken up into over 30 different subsets (and that is assuming that four matrix blends are available on the hardware). So even though each subset uses the same texture and material, the triangles in each subset use different combinations of 4 bone matrices which need to be set on the device before we issue each draw call. This batching problems stems not from only being able to have four bone influences per vertex, but from the implied one-to-one mapping between vertex weights and matrix slots on the device. As there are at most four weights, they always map to the first four matrices set on the device.

We can check how many blend weights the mesh returned from the `ConvertToBlendedMesh` function needs to use and make some decisions of our own. If this value is beyond the limits of the hardware, we can choose to invoke software vertex processing at this point (device type permitting).

When it comes time to render the mesh, we will loop through each subset, set the world (bone) matrices that the triangles in that subset are influenced by (which `ID3DXSkinInfo` tells us), and then render that

subset with vertex blending enabled, using the `D3DRS_VERTEXBLEND` renderstate that we will take a look at shortly.

In this section we will discuss non-indexed blending. This is the skinning technique which is most commonly supported by hardware (even on older T&L cards) but has the disadvantages just described. Later in this lesson we will discuss the second hardware accelerated skinning technique called indexed blending, where we are no longer limited to only four influences per subset. This will allow for much more efficient batching.

11.8.2 Setting Multiple World Matrices

In order for the DirectX pipeline to do a multi-matrix blend of the vertices in our skinned mesh, we need a way to bind more than one world matrix to the device at a time. When using non-indexed blending, a vertex can have up to three weights, so it will require the pipeline to transform it using up to four world matrices. Whereas in the software skinning case we simply passed our entire array of bone matrices into the `UpdateSkinnedMesh` function, now we will need to send the device only the bone matrices that the subset we are about to render will use.

As it turns out, the software skinning code we used to extract the bone offset matrices from `ID3DXSkinInfo` and store them in the mesh container can be reused. This part of the system will not change -- we will continue to use the `ID3DXSkinInfo` object passed into the `CreateMeshContainer` function to retrieve the bone offset matrices and store them in the mesh container. We will similarly traverse the hierarchy to find the matching absolute bone matrices so that we can store those in the mesh container as well. Therefore, when it comes time to render, our mesh container will store an array of bone offset matrices and a matching array of absolute bone matrix pointers just as it did in the software skinning case.

The difference now is that we will not simply combine all bone matrices with their corresponding bone offset matrices into the global matrix buffer and run the update (although we will still need to perform the concatenation). Instead, we will first need to determine which bones in our bone array are needed by the subset we are about to render. This is because these are the bones that the weights defined in the subset vertices are relative to. When we examine our modified `CreateMeshContainer` function a bit later in the lesson, we will see how to convert the skin into a mesh that contains the vertex weights. We will be able to subsequently retrieve the mesh in the desired vertex format along with something called a *bone combination table*. The bone combination table will be an array of structures, where each element in the array will contain the bone indices used by a given subset. Before we render skin subset N, we will check element N in our bone combination table to determine which bones are relevant. This structure in the table will store an array of bones indices which we can use to locate the needed world matrices in the mesh container. Because we stored our bone matrix pointers inside the mesh container in the same order that the bone offset matrices were stored in the `ID3DXSkinInfo` object, these bone combination table indices will directly index into our array of bone matrix pointers and bone offset matrices stored in the mesh container.

As an example, let us imagine that we are about to render subset 5 of a mesh, and element 5 in our bone combination table tells us that this subset uses bones 6 and 20. We know that we must combine each absolute bone matrix with its bone offset matrix stored in the corresponding mesh container arrays before we set them on the device as a world transformation matrix:

```
D3DXMatrixMultiply(&Matrix1,
                  &pMeshContainer->pBoneOffsetMatrices[6],
                  pMeshContainer->ppBoneMatrixPtrs[6] );

D3DXMatrixMultiply(&Matrix2,
                  &pMeshContainer->pBoneOffsetMatrices[20],
                  pMeshContainer->ppBoneMatrixPtrs[20] );
```

As you can see, the two bones that influence the subset we are about to render have been combined with their corresponding bone offset matrices such that we now have two temporary combined matrices which we can bind to the device as world matrix 1 and world matrix 2. For each vertex in this subset, its first weight will determine how influential Matrix1 (bone 6) will be and the second weight will determine how influential Matrix2 (bone 20) will be.

For the time being, do not worry about how we find out which bones affect which subset; we will see this all later when we take a detailed look at the ConvertToBlendedMesh function. The bigger point here is that in the above example we have created two combined matrices which we now need to set on the device so that the pipeline can access them. So how do we set more than one world matrix on the device?

The World Matrix Palette

A Direct3D device actually has the ability to store 511 matrices. When we call the SetTransform device method and pass in a transform state such as D3DTS_VIEW to set the view matrix, this transform state actually equates to an index into a larger palette of matrices which places the view matrix at the index in this matrix palette where the device expects it to be stored. The first 256 matrix positions (transform states 0 – 255) are used by the pipeline to store the view matrix, the projection matrix, the viewport matrix, and other matrices such as the texture matrices for each texture stage. We will not normally place matrices into these positions using raw matrix indices. Rather, we use transform states such as D3DTS_VIEW and D3DTS_PROJECTION to map to matrix palette indices somewhere in the 0 – 255 range. However, transform states 256 – 511 are reserved to hold up to 256 world matrices. Up until now, we have only used one of these world matrices so you might have assumed that there was only one world matrix slot on the device. As it happens, when we set a world matrix like so:

```
pDevice->SetTransform ( D3DTS_WORLD , &mat );
```

we are actually setting this matrix in the device matrix palette at index 256 (the first palette position where world matrices begin). That is, D3DTS_WORLD is defined in the DirectX header files with a value of 256. When we are not using vertex blending, we only need to use this first world matrix position. But we now see that we have the ability to set many more world matrices simply by passing in

the index of the matrix we wish to set. For example, if we wanted to set three world matrices, we could do the following:

```
pDevice->SetTransform ( 256 , &mat1 );
pDevice->SetTransform ( 257 , &mat2 );
pDevice->SetTransform ( 258 , &mat3 );
```

Bear in mind however that setting the matrices does not mean that the transformation pipeline will use them -- we have to enable vertex blending for that to happen. If the vertex blending render state has not been set to TRUE, then only the first world matrix (256 or D3DTS_WORLD) will be used to transform vertices. Since it is not very intuitive to set the world matrices using raw index numbers, DirectX supplies us with a macro that maps indices 0-255 into the 256-511 range. This is useful because if we know we have to set four bone matrices to render a given subset, we tend to think of these as being matrices 0-4 and not matrices 256-259. Therefore, when setting world matrices we will use the D3DTS_WORLDMATRIX macro, defined in the DirectX header files as:

```
#define D3DTS_WORLDMATRIX(index) (D3DTRANSFORMSTATETYPE)(index + 256)
```

As you can see, this simple macro just adds 256 to the passed index. Thus, we would rewrite the above example code as:

```
pDevice->SetTransform ( D3DTS_WORLDMATRIX ( 0 ) , &mat1 );
pDevice->SetTransform ( D3DTS_WORLDMATRIX ( 1 ) , &mat2 );
pDevice->SetTransform ( D3DTS_WORLDMATRIX ( 2 ) , &mat3 );
```

Although the result is the same, it is much more intuitive to read. We can see that matrix (0) maps to weight (0) in the vertex, matrix (1) maps to weight (1) in the vertex, and so on. In this example where only three matrices are being used, you should keep in mind that the weight for matrix (2) would not be stored in the vertex since the pipeline will automatically calculate it by doing $1.0 - (\text{weight0} + \text{weight1})$. If a maximum of N matrices are being used, there should be N-1 weights in the vertex structure. Weight N is always calculated by the pipeline.

You may be wondering why 256 world matrix slots have been allocated when we know that a maximum of four matrices is supported. After all, in the current version of DirectX, only three weights can be defined in the FVF format of the vertex (the fourth weight is calculated on the fly) and these are paired up with the corresponding matrices that are set on the device. Since there is an implicit mapping between weight positions in the vertex and matrix indices on the device, it must follow then that given the restriction of four weights, only four of the possible 256 matrices will ever be used during blending. The other 252 matrices would appear to be wasted space. As it turns out, when performing non-indexed blending, this is absolutely the case. At most, we will only ever use the first four world matrix slots on the device (Fig 11.20). It is because of this limitation that the non-indexed skinning technique suffers such inefficient batching (all of the triangles in a given subset must use the same four bone matrices in order to be rendered together).

An example vertex with 3 weights

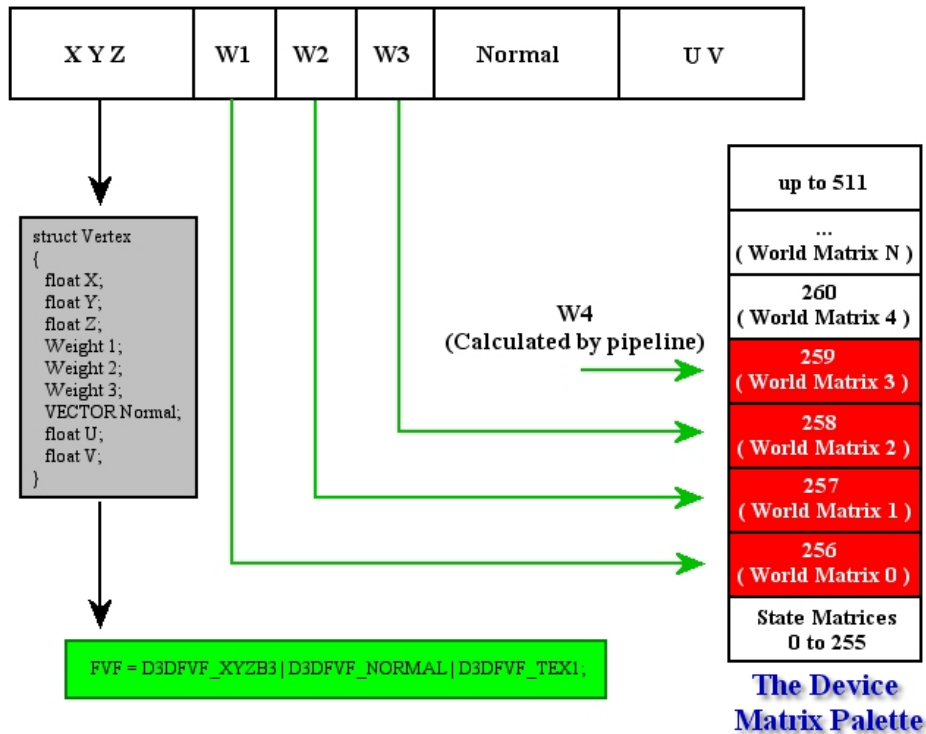


Figure 11.20

In Fig 11.20 we see an example of a vertex format that contains three weights, which gives us access to the first four world matrices on the device. Notice the FVF flags in the green box that we use to define the vertex format for the vertex buffer that would contain these weighted vertices. The `D3DFVF_XYZB3` flag is an FVF flag we have not encountered yet in this series. It is used in place of the typical `D3DFVF_XYZ` to specify that each vertex will have three blend weights defined. As you might imagine, there are other flags for defining vertices with different blending weights (one, two, three, four, or five weights as it turns out):

Additional FVF blending flags:

D3DFVF_XYZB1

D3DFVF_XYZB2

D3DFVF_XYZB3

D3DFVF_XYZB4 // not currently supported

D3DFVF_XYZB5 // not currently supported

These FVF flags are mutually exclusive since they each define a vertex as having untransformed X, Y, and Z components along with some specified number of weights. Fig 11.20 also shows us that the weights should be defined immediately after the XYZ position component in the vertex structure. The number of float weights you place in your vertex must match the FVF flags that you intend to use.

We will not actually need to create a vertex buffer ourselves using the flags listed above because the `ConvertToBlendedMesh` function will create the mesh (and therefore its vertex buffer) for us in the correct format. However, you are certainly permitted to use the pipeline's vertex blending features

without having used this particular function to set up the mesh on your behalf. In such cases you would need to create the vertex buffer yourself with the appropriate FVF flags and vertex weight values.

Later in the lesson we will discuss the second pipeline skinning technique called indexed blending. It is much more efficient, but unfortunately not as widely supported in hardware. Using this technique, each vertex can contain up to 3 weights as in the non-indexed case, but can also contain 4 indices using a packed DWORD. These indices allow the vertex to specify not only the weights themselves, but exactly which bone matrix in the device matrix palette that each weight applies to -- up to 256 now allowed. Consequently, subsets will be much larger and even a single triangle could be influenced by 12 unique bone matrices. Since a subset could contain many triangles all influenced by different bones somewhere in the 256 matrix palette, this makes a significant difference. We mentioned the case of Tiny.x earlier and saw that over 30 skin subsets were created when using non-indexed blending (each of which could only access four matrices at a time). In the indexed case, the mesh contains only a single subset and can be rendered with a single draw call. This is because we can set all of the bone matrices on the device for the character simultaneously before we render the mesh. The vertices of the mesh accurately describe to the pipeline (using their stored indices), which of the currently set matrices its weights are defined for.


For the time being, let us not get distracted by indexed vertex blending, for we still have a ways to go in our examination of non-indexed blending. Like it or not, non-indexed blending will often be the only hardware accelerated vertex blending technique supported on an end user's system, so you will definitely want to accommodate it in your skinned mesh implementation.

11.8.3 Enabling Vertex Blending

To enable vertex blending on the device we use the D3DRS_VERTEXBLEND render state:

```
pDevice->SetRenderState ( D3DRS_VERTEXBLEND , D3DVERTEXBLEND_FLAGS );
```

The second parameter should be a member of the D3DVERTEXBLEND_FLAGS enumerated type. The default state for this render state is D3DVBF_DISABLE, which means the pipeline will perform no vertex blending and a single world matrix (stored at index 256) will be used for vertex transformations. This is the mode we have used in all our demos up until this point where a single world matrix was needed.



```
typedef enum
_D3DVERTEXBLEND_FLAGS
{
    D3DVBF_DISABLE = 0,
    D3DVBF_1WEIGHTS = 1,
    D3DVBF_2WEIGHTS = 2,
    D3DVBF_3WEIGHTS = 3,
    D3DVBF_TWEENING = 255,
    D3DVBF_0WEIGHTS = 256
} D3DVERTEXBLEND_FLAGS;
```

Before rendering the mesh, we will tell the device how many matrices we wish to use per vertex. One point that needs to be clear is that if we pass D3DVBF_2WEIGHTS for example, then the vertices in our mesh must have *at least* two weight components defined in the FVF. However, keep in mind that every vertex in the mesh must have the same number of weights because all vertices in a mesh vertex buffer must be of the same format.

We will use the `ConvertToBlendedMesh` function to convert our initial skin mesh into one that has the correct number of weights. This function essentially informs the pipeline of the number of weights defined in each vertex, which will be equal to the maximum number of bone influences for a single vertex in the mesh (minus 1 because the last weight is never stored, it is calculated on-the-fly).

If our mesh had all of its vertices influenced by only two bones, but one vertex in the mesh happened to be influenced by four bones, then every vertex in the mesh would have three weights defined and we would use the `D3DVBF_3WEIGHTS` flag before rendering this mesh. Any vertices that are only influenced by two bones (and would normally only need one weight) will still have three weights -- the two additional weights are added to pad the vertex structure. They will be assigned weights of 0.0 so that they do not affect the vertex transformation.

Although these ‘padding’ weights will not affect the final transformation results for the vertex, it is necessary to note that they will still be used to blend matrices, even though the result will be a no-op because it is scaled to zero. We will see later how we can introduce a possible optimization during rendering in the non-indexed case. Essentially, before rendering a subset, we will find out the maximum number of matrices used by that subset and adjust the `D3DRS_VERTEXBLEND` render state to this amount. This will allow us to make sure that the additional (padding) weights are not used to needlessly blend matrices that will have no effect on the final vertex position. Using our previous example, where the mesh vertices have been padded to three weights and most of the subsets are influenced by only two matrices, we would determine before we render a subset which of the two categories it falls into. If it is influenced by two matrices then we can set the `D3DRS_VERTEXBLEND` to `D3DVBF_1WEIGHTS` so that the additional padded weights (weights 2 and 3 in this example) and their respective matrices are not needlessly included in the vertex transformation.

Therefore, the second parameter to this function describes not the total number of weights defined in the vertices, but the number we wish to use for transformations (which may be less). Let us examine the possible values the `D3DRS_VERTEXBLEND` render state can be set to and see what they mean when used.

D3DVBF_0WEIGHTS

This value informs the device that the vertices we are about to render do not contain any weights and therefore will be transformed by only one matrix with an assumed weight of 1.0. This might sound like a very strange flag to be able to set because surely a vertex that uses one matrix with a weight of 1.0 is a vertex that is not being blended. Indeed it would seem that this flag would have the same effect as using the `D3DVBF_DISABLE` flag. As it turns out, this is actually true in the case of non-indexed blending and in fact, there will be no need to use this flag under those circumstances. However, in the indexed blending case, we might have a triangle where each of its three vertices will need to use different world matrices. Therefore, while vertex blending is not taking place, it is still possible that we might have a mesh where every vertex is only attached to one bone, but the vertices belonging to the same triangles/subsets might not all use that same bone for transformation. While no actual blending is taking place, three matrices could still be used to transform a single triangle (e.g., one matrix for each vertex). This would not be possible using only the single world matrix that is available using the `D3DVBF_DISABLE` flag. This will all make more sense when we discuss indexed skinning later in the lesson.

D3DVBF_1WEIGHTS

This flag informs the device that we wish to render our triangles using vertex blending where each vertex will have a single weight. Remember that if the vertex has one weight, then this means it is influenced by two bones and will be blended using two matrices. The second weight is calculated by the pipeline as $\text{Weight2} = 1.0 - \text{Weight1}$. After setting this state, any triangles that you render must have at least one vertex weight defined or behavior will be undefined.

D3DVBF_2WEIGHTS

This flag informs the pipeline that we wish to render our triangles such that they will be transformed using three world matrices. After this state has been set, any triangles we render must have a vertex format that contains at least two weights. When this state is set the vertex will be transformed using the first three world matrices on the device. The weight of the third matrix is calculated as $\text{Weight3} = 1.0 - (\text{Weight1} + \text{Weight2})$.

D3DVBF_3WEIGHTS

This flag informs the pipeline that we wish to render some triangles such that they will be transformed using four world matrices. After this state has been set, any triangles we render must have a vertex format that contains at least three weights. When this state is set, the vertex will be transform using the first four world matrices on the device. The weight of the fourth matrix is calculated by the pipeline using the calculation: $\text{Weight3} = 1.0 - (\text{Weight1} + \text{Weight2} + \text{Weight3})$.

D3DVBF_TWEENING

We will not be using vertex tweening in this lesson and therefore we will not require this flag. However you already have a good high level understanding about how tweening works, so you should be able to use the SDK documentation to fill in the gaps if you would like to use this technique in your applications. (Recall that tweening was an interpolation technique between two pre-defined sets of mesh vertices in different key animation poses.)

D3DVBF_DISABLE

This flag will disable vertex blending and set the device back to its default mode of using a single world matrix to transform vertices.

We have now covered all of the theory involved in the non-indexed skinning technique, so it is time to discuss some implementation details. We will need to examine what our `CreateMeshContainer` function will now look like for the non-indexed skinning case. We will also want to have a discussion about the very important `ID3DXSkinInfo::ConvertToBlendedMesh` function, since it will do so much of the setup work for us.

11.8.4 Storing the Skinned Mesh

Whether we are using software skinning, non-indexed skinning, or indexed skinning, our application loads the skinned mesh and bone hierarchy in exactly the same way: using the `D3DXLoadMeshHierarchyFromX` function. It is in the `CreateMeshContainer` function of our `ID3DXAllocateHierarchy` derived class that we are passed a regular mesh and an `ID3DXSkinInfo` interface. Our job will be to use the tools available to create a mesh that is capable of being vertex

blended by the pipeline. Therefore, we will skip straight to the CreateMeshContainer function and the mesh container structure we will need to store the required information.

Our mesh container will need a few additional members when we perform skinning in the pipeline. Also, when performing indexed or non-indexed skinning we will no longer need our temporary global matrix array as we did when performing software skinning. This is because this array was used to store the resulting matrices of combining the bone matrices with their corresponding bone offset matrices before sending them to UpdateSkinnedMesh. We will no longer need this temporary matrix buffer because we will be setting the matrices (a maximum of a four at a time in the current case) directly on the device. The new mesh container will accommodate the CreateMeshContainer function for a non-indexed skinned mesh and is defined next.

```
struct D3DXMESHCONTAINER_DERIVED : public D3DXMESHCONTAINER
{
    DWORD          NumAttributeGroups;
    DWORD          NumInfl;
    LPD3DXBUFFER    pBoneCombinationBuf;
    D3DXMATRIX**    ppBoneMatrixPtrs;
    D3DXMATRIX*     pBoneOffsetMatrices;
    DWORD          iAttributeSW;
};
```

Four new data members have been added to this structure and one has been removed. We no longer need to store the original mesh since the concept of a source and destination mesh is no longer valid. We will now store a single model space mesh in the MeshData member of the base structure and it is this mesh that will be transformed and rendered by the pipeline. Notice that the two matrix arrays remain. As before, the ppBoneMatrixPtrs array will contain pointers to all of the absolute bone matrices in the hierarchy that affect this mesh and we also maintain the corresponding bone offset matrix array. Most of the new members are used to store information that is returned from the ConvertToBlendedMesh function, which we will look at shortly.

DWORD NumAttributeGroups

When ID3DXSkinInfo::ConvertToBlendedMesh is called, our mesh will be cloned into a new format where the vertices contain the correct number of weights. But this function also does a lot more than simply clone the mesh into a different vertex format. Most importantly, it also breaks the mesh into different subsets so that no individual subset is influenced by any more than four matrices (or perhaps less depending on the hardware), since this is all we have available with the non-indexed skinning technique. This is likely to create many more subsets/attribute groups than the original mesh had and we will need to know how many subsets the new mesh has been divided into because there will be exactly this many elements in our bone combination buffer (also returned from the ConvertToBlendedMesh function and stored in the mesh container). Each element in the bone combination buffer will represent a subset in the mesh and will contain information such as the matrix indices into our bone/bone offset arrays which we will need in order to render that subset. We will look at the bone contribution table in a moment.

The NumAttributeGroups DWORD describes how many subsets the blended mesh has and is the number we will need to loop against in order to set the matrices for each subset and render it.

DWORD NumInfl

This member will store a value that will be returned from `ConvertToBlendedMesh` that describes the maximum number of influences that a single vertex has in the resulting mesh. Subtracting one from this value will tell us exactly how many weights we have in the vertex structure of the converted mesh.

If this value is set to 4 then it means that there is at least one vertex in the mesh that is influenced by four matrices (and will therefore need to have 3 weights stored for it). In that case, every vertex in the mesh will have three weights. Any vertices that are influenced by fewer than 3 weights will have padding weights set to 0.0 as needed.

LPD3DXBUFFER pBoneCombinationBuf

This member stores a pointer to an `ID3DXBuffer` interface. The interface to this buffer is returned from the `ConvertToBlendedMesh` function and is stored in the mesh container for use during rendering. The buffer will contain the bone combination table for this mesh -- an array of `D3DXBONECOMBINATION` structures. Each structure in the array contains information for a single subset in the converted mesh. This means that the array will contain `NumAttributeGroups` `D3DXBONECOMBINATION` structures.

When rendering the mesh, we will loop through each of its subsets/attribute groups and fetch its `D3DXBONECOMBINATION` structure from this buffer. This structure will tell us which texture and material to use, which vertex and index ranges in the vertex and index buffers belong to this subset, and of course, which bone matrices need to be set on the device.

The `D3DXBONECOMBINATION` structure is defined as follows:

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

This is a very important structure for a number of reasons. First, it is our only link between subsets in the converted mesh and their relationship to the bones in our mesh container. It is also our only link between the textures and materials that these new subsets use. Remember that although we are passed the `D3DXMATERIAL` buffer (in `CreateMeshContainer`) which describes the texture and material to use for each subset in the original mesh, after we have converted the mesh using the `ConvertToBlendedMesh` function, the subsets will have been completely rearranged and subdivided.

Note: The bone combination buffer is calculated automatically during `ConvertToBlendedMesh`, so we will not have to figure out these values ourselves.

Before continuing our mesh container examination, let us take some time to discuss the members of this new structure. Keep in mind that there will be one of these structures in the buffer for each subset in the converted mesh.

DWORD AttribId

This value is our means of indexing into the original D3DXMATERIAL buffer passed into the CreateMeshContainer function. It will describe the texture and material that should be used to render this subset. Remember that the converted mesh will have had its subsets completely recalculated, so there is no longer have a 1:1 mapping with the D3DXMATERIAL buffer. That is why this link between the new subset and old D3DXMATERIAL array is needed.

DWORD FaceStart**DWORD FaceCount**

These two members describe the range of triangles in the index buffer of the converted mesh which are considered to be part of this subset. When the ConvertToBlendedMesh function executes, each new subset it generates will have its triangles stored together in the index buffer. FaceStart describes the index of the first triangle in the index buffer that belongs to this subset and FaceCount describes how many triangles (starting at FaceStart) are included in this subset.

DWORD VertexStart**WORD VertexCount**

These two members describe the range of vertices in the vertex buffer that belong to this subset. When the ConvertToBlendedMesh function executes, each new subset it generates will have its vertices stored together in the vertex buffer. VertexStart describes the index of the first vertex in the vertex buffer that belongs to triangles in this subset. VertexCount describes how many vertices (starting at VertexStart) are used by triangles in this subset.

DWORD *BoneId

This member is an array of DWORD indices describing which matrices in our mesh container matrix arrays are used by this subset. This tells us which matrices need to be set on the device in order to render this subset. The size of this array will always be equal to the NumInfl value we added to our mesh container. As an example, let us imagine once again the case where a mesh might have been converted such that each vertex has three weights. This means that, at most, a vertex will be blended using four matrices. In this instance, the BoneId array of every subset will have four elements in it. These four elements are the indices of the four matrices that the vertices in this subset need to have sent to the device before they are rendered. If the current subset being processed only uses two matrices out of the possible four, then the last two bone indices in the subsets BoneId array will be set to UINT_MAX and their corresponding weights in each vertex will be set to zero.

To better illustrate what BoneId contains, let us imagine that the current subset we wish to render has a BoneId array containing the values {4, 3, 1, UINT_MAX}. Since the vertices in this subset are influenced by only three of the possible four matrices we can set, we would need to set the following matrices on the device before rendering this subset:

```
D3DXMatrixMultiply(&Matrix0,
    &pMeshContainer->pBoneOffsetMatrices[4],
    pMeshContainer->ppBoneMatrixPtrs[4] );

D3DXMatrixMultiply(&Matrix1,
    &pMeshContainer->pBoneOffsetMatrices[3],
    pMeshContainer->ppBoneMatrixPtrs[3] );
```

```

D3DXMatrixMultiply(&Matrix2,
    &pMeshContainer->pBoneOffsetMatrices[1],
    pMeshContainer->ppBoneMatrixPtrs[1] );

pDevice->SetTransform ( D3DTS_WORLDMATRIX ( 0 ) , &matrix0 );
pDevice->SetTransform ( D3DTS_WORLDMATRIX ( 1 ) , &matrix1 );
pDevice->SetTransform ( D3DTS_WORLDMATRIX ( 2 ) , &matrix2 );

```

We now see that the values stored in the BoneId array for each subset directly index into our mesh container matrix arrays. This is made possible by the way we store our bone pointers in the same order they are listed in the ID3DXSkinInfo object. We saw how to store the matrices in the mesh container in this order during our software skinning discussion and this process is unchanged.

Of course, for the above code to function properly we will need to enable vertex blending before we render the subset. Since we know that each vertex in our example mesh will have three weights defined and that any unused weights will be set to zero, we can safely set the vertex blend mode as follows and render all subsets of our mesh with this single setting:

```

pDevice->SetRenderState ( D3DRS_VERTEXBLEND , D3DVBF_3WEIGHTS );

```

This will work properly because all of our subset vertices will have three weights defined. This is true even if the subset we are about to render only uses three matrices (two of its three weights) as is the case in the above example where the last element in the BoneId array is set to UINT_MAX. Since the last weight will be set 0.0, any matrix set in the fourth matrix slot on the device will have its contribution scaled to 0.0 and will not effect the position of the transformed vertex. You will notice in the above code example that we only set three of the four possible matrices because the subset only uses three matrices (as described by the BoneId array). The contents of the fourth device matrix are insignificant; even if it is not set to an identity matrix and just contains garbage data, its contribution is reduced to zero by the pipeline so it will have no effect.

The BoneId array allows us to perform a rendering optimization in the non-indexed skinning case. For example, we know that if a subset has a BoneId array of {4, 10, UINT_MAX, UINT_MAX} then we only need to set two matrices, even though the vertex will have three weights. The last two weights will be set to zero, so we do not have to bother setting the third and fourth device matrices as just discussed. This is wasteful however, since every vertex in this subset will still be blended with all four matrices even though the final two matrix multiplies will have no effect on the outcome. Therefore, rather than just setting the D3DRS_VERTEXBLEND renderstate to the number of weights per vertex in the mesh (3 in our example) and rely on the zero weights of the vertex to cancel out the contributions from unused matrix slots, we can check the BoneId array to find out exactly how many matrices will really be needed by this subset. Once we find out this information, we could set the D3DRS_VERTEXBLEND render state to process only this number of vertex weights/matrices.

So in the example subset above, although the vertices contain three weights (because some other subset must have needed to use this many) this particular subset only needs to use two weights for its three matrix contributions. If we set the D3DRS_VERTEXBLEND state to D3DVBF_2WEIGHTS before rendering this subset, we can avoid the overhead of unnecessary per-vertex matrix multiplication. Since

only two of the three defined weights will be used by the pipeline, the fourth matrix slot on the device will be completely ignored.

The following code snippet demonstrates how to test the BoneId array of the current subset we are about to render to see how many matrices will be needed. We then use this value to set the D3DRS_VERTEXBLEND render state so that non-contributing matrices and weights are ignored.

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

// Render each subset
for (iAttrib = 0; iAttrib < pMeshContainer->NumAttributeGroups; iAttrib++)
{
    NumBlend = 0;
    for (DWORD i = 0; i < pMeshContainer->NumInfl; ++i)
    {
        if (pBoneComb[iAttrib].BoneId[i] != UINT_MAX)
        {
            NumBlend = i;
        }
    }

    // Use only relevant matrices / weights
    pDevice->SetRenderState(D3DRS_VERTEXBLEND, NumBlend);

    // Set matrices and render subset here
}
```

Now that we know what the bone contribute table in our mesh container will contain, we can continue our discussion of the new members added to our mesh container.

DWORD iAttributeSW

This value will store the zero-based index of the first subset in the mesh that needs to be rendered with software vertex processing enabled. For example, if this value is set to 5, it means that the first five subsets can be rendered with hardware vertex processing and that the remaining subsets (starting from index iAttributeSW up to NumAttributeGroups) need to be rendered with software vertex processing. This is because some of the subsets in that range will need to use more vertex blends than the hardware can support. In software vertex processing mode, we know the pipeline will always have the capability to blend up to four matrices, and the converted mesh will never have any more than three weights defined, so all is fine. However, the hardware may not support all four matrix blends and may instead support only two for example. So if we have a mesh where some of the subsets require more matrix blends than the hardware supports, we will need to render those subsets using software vertex processing.

ConvertToBlendedMesh will try to intelligently arrange the mesh based on the hardware capabilities of the device. For example, if the hardware supports only two weights (three matrix influences) but the original bone/vertex mapping describes one or more vertices as being influenced by more bones than three (four or more), then the function will try to return a mesh that only has three weights defined

without severely compromising the skin/bone relationship. In that case, means we can render the entire mesh using hardware vertex processing.

However, if the device supports a maximum of only two influences per vertex (i.e. one weight) then the function may not be able to approximate the mesh down to this level without severely compromising the relationship between the skin and bones. Therefore some of the subsets in the returned mesh may still require more matrix blends than the device can handle in hardware. It is in this instance that we will use the `iAttributeSW` flag to distinguish these subsets so that we can optimize the rendering of the mesh using two passes. In the first pass we will render all of the subsets that are influenced by no more than two bones using hardware vertex processing. In the second pass we will enable software vertex processing and render the remaining subsets that have three or more influences. Because we have the index of the first subset that is non-hardware compliant, we do not have to loop through and test all subsets again from the beginning during the second pass. Instead we can start checking subsets from this index.

After we call the `ConvertToBlendedMesh` function to create the new skinned mesh (in our `CreateMeshContainer` function), our first job will be to check how many matrix blends the hardware supports. We do this by querying the **MaxVertexBlendMatrices** member of the `D3DCAPS9` structure as shown below.

```
D3DCAPS9 d3dcaps;
pDevice->GetDeviceCaps ( &d3dcaps );
DWORD MaxNumberBlends = d3dcaps.MaxVertexBlendMatrices;
```

Once we have this value, we will loop through each subset in the `D3DXBONECOMBINATION` buffer and search each `BoneId` array to see if that subset requires more matrix influences than the hardware supports. Once we find the first subset in the buffer that exceeds the hardware capabilities, we will set the `iAttributeSW` member to this index. Now we can render the mesh using the two pass approach just discussed. In the first pass we will render all subsets in hardware using this logic.

```
for ( int I = 0 ; I < NumSubsets; I++ )
{
    if ( Subset[I] can be rendered in hardware )    RenderSubset[I];
}
```

In this first pass we have looped through every subset and only render subsets that are not influenced by more matrices than the hardware supports. In the next pass, we will do the same again only this time with software vertex processing enabled. However, because we have stored the index of the first non-hardware compliant subset, we do not have to start testing from the beginning. We can start from this index because we know that no subsets exist before this index that cannot be rendered in hardware.

```
pDevice->SetSoftwareVertexProcessing ( TRUE );

for ( int I = iAttributeSoftware ; I < NumSubsets; I++ )
{
    if ( Subset[I] can NOT be rendered in hardware )    RenderSubset[I];
}
```

Fortunately, virtually all recent video cards support more than two matrix blends in hardware, so this will only be an issue on fairly old graphics cards.

11.8.5 ID3DXSkinInfo::ConvertToBlendedMesh

Before we look at our new CreateMeshContainer function for non-indexed skinning, we will examine the ID3DXSkinInfo::ConvertToBlendedMesh function. This function that will take the regular mesh we are passed to our CreateMeshContainer callback function and output a new mesh that has the correct number of weights in the vertices. This function will never return a mesh with more than three weights (four influences) defined for each vertex even if the bone to vertex data stored inside the ID3DXSkinInfo object has vertices that are mapped to more than four bones. In this case, the function will approximate the mesh as best as it can so that it meets this requirement. It will also try to take the current hardware into account so that it will return a mesh that is within the maximum matrix blend capabilities of the device. This was discussed in the previous section.

ConvertToBlendedMesh will be called from inside our CreateMeshContainer function. We will pass in the interface to the standard mesh that we were passed by D3DXLoadMeshHierarchyFromX. This function will use the ID3DXSkinInfo's internal bone and weight information to clone a new mesh into a format that can be vertex blended by the pipeline. The new mesh will still contain all of the old input mesh data such as vertex position, vertex normals, texture coordinates, etc., but now it will have additional information that allows it to use multiple device matrices for blending.

The parameter list might seem pretty overwhelming at first, but most of them are actually output parameters that we can store in our mesh container and use to render the mesh later. We have already covered most of them in the context of the mesh container structure earlier in the lesson.

```
HRESULT ID3DXSkinInfo::ConvertToBlendedMesh
(  
    LPD3DXMESH      pMesh,  
    DWORD           Options,  
    CONST LPDWORD   pAdjacencyIn,  
    LPDWORD         pAdjacencyOut,  
    DWORD           *pFaceRemap,  
    LPD3DXBUFFER    *ppVertexRemap,  
    DWORD           *pMaxVertexInfl,  
    DWORD           *pNumBoneCombinations,  
    LPD3DXBUFFER    *ppBoneCombinationTable,  
    LPD3DXMESH      *ppMesh  
);
```

LPD3DXMESH pMesh

This first parameter is a pointer to the source mesh interface passed into our CreateMeshContainer method by the D3DX loading function. This mesh is a standard ID3DXMesh that contains the skin geometry and it will be cloned into a blend-enabled mesh by this function. When this function returns and the mesh has been cloned, this mesh will no longer be needed and its interface can be released.

DWORD Options

This parameter is like the options flags used in the other mesh creation and cloning functions we have studied previously. It can be a combination of D3DXMESH flags and D3DXMESHOPT flags. Recall that the D3DXMESH flags allow us to choose properties such as which resource pool the vertex and index buffers of the mesh will be created in and whether or not this mesh should have the capability to be rendered with software vertex processing. The D3DXMESHOPT allows us to specify which optimizations should be applied to the vertex and index data of the cloned mesh. For example, the following flag combination would inform the function to create the vertex buffer and index buffer of the cloned mesh in the managed resource pool with vertex cache optimization. We know from previous lessons that this optimization actually performs the compact optimization, the attribute sort optimization, and will optimize the mesh to achieve better vertex cache coherency where possible.

D3DXMESH_MANAGED | D3DXMESHOPT_VERTEXCACHE

The optimizations will not be as effective as those applied to a regular mesh because now the triangles will also need to be batched into subsets based on bone contribution. We may have ten triangles which all share the same texture and material and, under normal circumstances, they would all end up in a single subset. But if these triangles do not all use the same N bones, they will be batched into additional subsets based on shared bones.

CONST LPDWORD pAdjacencyIn

This is a DWORD pointer that contains the adjacency information for the source mesh. We do not need to calculate this information ourselves since our CreateMeshContainer function will be passed the adjacency information of the source mesh by D3DXLoadMeshHierarchyFromX.

LPDWORD pAdjacencyOut [output]

Although we can pass NULL as this parameter, if we are interested in the adjacency information of the resulting mesh then we can pass in a pre-allocated DWORD array large enough to hold three DWORDs for every triangle in the source mesh. When the function returns, this array will be populated with the adjacency information of the newly cloned mesh. If you do pass NULL as this parameter, you can always generate the adjacency information for the output mesh at a later time using the ID3DXMesh::GenerateAdjacency method.

DWORD *pFaceRemap [output]

The pFaceRemap parameter should be large enough to hold one DWORD for every face in the source mesh. When the function returns, each element in the array will describe how the index of a face in the source mesh has been mapped to an index in the output mesh. If pFaceMap[5]=10 for example, this means that the 6th face in the original index buffer of the source mesh has now been moved to the 11th face slot in the index buffer of the output mesh. This gives us a chance to update any external structures that we may be maintaining that are linked to the original faces by index. You can set this pointer to NULL if you do not require the re-map information.

LPD3DXBUFFER *ppVertexRemap [output]

The ppVertexRemap parameter points to the address of an ID3DXBuffer interface which on function return will contain an array of DWORDs for each vertex in the source mesh describing how the vertex has been moved in the vertex buffer of the output mesh. The function will create this buffer during the call so we will not pre-allocate it. If you do not need this information then you can set this parameter to NULL.

DWORD *pMaxVertexInfl [output]

For this parameter we pass in the address of a DWORD which on function return will tell us the maximum number of bones that influence a single vertex in the mesh. This is the value that we will store in the NumInfl member of our mesh container. It is significant because subtracting one from this value will tell us how many weights each vertex has defined (we subtract one because the weight for the last influence is always calculated on the fly). If this value returns 4 for example, then it means the vertex structure of this mesh will have three weights defined and there will be at least one subset that needs access to all four matrices on the device. As mentioned previously, this does not mean that all vertices in the mesh need to use four matrices even though they will all store three weights. Any weights in the vertex that are not used will be set to 0.0 so that the corresponding matrix will not contribute.

This value is also significant because it tells us how large each BoneId array is in the D3DXBONECOMBINATION structure for each subset in our bone combination table. If this value is 3 for example, then we know that for every element in the bone combination table, each D3DXBONECOMBINATION structure will contain a BoneId array with three elements. Not all elements in the BoneId array for a given subset are necessarily used. If a given subset only needs to use two matrices, its BoneId array would still have 3 elements (in our current example) but the value of the third element would be set to UINT_MAX instead of a valid bone matrix index. This allows us to perform a rendering optimization by only setting the matrices that are used and setting the D3DRS_VERTEXBLEND render state appropriately so that no-op vertex matrix transformations are not carried out.

DWORD *pNumBoneCombinations [output]

There will be one D3DXBONECOMBINATION structure in this buffer for each subset in the output mesh. The bone combination buffer itself will be calculated by this function and returned to us so that we can store it away in our mesh container. This parameter is the address of a DWORD which on function return will tell us how many D3DXBONECOMBINATION structures are stored in the returned ID3DXBuffer (next parameter). Since there will be one D3DXBONECOMBINATION structure for each subset in the output mesh, this value will also tell us how many subsets exist in the output mesh, which we will need to know during rendering.

The value stored in this DWORD will be copied into the NumAttributeGroups member of our mesh container and will be used during rendering to loop through each subset that needs to be rendered.

LPD3DXBUFFER *ppBoneCombinationTable [output]

This parameter is the address of an ID3DXBuffer interface pointer. The function will allocate the ID3DXBuffer and on function return it will store the entire bone combination table for the mesh. For each subset in the output mesh (described by the previous parameter pNumBoneCombinations) there will be a D3DXBONECOMBINATION structure describing exactly which bones need to be set on the device in order to render that subset. These indices are stored inside each structure's BoneId array and they reference directly into the ppBoneMatrixPtrs and pBoneOffsetMatrices arrays in our mesh container. Each D3DXBONECOMBINATION structure also contains the ranges of vertices and indices in the output mesh vertex and index buffers which belong to a particular subset. The D3DXBONECOMBINATION table also stores an AttributeId which indexes into the D3DXMATERIAL buffer that was passed into CreateMeshContainer.

LPD3DXMESH *ppMesh [output]

The final parameter to this method is the address of an ID3DXMesh interface pointer. On function return, it will point to a mesh that will have pMaxVertexInfl weights defined in each vertex, where each weight will have been assigned its correct value. This is the mesh that we will use for rendering, and as such, the source (input) mesh can be released since it will no longer be needed.

11.8.6 The CreateMeshContainer Function

The first part of the mesh container creation function is completely unchanged from the software skinning version, so we will not need to explore it in much detail here. The following code stores the passed mesh pointer in a newly allocated mesh container (temporary, as this will be replaced with the converted blended mesh) and also copies the name of the mesh passed into the function. Once again, our example will assume that we will need vertex normals in our mesh so that it can be correctly lit by the pipeline. If normals are not present, then we clone the mesh to make space for vertex normals and then calculate them thereafter.

```
HRESULT CAllocateHierarchy::CreateMeshContainer
( LPCTSTR Name, LPD3DXMESHDATA pMeshData, LPD3DXMATERIAL pMaterials,
  LPD3DXEFFECTINSTANCE pEffectInstances, DWORD NumMaterials,
  DWORD *pAdjacency,
  LPD3DXSKININFO pSkinInfo, LPD3DXMESHCONTAINER *ppNewMeshContainer )
{
    D3DXMESHCONTAINER_DERIVED *pMeshContainer = NULL;
    UINT iBone, cBones;
    LPDIRECT3DDEVICE9 pd3dDevice = NULL;
    LPD3DXMESH pDestinationMesh = NULL;
    *ppNewMeshContainer = NULL;

    // Get a pointer to the mesh of the skin we have been passed
    pDestinationMesh = pMeshData->pMesh;

    // Allocate a mesh container to hold the passed data.
    // This will be returned from the function
    // where it will be attached to the hierarchy.
    pMeshContainer = new D3DXMESHCONTAINER_DERIVED;
    memset(pMeshContainer, 0, sizeof( D3DXMESHCONTAINER_DERIVED ));

    // If this mesh has a name then copy that into the mesh container too.
    if ( Name ) pMeshContainer->Name = _tcsdup( Name );

    // Get a pointer to the d3d device
    pDestinationMesh->GetDevice(&pd3dDevice);

    // if no normals exist in the mesh, add them ( we might need to use lighting )
    if ( !(pDestinationMesh->GetFVF() & D3DFVF_NORMAL) )
    {
        pMeshContainer->MeshData.Type = D3DXMESHTYPE_MESH;

        // Clone the mesh to make room for the normals
        pDestinationMesh->CloneMeshFVF( pMesh->GetOptions(),
                                         pMesh->GetFVF() | D3DFVF_NORMAL,
```

```

        pd3dDevice,
        &pMeshContainer->MeshData.pMesh );

    pDestinationMesh = pMeshContainer->MeshData.pMesh;

    // Now generate the normals for the pmesh
    D3DXComputeNormals( pDestinationMesh, NULL );
}
else // if normals already exist, just add a reference
{
    pMeshContainer->MeshData.pMesh = pDestinationMesh;
    pMeshContainer->MeshData.Type = D3DXMESHTYPE_MESH;
    pDestinationMesh->AddRef();
}

```

We now have a regular mesh stored in the mesh container (which is correct because we must remember that the same X file could contain skinned meshes and regular meshes) and at this point we would usually parse the passed texture and material information and store it in some meaningful way. We have already covered how to do this in previous lessons so will not show it here. You can check the accompanying workbook for this lesson for more details.

If we have been passed a valid ID3DXSkinInfo interface, then it means that the passed mesh is to be used to skin the hierarchy (or a section of it) and we will need to convert the passed mesh into a mesh that includes bone weights.

The first section of the code that is executed when we have been passed a skinned mesh is also unchanged -- we store the ID3DXSkinInfo pointer in our mesh container for later use since we will need it after the hierarchy is loaded to map bone offset matrices to absolute bone matrices. We then increase the reference count because we have made a copy of the interface, and we retrieve the bone offset matrices and store them in the mesh container. All of this is unchanged from the software skinning case.

```

// If there is skinning information,
// save off the required data and then set up for skinning
if (pSkinInfo != NULL)
{
    // first save off the SkinInfo and original mesh data
    pMeshContainer->pSkinInfo = pSkinInfo;
    pSkinInfo->AddRef();

    // We now know how many bones this mesh has
    // so we will allocate the mesh containers
    // bone offset matrix array and populate it.
    NumBones = pSkinInfo->GetNumBones();
    pMeshContainer->pBoneOffsetMatrices = new D3DXMATRIX [NumBones];

    // Get each of the bone offset matrices so that
    // we don't need to get them later
    for (iBone = 0; iBone < NumBones; iBone++)
    {
        pMeshContainer->pBoneOffsetMatrices[iBone] =
            *(pMeshContainer->pSkinInfo->GetBoneOffsetMatrix(iBone));
    }
}

```

The next section is where things change compared to software skinning. It is time to convert the mesh that we were passed into a blend-enabled mesh using `ID3DXSkinInfo::ConvertToBlendedMesh`. We currently have the source mesh stored in the `MeshData` member of the mesh container. This is where we want the resulting mesh to be stored, so we copy the source mesh pointer to a temporary `SourceMesh` pointer.

```
ID3DXMesh * pSourceMesh = pMeshContainer->MeshData.pMesh;
pMeshContainer->MeshData.pMesh=NULL;

pMeshContainer->pSkinInfo->ConvertToBlendedMesh
(
    pSourceMesh,
    D3DXMESH_MANAGED|D3DXMESHOPT_VERTEXCACHE,
    pAdjacency,
    NULL, NULL, NULL,
    &pMeshContainer->NumInfl,
    &pMeshContainer->NumAttributeGroups,
    &pMeshContainer->pBoneCombinationBuf,
    &pMeshContainer->MeshData.pMesh
);

pSourceMesh->Release ();
```

We pass in the source mesh and request that the resulting mesh be created as a managed resource which has the vertex cache optimization applied to it. We also pass in the adjacency information that was passed to `CreateMeshContainer` function and pass `NULL` for the following three parameters (we will have no need for the adjacency, vertex re-map, or face re-map information that this function generates for the output mesh). In the mesh container's `NumInfl` member will be stored the maximum number of bones that influence a single vertex in the mesh (which also tells us indirectly how many weights are stored in each vertex) and in the `NumAttributeGroups` member we store the number of subsets in the output mesh. The bone combination buffer is stored in the mesh container's `pBoneCombinationBuf` member and the converted output mesh is stored in the mesh container's `MeshData.pMesh` member.

Finally, we release the source mesh; we will be using the new mesh to render from this point forward because it contains the bone weight information the vertex blending module of the transformation pipeline needs for proper transformations.

At this point we have a mesh that contains the correct number of weights, but it is possible that some subsets in the output mesh may require the use of more matrices than is supported by the hardware. Recall that when the device is in software vertex processing mode we will always be able to render using up to four matrix influences. So taking this into account, before we return from the `CreateMeshContainer` function, we will loop through each subset in the output mesh and examine each subset's bone combination table. For each subset, we will examine the `BoneId` array of its `D3DXBONECOMBINATION` structure to count how many bone influences affect it. As soon as we find a subset that has more bone influences than the hardware supports, we will store the index of this subset in the mesh container's `iAttributeSoftware` member. All subsets up to this index can be rendered using hardware vertex processing and all subsets after that index (itself included) *may* have to be rendered with software vertex processing.

We start by getting a pointer to the mesh container's bone combination table which is stored inside an ID3DXBuffer:

```
LPD3DXBONECOMBINATION rgBoneCombinations =  
(LPD3DXBONECOMBINATION) (pMeshContainer->pBoneCombinationBuf->GetBufferPointer());
```

We will now loop through each subset in the mesh:

```
for (pMeshContainer->iAttributeSW = 0;  
     pMeshContainer->iAttributeSW < pMeshContainer->NumAttributeGroups;  
     pMeshContainer->iAttributeSW++)  
{
```

For each subset we will loop through each element of its BoneId array which contains bone matrix indices that are used by this subset. If for example, the NumInfl member returned by the ConvertToBlendedMesh equals 4, then it means that the BoneId array will be 4 elements long (i.e., large enough to hold four matrix indices). If a particular subset only uses two bone matrices, the third and fourth elements will be set to UINT_MAX to indicate that they are unused. We can use this fact to count how many matrix blends the pipeline will need to perform to render this subset, as shown below.

```
    DWORD cInfl = 0;  
    for (DWORD iInfl = 0; iInfl < pMeshContainer->NumInfl; iInfl++)  
    {  
        if (rgBoneCombinations[pMeshContainer->iAttributeSW].BoneId[iInfl]  
            != UINT_MAX)  
        {  
            ++cInfl;  
        }  
    }
```

At this point in the code, the DWORD local variable cInfl will tell us how many matrix blends the current subset we are checking will require. If this is larger than the maximum number of matrix blends supported by the hardware then we will break from the loop since we have just found the first subset that the hardware will not support. The index of this subset will be stored in the mesh container's iAttributeSoftware member because this was used as the loop variable to iterate through each subset.

```
        if (cInfl > m_d3dCaps.MaxVertexBlendMatrices)  
        {  
            break;  
        }  
    }
```

The iAttributeSoftware variable will now contain either the index of the first subset that cannot be rendered using hardware vertex processing or it will be equal to the number of subsets in the mesh if a non-hardware compliant subset was not found and the loop continued to completion. So if iAttributeSoftware is not equal to the number of subsets in the mesh then it means that part of this mesh will need to be rendered using software vertex processing.

Note however that the current mesh cannot be rendered using software vertex processing because when we create a mesh which we intend to use with software vertex processing on a mixed-mode device, we

must use the D3DXMESH_SOFTWAREPROCESSING mesh creation flag (which we did not do previously). Therefore, if this mesh needs to be software vertex processed, we will need to clone the mesh using this flag so that the resulting mesh can be properly rendered.

```
if (pMeshContainer->iAttributesSW < pMeshContainer->NumAttributeGroups)
{
    LPD3DXMESH pMeshTmp;
    pMeshContainer->MeshData.pMesh->CloneMeshFVF
    (
        D3DXMESH_SOFTWAREPROCESSING|pMeshContainer->MeshData.pMesh->GetOptions(),
        pMeshContainer->MeshData.pMesh->GetFVF(),
        m_pd3dDevice,
        &pMeshTmp
    );

    pMeshContainer->MeshData.pMesh->Release();
    pMeshContainer->MeshData.pMesh = pMeshTmp;
    pMeshTmp = NULL;
}
} // end if ID3DXSkinInfo!=NULL
} // End Function
```

The code clones the mesh into a temporary mesh and then releases the old mesh before assigning the newly cloned software processing enabled mesh to the mesh container's MeshData.pMesh member.

That brings us to the end of our non-indexed version of the CreateMeshContainer function. When D3DXLoadMeshHierarchyFromX returns control back to our application, the hierarchy will be loaded and all of our skinned meshes will be stored in their respective mesh containers along with their bone combination tables. Notice how the CreateMeshContainer function no longer requires the code that was used in the software skinning case that allocated (and resized) the temporary global matrix array. That array was only used to pass matrices into the UpdateSkinnedMesh function, which is not used when performing hardware skinning.

Do not forget that after D3DXLoadMeshHierarchyFromX returns control back to your program, you will still need to traverse the hierarchy and store the absolute bone matrices of each frame in the mesh container's ppBoneMatrixPtrs array, just as we did in the software skinning technique. You will recall that we wrote a function to do this called SetupBoneMatrixPointers and this function is unchanged regardless of the skinning technique used.

We have now seen everything there is to see with regards to setting up the hierarchy and storing the skinned meshes in a format which can be used by the non-indexed vertex blending module of the pipeline.

11.8.7 Transforming and Rendering the Skin

Animating the hierarchy is no different from the software skinning case (or in fact any animation of the frame hierarchy) -- we use the animation controller's `AdvanceTime` method to apply the animation data to the relative bone matrices of the hierarchy and then perform a pass through the hierarchy to recalculate the absolute matrix for each frame. So the next and final stage of the non-indexed skinning technique we need to look at is the code that would be used to actually render the mesh. We already know how to recursively traverse the hierarchy and call the `DrawMeshContainer` function for each mesh container found, so we will focus on this `DrawMeshContainer` function, just as we did in the software skinning technique.

Our drawing function in this example is passed a pointer to the mesh container to be rendered and a pointer to the owner frame. This frame is important if this is not a skinned mesh because this frame's combined matrix will be the matrix we wish to use as the world matrix to render the mesh.

The first thing this function does is cast the passed mesh container and frame to our derived class types.

```
void DrawMeshContainer (LPD3DXMESHCONTAINER pMeshContainerBase,
                      LPD3DXFRAME pFrameBase)
{
    D3DXMESHCONTAINER_DERIVED*pMeshContainer =
        (D3DXMESHCONTAINER_DERIVED*)pMeshContainerBase;

    D3DXFRAME_MATRIX *pFrame = (D3DXFRAME_MATRIX*)pFrameBase;

    UINT NumBlend;
    UINT iAttrib;
    LPD3DXBONECOMBINATION pBoneComb;
    UINT iMatrixIndex;
    D3DXMATRIXA16 matTemp;
    IDirect3DDevice9 * pDevice;

    pMeshContainer->MeshData.pMesh->GetDevice (&pDevice);
```

This function can be called for both skinned meshes and regular meshes so we will need to make sure (as we did in the software skinning version of this function) that we cater to both types. Because we are covering each of the skinning techniques in isolation for the purpose of clarity, in this version of the code we are assuming that if the `ID3DXSkinInfo` pointer stored in the mesh container is not `NULL`, then the mesh stored here is setup to be used with the non-indexed skinning technique.

The first thing we do is fetch the bone combination table of the mesh container so that we can get the matrix indices needed to render each subset.

```
// first check if this is a skinned mesh or just a regular mesh
if (pMeshContainer->pSkinInfo != NULL)
{
```

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

Our next task will be very similar to the code we used to calculate the index of the `iAttributeSoftware` flag. For each subset we will first count how many matrices it requires for rendering. We will consider this the first pass over the mesh where we draw all subsets that can be rendered using hardware vertex processing.

```
// Draw all subsets that can be rendered with Hardware Processing First
for (iAttrib = 0; iAttrib < pMeshContainer->NumAttributeGroups; iAttrib++)
{
    NumBlend = 0;
    for (DWORD i = 0; i < pMeshContainer->NumInfl; ++i)
    {
        if (pBoneComb[iAttrib].BoneId[i] != UINT_MAX)
        {
            NumBlend = i;
        }
    }
}
```

For each subset we will check to see whether or not the number of matrices that influence the subset (stored in the `NumBlend` local variable) is within the capabilities of the hardware. If so, we will fetch the matrix indices from this subset's `BoneId` array and use them to fetch the corresponding bone matrix and bone offset matrix stored in the mesh container. We will combine these two matrices to create a single world matrix and set that matrix in the relevant position in the device matrix palette (indices 0 through 3). The position of the index in the `BoneId` array describes which of the four matrix slots on the device the matrix should be bound to, as shown below.

```
// Can we render this subset in hardware?
if (m_d3dCaps.MaxVertexBlendMatrices >= NumBlend + 1)
{
    // Fetch each bone matrix for this subset.
    // Combine with its offset matrix, and set it
    for (DWORD i = 0; i < pMeshContainer->NumInfl; ++i)
    {
        iMatrixIndex = pBoneComb[iAttrib].BoneId[i];
        if (iMatrixIndex != UINT_MAX)
        {
            D3DXMatrixMultiply( &matTemp,
                                &pMeshContainer->pBoneOffsetMatrices[iMatrixIndex],
                                pMeshContainer->ppBoneMatrixPtrs[iMatrixIndex] );

            pDevice->SetTransform( D3DTS_WORLDMATRIX( i ),
                                  &matTemp );
        }
    }
} // End for each matrix influence
```

At this point, the matrices are correctly set on the device for the current subset we are about to render and the local `NumBlend` variable tells us how many matrices have been set for this subset. This allows us to optimize the rendering by setting the `D3DRS_VERTEXBLEND` to process only the required number of matrices, instead of wastefully processing every weight defined in the vertex. We finally render the current subset (and repeat for every subset in the mesh).

```

        pDevice->SetRenderState(D3DRS_VERTEXBLEND, NumBlend );

        // lookup the material used for this subset of faces
        Set Textures and Materials Here ( Not Shown )

        pMeshContainer->MeshData.pMesh->DrawSubset(iAttrib);

    } // End can be rendered in hardware

} // End for each attribute

```

At this point we will have looped through each subset and will have rendered all subsets that can be rendered using hardware vertex processing. However, some subsets may not have been rendered in the above loop if they required more matrix blends than the hardware supported. Therefore, we will now loop through the subsets and basically do what the above code did all over again. This time however, we will enable software vertex processing and render only the subsets that have more bone influences than the hardware supports. This is the second pass we discussed earlier in the lesson. Because we have stored (in the mesh container's `iAttributeSoftware` member) the index of the first subset that exceeds the hardware capabilities, we do not have to loop through every subset this time -- we just have to start from this index.

```

// If necessary draw any subsets that the hardware could not handle
if (pMeshContainer->iAttributeSW < pMeshContainer->NumAttributeGroups)
{
    pDevice->SetSoftwareVertexProcessing(TRUE);

    for ( iAttrib = pMeshContainer->iAttributeSW;
          iAttrib < pMeshContainer->NumAttributeGroups;
          iAttrib++)
    {
        NumBlend = 0;
        for (DWORD i = 0; i < pMeshContainer->NumInfl; ++i)
        {
            if (pBoneComb[iAttrib].BoneId[i] != UINT_MAX)
            {
                NumBlend = i;
            }
        }

        if (m_d3dCaps.MaxVertexBlendMatrices < NumBlend + 1)
        {
            // Fetch each bone matrix for this matrix.
            // Combine with its offset matrix. And set it
            for (DWORD i = 0; i < pMeshContainer->NumInfl; ++i)
            {
                iMatrixIndex = pBoneComb[iAttrib].BoneId[i];
                if (iMatrixIndex != UINT_MAX)
                {
                    D3DXMatrixMultiply( &matTemp,
                                         &pMeshContainer->pBoneOffsetMatrices[iMatrixIndex],
                                         pMeshContainer->ppBoneMatrixPtrs[iMatrixIndex] );
                }
            }

            pDevice->SetTransform( D3DTS_WORLDMATRIX( i ),

```

```

                                &matTemp );
        }
    }

    m_pd3dDevice->SetRenderState(D3DRS_VERTEXBLEND, NumBlend );

    // Set Textures and Materials for the current Subset Here

    // draw the subset -- correct material/matrices are loaded
    pMeshContainer->MeshData.pMesh->DrawSubset(iAttrib);

    } // end if can be rendered in hardware
} // end for each subset

pDevice->SetSoftwareVertexProcessing(FALSE);

} // end if software vertex processing subsets exist in this mesh

```

Finally, we disable vertex blending just in case any other objects in our scene need to be rendered.

```

    pDevice->SetRenderState(D3DRS_VERTEXBLEND, 0);
} // end if this is a skinned mesh

```

If this was not a skinned mesh, then we render the mesh normally as shown below.

```

else // this is just a regular mesh so draw normally
{
    pDevice->SetTransform(D3DTS_WORLD, &pFrame->mtxCombined);
    DrawMesh ( pMeshContainer->MeshData.pMesh )
}

pDevice->Release();

} // end function

```

And that is all there is to hardware non-indexed skinning. Although the code is fairly long, this is only because we repeat a lot of it to handle the two separate passes.

11.8.8 Non-Indexed Skinning Summary

Non-indexed Skinning: Advantages

- Vertex blending is performed in hardware by the GPU, making it faster than software skinning.
- We have the ability to optimize rendering of a subset by setting the D3DRS_VERTEXBLEND renderstate so that only the influential matrices are used. This

avoids the need for the pipeline to factor in matrix calculations that have zero weights which would not contribute towards the final world space position of the vertex.

- This form of hardware skinning is very well supported even on older T&L graphics hardware, although some hardware may not support all four possible matrix blends. Even if you do decide to support indexed skinning (discussed next) you should always implement this skinning technique as a fallback option because indexed skinning is not as widely supported in hardware.
- This skinning method can be used even if hardware support is not available as long as we create either a software or mixed-mode device. In this case, the vertex blending will be done in the software vertex blending component of the pipeline. This means that we can implement just this technique and have it work even when no hardware support is available for vertex blending.

Non-indexed Skinning: Disadvantages

- Only four bone influences per vertex (although this is usually more than enough).
- Because there is no information within each vertex specifying how each weight in the vertex maps to matrices in the device matrix palette, an implicit mapping is used where each of the four weights map to the first four matrices in the palette. This means that we can only use four out of the 256 matrix slots on the device. This results in the mesh being broken into many more subsets because each subset can only contain triangles that are influenced by the same four bone matrices.
- Only four bone influences per subset.
- Batch rendering severely compromised.

11.9 DirectX Skinning III: Indexed Skinning

The final skinning technique we will cover in this chapter is indexed skinning. Just like the non-indexed skinning technique, the geometry blending mechanism used is integral to the DirectX transformation pipeline. Indexed skinning is the most desirable skinning technique because it inherits the best aspects from both software skinning and pipeline skinning without the limitations of the non-indexed approach. Unfortunately, indexed skinning is not as widely supported in older hardware, but because it is part of the DirectX transformation pipeline, even if hardware support is not available, we can still perform this skinning technique with software vertex processing (assuming we have created a device for which software vertex processing is applicable).

On the bright side, indexed skinning works almost identically to its non-indexed cousin as far as implementation details go. So we will only have to cover a few changes to add indexed skinning support to our current code base.

In the previous section we learned that one of the biggest disadvantages of using non-indexed skinning was that we can only use four out of the 256 possible device palette world matrix slots. This was because each vertex in the blend-enabled mesh contains a number of weights, but no information describing which matrix slots on the device those weights are applicable to. The result was that a one to one mapping is implied, such that weights 0 – 3 in the vertex are always used to weight the contributions of world matrices 0 – 3 on the device (actual indices = 256 – 259). So even though the device supports 256 world matrices, any subset that we render could only consist of vertices that used the first four. One undesirable side effect was that batching was seriously compromised. Even if the entire mesh used only a single texture and material (which would usually cause all faces to be grouped into a single subset), the mesh would still have to be broken into multiple subsets, where each subset contained only the triangles that share the same four bone matrices.

When we think about the non-indexed case for a moment and lament the wasted matrices in the device matrix palette, we soon realize that our problems could all be solved if we had the ability to store matrix indices in each vertex. For example, if a vertex has N weights, then we would need to store N+1 matrix indices (because we recall that the N+1 weight is calculated on the fly in the pipeline). If we could do this, then when the pipeline transforms a vertex, it can examine each weight and its corresponding matrix index (the index would be a value between 0 and 255 describing which matrix in the device matrix palette the weight is defined for) and use the correct matrix to transform the vertex. If we had a mesh with 50 bones where all faces used the same texture, we could have a single subset because we would then have the ability to set all 50 bone matrices in the matrix palette simultaneously before rendering the mesh. Thus our batching optimizations are left intact. And if, practically speaking, each subset was no longer limited by how many matrices it can use and we consider that a vertex can still have four bone influences, then any given triangle could actually be manipulated by up to 12 unique bone matrices.

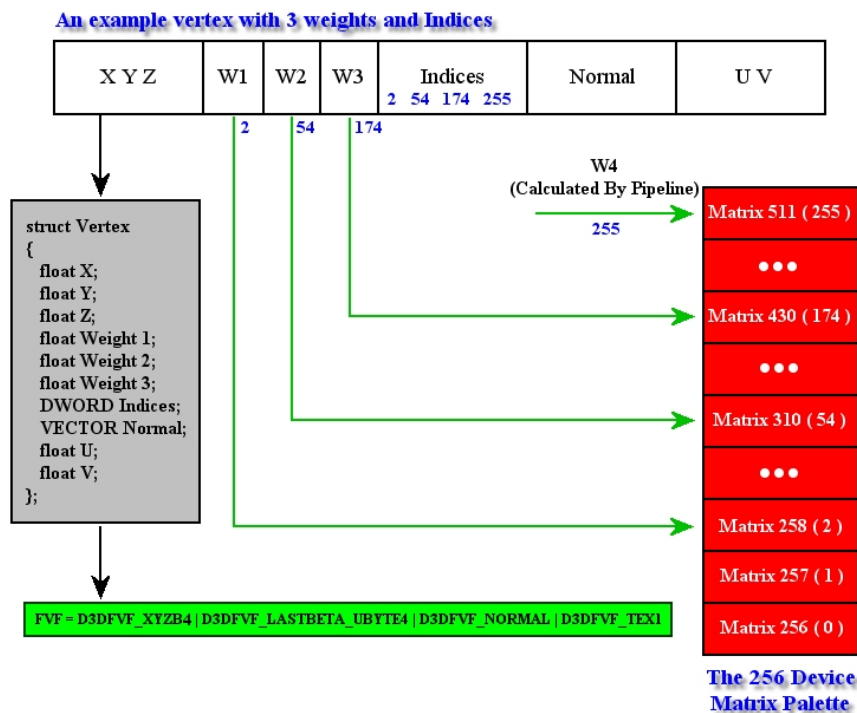


Figure 11.21

It just so happens that there is an FVF flag that allows us to place up to four matrix indices in our vertex structure in addition to the vertex weights. In Fig 11.21, the vertex has three weights (i.e. four bone influences) along with a new DWORD member called Indices. In each of the four bytes of the DWORD the index of one of the weights is stored. This tells the pipeline which matrix in the palette should be used with a given weight in the vertex to transform the mesh. In the example above, you can see that the vertex is to be transformed by the matrices stored in world matrix slots {2, 54, 174, 255}. These are the matrices used to transform this vertex for weights 1 through 4 respectively. We can imagine how another vertex in the same triangle might have a completely different set of indices, and this would hold true for every other vertex within the subset. Before rendering a subset, we just need to set all of the matrices that will be used by it (possibly even the entire bone hierarchy) in their respective matrix slots and then draw it.

11.9.1 Storing Matrix Palette Indices

Before continuing, we will take a look at the FVF flags that must be used to describe a vertex structure that includes both bone weights and matrix palette indices. These indices will take the form of the four individual bytes of a single DWORD. You would normally need to know these FVF flags when creating the vertex buffer that will store the indexed skin, but as it happens we can use a method of the ID3DXSkinInfo interface to work this out on our behalf. Like the ConvertToBlendedMesh function that we studied previously, which clones a normal mesh into one that includes weights, there is a second method that we will use for indexed skinning called ConvertToIndexedBlendedMesh. It does virtually the same thing as its sibling, only this time returns a mesh that has weights and indices defined in each vertex. Therefore, setting up the mesh for the indexed skinning technique is essentially identical to the non-indexed case. The exception is that we will call ConvertToIndexedBlendedMesh in our CreateMeshContainer function instead of the non-indexed version. Nevertheless, we will still look at the correct FVF flags that would be needed to create a vertex buffer that supports both weights and matrix palette indices just in case you need to build the buffer by hand.

Just for review, a vertex structure that would allow for three weights (four bone influences) is shown below along with the accompanying FVF flags.

```
struct NonIndexVertex
{
    D3DXVECTOR3 Pos;
    float Weight1;
    float Weight2;
    float Weight3;
    D3DXVECTOR3 Normal;
};

FVF = D3DFVF_XYZB3 | D3DFVF_NORMAL;
```

Recall that D3DFVF_XYZB3 informs the pipeline that the vertex will have an untransformed XYZ position and three float weights defined. Following the weights in this example we have also defined a vertex normal.

If we want to use the same basic structure, but include a matrix index for each weight then we have:

```
struct IndexVertex
{
    D3DXVECTOR3 Pos;
    float Weight1;
    float Weight2;
    float Weight3;
    DWORD Indices;
    D3DXVECTOR3 Normal;
};
```

The FVF for this structure is slightly less intuitive. There is a new FVF flag being used along with a change to one of the others.

FVF = D3DFVF_XYZB4 | D3DFVF_LASTBETA_UBYTE4 | D3DFVF_NORMAL;

The first thing you will notice is that we are now describing the vertex structure as containing four weights instead of three (D3DFVF_XYZB4). Technically this is not true of course since we only used three weights, but there is a catch. Since a DWORD and a FLOAT are both four bytes each in size, the above vertex data structure could be rewritten as:

```
struct IndexVertex
{
    D3DXVECTOR3 Pos;
    float Weight1;
    float Weight2;
    float Weight3;
    float Indices;
    D3DXVECTOR3 Normal;
};
```

Or we could even write the structure like so:

```
struct IndexVertex
{
    D3DXVECTOR3 Pos;
    float Weights[4];
    D3DXVECTOR3 Normal;
};
```

So we can see that while we might *define* the indices member as a DWORD (in the first structure) following the three weights to increase programmer readability, as far as the pipeline is concerned we are just passing it another four bytes. Therefore, our structure now has *space* for four weights defined.

Note that since a vertex cannot be influenced by more than four matrices, defining four weights would actually break this rule because it says it is influenced by five matrices. So our example flag would not actually work at all in the non-indexed case. The secret to the whole process is in the use of the second FVF flag shown above: D3DFVF_LASTBETA_UBYTE4. This flag tells the pipeline that the last weight (a weight is also sometimes called a *beta*) will be treated differently. It informs the pipeline that if it has been instructed to blend using N blend weights in the vertex, the Nth blend weight should not be used as a blend weight, but instead should be used as a four byte matrix palette index container.

So our FVF set above works with the vertex type we saw in Fig 11.21 -- a vertex with three actual blend weights (four bone contributions) and an additional four byte area containing the indices of up to four matrices in the device palette.

However, it must be made absolutely clear that D3DFVF_LASTBETA_UBYTE4 does not say that the last weight *defined in the vertex* will be the weight that will be used as an index container. Instead it says that the last weight that the pipeline has been instructed to process during vertex blending will be used as the index container. In other words, the last weight (last beta) is determined by the D3DRS_VERTEXBLEND render state and not by the number of weights defined in the vertex format. For example, if the D3DRS_VERTEXBLEND renderstate is set to D3DVBF_2WEIGHTS so that the pipeline will only process two weights for the vertex (three bone influences), the last beta in this example would be the third weight defined in the vertex (weight[2]). This weight would be treated as a DWORD with 4-byte indices. This is true even if the vertex contained four weights. So it would not be correct in our example, where the index data would be stored in the fourth weight, not the third.

We can see then that the D3DRS_VERTEXBLEND render state will also have to be set appropriately before rendering the mesh so that the correct weights are used for blending and the correct weight is used to feed matrix index data to the pipeline. Thus, we can no longer use the optimization that we did in the non-indexed case where we toggled the D3DRS_VERTEXBLEND render state between subsets to rid ourselves of zero weight matrix contributions. If we store our indices in the fourth weight for example, we will need this to be interpreted as the Indices member for all vertices that we render as defined by our vertex structure, regardless of the fact that there may be no-ops that result. The loss of this optimization is more than made up for however by the massive gains made from having much larger subsets.

Here is another example of a structure that will be indexed blended using three bone contributions (i.e. two weights).

```
struct IndexVertex
{
    D3DXVECTOR3 Pos;
    float Weight1;
    float Weight2;
    DWORD Indices;
    D3DXVECTOR3 Normal;
};

FVF = D3DFVF_XYZB3 | D3DFVF_LASTBETA_UBYTE4 | D3DFVF_NORMAL;
```

Our final example is an indexed vertex that has two bone contributions (1 weight) and its corresponding flags.

```
struct IndexVertex
{
    D3DXVECTOR3 Pos;
    float Weight1;
    DWORD Indices;
    D3DXVECTOR3 Normal;
};

FVF = D3DFVF_XYZB2 | D3DFVF_LASTBETA_UBYTE4 | D3DFVF_NORMAL;
```

11.9.2 Determining Support for Indexed Skinning

Before creating our mesh to use indexed skinning, we will need to determine what level of support the hardware provides. If the device is in software vertex processing mode, then indexed skinning will always be supported and we will be able to access all 256 world matrices on the device. But when using hardware vertex processing, things are not so simple.

First, if the device supports indexed blending at all, it may only support a subset of the 256 possible matrices. For example, a hardware device might support indexed vertex blending but only use the first 64 world matrix device slots. This is information we will need to know because it affects the way `ConvertToIndexedBlendedMesh` calculates the skinned mesh. If only 64 matrix palette entries are available, then the mesh will need to be adjusted to work with this limitation. This will typically involve breaking the mesh into separate subsets so that no single subset requires more than 64 matrices to be set simultaneously in order to render it. Therefore, when we call the `ConvertToIndexedBlendedMesh` function from our `CreateMeshContainer` function to convert our mesh into an indexed skinning compliant mesh, we will also pass in the number of matrices that the current device can support so that the function can make the appropriate adjustments.

We can query the device indexed skinning support by checking the `MaxVertexBlendMatrixIndex` flag in the `D3DCAPS9` structure.

```
D3DDCAPS9 caps;
pDevice->GetCaps ( &caps );
DWORD MaxMatIndex = caps.MaxVertexBlendMatrixIndex;
if ( MaxMatIndex == 0 )
{
    //No Hardware Support Available, Enable Software Processing
}
```

The `MaxVertexBlendMatrixIndex` member contains the value of the maximum index into the matrix palette that the device supports. For example, if this value is set to 63 then it means that this device does support hardware skinning but the pipeline can only use world matrices 0 through 63. Therefore, a single subset in the mesh can be influenced by no more than 64 matrices. In this example, the `ConvertToIndexedBlendedMesh` function would create a skinned mesh in which every subset uses no more than the first 64 matrices. This might result in more subsets than would otherwise be needed.

If `MaxVertexBlendMatrixIndex` is set to 0, then indexed geometry blending is not supported by the hardware at all. In this case we can choose to either fallback to software vertex processing or use the non-indexed skinning technique discussed earlier.

11.9.3 Storing the Skinned Mesh

Let us now look at how we might declare our mesh container structure to store the information needed to perform indexed skinning. Note that the frame hierarchy is stored in exactly the same way as we saw with the previous skinning techniques; it is only the way we create and render the mesh that has changed.

```
struct D3DXMESHCONTAINER_DERIVED: public D3DXMESHCONTAINER
{
    DWORD                NumAttributeGroups;
    DWORD                NumInfl;
    LPD3DXBUFFER          pBoneCombinationBuf;
    D3DXMATRIX**          ppBoneMatrixPtrs;
    D3DXMATRIX*           pBoneOffsetMatrices;
    DWORD                NumPaletteEntries;
    bool                UseSoftwareVP;
};
```

There are two new members in our mesh container; the rest of the members have the same meaning they had in the non-indexed case.

DWORD **NumPaletteEntries**

This **DWORD** will store the size of the device matrix palette that is available for skinning. We will calculate this ourselves by querying the `D3DCAPS9::MaxVertexBlendMatrixIndex`. We will also have to factor in whether or not the mesh has vertex normals. As it happens, if the mesh does have vertex normals, then the number of matrices that the device can use is cut in half. If a device supports 200 matrices (`MaxVertexBlendMatrixIndex = 199`) then we can only use 100 of these matrices when the mesh has vertex normals because the normals will also have to be transformed. Therefore, if our mesh has normals, we will divide the `MaxVertexBlendMatrixIndex` value by two before passing it into `ConvertToIndexedBlendedMesh`. The `ConvertToIndexedBlendedMesh` function needs to know how many matrices are available for indexed skinning (i.e. the palette size) so that it can force the output mesh to use no more than this many matrices per subset. As discussed previously, if a subset needs to use more matrices than the device currently supports, the mesh will be subdivided into smaller subsets where each subset only uses the number of matrices available.

bool **UseSoftwareVP**

We will use this simple Boolean to indicate whether the device supports indexed skinning. If not, then we will have to place the device into software vertex processing mode in order to render it using the indexed skinning technique. When software vertex processing is used, the device will always support 256 world matrices (128 if the mesh has vertex normals).

11.9.4 The CreateMeshContainer Function

The first section of our updated CreateMeshContainer function is completely unchanged. We allocate the mesh container, copy the name of the mesh and, in this example, clone the source mesh if it does not contain vertex normals. This is not a requirement for skinning, but since we intend to use the DirectX lighting pipeline in our code, we will get this step out of the way now.

```
HRESULT CAllocateHierarchy::CreateMeshContainer
( LPCTSTR Name, LPD3DXMESHDATA pMeshData, LPD3DXMATERIAL pMaterials,
  LPD3DXEFFECTINSTANCE pEffectInstances, DWORD NumMaterials,
  DWORD *pAdjacency,
  LPD3DXSKININFO pSkinInfo, LPD3DXMESHCONTAINER *ppNewMeshContainer )
{
    D3DXMESHCONTAINER_DERIVED *pMeshContainer = NULL;
    UINT iBone, cBones;
    LPDIRECT3DDEVICE9 pd3dDevice = NULL;
    LPD3DXMESH pDestinationMesh = NULL;
    *ppNewMeshContainer = NULL;

    // Get a pointer to the mesh of the skin we have been passed
    pDestinationMesh = pMeshData->pMesh;

    // Allocate a mesh container to hold the passed data.
    // This will be returned from the function
    // where it will be attached to the hierarchy.
    pMeshContainer = new D3DXMESHCONTAINER_DERIVED;
    memset(pMeshContainer, 0, sizeof( D3DXMESHCONTAINER_DERIVED ));

    // If this mesh has a name then copy that into the mesh container too.
    if ( Name ) pMeshContainer->Name = _tcsdup( Name );

    // Get a pointer to the d3d device.
    pDestinationMesh->GetDevice(&pd3dDevice);

    // if no normals exist in the mesh, add them ( we might need to use lighting )
    if (!(pDestinationMesh->GetFVF() & D3DFVF_NORMAL))
    {
        pMeshContainer->MeshData.Type = D3DXMESHTYPE_MESH;

        // Clone the mesh to make room for the normals
        pDestinationMesh->CloneMeshFVF( pMesh->GetOptions(),
                                        pMesh->GetFVF() | D3DFVF_NORMAL,
                                        pd3dDevice,
                                        &pMeshContainer->MeshData.pMesh );

        pDestinationMesh = pMeshContainer->MeshData.pMesh;

        // Now generate the normals for the pmesh
        D3DXComputeNormals( pDestinationMesh, NULL );
    }
    else // if normals already exist, just add a reference
    {
        pMeshContainer->MeshData.pMesh = pDestinationMesh;
    }
}
```

```

    pMeshContainer->MeshData.Type = D3DXMESHTYPE_MESH;
    pDestinationMesh->AddRef();
}

```

We now have a regular mesh stored in the mesh container, so it is time to check to see if we have been passed skinning information. When this is the case, we will start off with the same approach as in the previous version of the function -- we will store the ID3DXSkinInfo interface and bone offset matrices in our mesh container.

```

// If there is skinning information, save off the required data
// and then set up for skinning
if (pSkinInfo != NULL)
{
    // first save off the SkinInfo and original mesh data
    pMeshContainer->pSkinInfo = pSkinInfo;
    pSkinInfo->AddRef();

    // We now know how many bones this mesh has
    // so we will allocate the mesh containers
    // bone offset matrix array and populate it.
    NumBones = pSkinInfo->GetNumBones();
    pMeshContainer->pBoneOffsetMatrices = new D3DXMATRIX [NumBones];

    // Get each of the bone offset matrices so that
    // we don't need to get them later
    for (iBone = 0; iBone < NumBones; iBone++)
    {
        pMeshContainer->pBoneOffsetMatrices[iBone] =
            *(pMeshContainer->pSkinInfo->GetBoneOffsetMatrix(iBone));
    }
}

```

Now things will start to get a little different. We need to know if the current hardware can perform indexed skinning as well as any matrix limitations that may exist. If the device does not support enough matrices for this mesh, then software vertex processing will have to be invoked later to render it. Imagine a device that supports only six matrices to use for indexing. At first, this might not seem like a problem because we know that a single vertex can only be influenced by four matrices at most. However, our DrawPrimitive rendering units in this case are triangles, not single vertices. So the smallest item we can batch contains three vertices. If each of these three vertices used different matrices, then that triangle would need access to 12 matrices in order to be transformed. This would be beyond the capabilities of our example device so we would have to use software vertex processing.

How do we know if a single triangle in this mesh uses more matrices than the hardware has to offer? It just so happens that ID3DXSkinInfo has a method that calculates the maximum number of bones that influence a single triangle in the mesh. This function is called GetMaxFaceInfluences and is shown next.

```

HRESULT GetMaxFaceInfluences
(
    LPDIRECT3DINDEXBUFFER9 pIB,
    DWORD NumFaces,
    DWORD *maxFaceInfluences
);

```

Because ID3DXSkinInfo only contains the bone-to-vertex information, it has no idea how those vertices are arranged into faces in the mesh. So we must pass in the index buffer followed by the number of triangles in the buffer so that this can all be worked out internally. The third parameter is the address of a DWORD which on function return will contain the maximum number of bones that influence a single triangle in the mesh. We can use this value to determine whether the current hardware can cope with this mesh.

The code that calls this function to fetch the maximum face influence of the mesh follows:

```

DWORD NumMaxFaceInfl;
DWORD Flags = D3DXMESHOPT_VERTEXCACHE;
LPDIRECT3DINDEXBUFFER9 pIB;
pMeshContainer->MeshData.pMesh->GetIndexBuffer(&pIB);
pMeshContainer->pSkinInfo->GetMaxFaceInfluences(
    pIB,
    pMeshContainer->MeshData.pMesh->GetNumFaces(),
    &NumMaxFaceInfl);
pIB->Release();

```

As long as we have 12 matrices in the device matrix palette, we have enough to render any triangle and thus the mesh can be rendered in hardware. If the above function returned more than twelve (unlikely), then ConvertToIndexedBlendedMesh will be able to approximate the mesh to fit. Therefore, we take either the value returned from the previous function or the value of 12 (whichever is less) and check that the hardware supports it. If not, then we will set the D3DXMESH_SOFTWAREPROCESSING flag when we create the skinned mesh and set the mesh container's UseSoftwareVP Boolean to TRUE so that we know we will need to enable software vertex processing during rendering.

In software vertex processing mode we will always have 256 matrices available, so we also calculate the palette entries we will need on the device. Remember that we will feed this value into the ConvertToIndexedBlendedMesh function and it will subdivide the mesh so that no subset uses more matrices than the NumPaletteEntries we specify. Of course, we want this palette size to be as large as possible (no larger than 256 though) so that the mesh can be grouped into larger subsets.

In the following code we calculate this by taking the minimum of 256 and the number of bones in the mesh. This means that if there are more bones in the mesh than 256, then the mesh will need to be subdivided to some degree so that no subset uses more than this amount. If however, the number of bones is less than 256 then these are all the palette entries we need. For example, if there are 128 bones in the mesh, then we will only need at most 128 matrix palette entries to achieve maximum batching efficiency (forgetting about the vertex normals issue for the moment). The number of palette entries is also stored in the mesh container's NumPaletteEntries member because we will need this later during rendering to know how many elements will be in every subset's BoneId array.

```

NumMaxFaceInfl = min(NumMaxFaceInfl, 12);

if (m_d3dCaps.MaxVertexBlendMatrixIndex + 1 < NumMaxFaceInfl){
    // HW does not support indexed vertex blending. Use SW instead
    pMeshContainer->NumPaletteEntries = min( 256,
        pMeshContainer->pSkinInfo->GetNumBones());
    pMeshContainer->UseSoftwareVP = true;
    Flags |= D3DXMESH_SOFTWAREPROCESSING;
}

```

The code above shows what happens when the device cannot support the maximum number of matrices that influence a single triangle in the mesh. The result is that we will need to switch over to software vertex processing at the appropriate time.

The following code demonstrates what happens when the hardware can support enough matrices to make sure that at least every triangle is rendered. Notice that in the hardware case we calculate the number of matrix palette entries we are going to use differently from the software case. This time we set the number of entries to either the number of bones in the mesh, or half the number of matrices the hardware can handle, whichever is less. The reason we divide the number of matrices the device can support by two is because we are assuming that the skin has normals. When the *hardware* is performing indexed skinning and the mesh has normals, half of the available matrix slots are reserved for vertex normal transformations. If the number of bones in the mesh is less than half the number of matrices supported by the device, then we can use this value since there is adequate space in the palette. If you are setting up a skin that does not contain normals, then the division by two will not be necessary and you will be able to use all of the matrix slots available on the device.

```
else{
    pMeshContainer->NumPaletteEntries =
        min( (m_d3dCaps.MaxVertexBlendMatrixIndex + 1) /2,
             pMeshContainer->pSkinInfo->GetNumBones() );
    pMeshContainer->UseSoftwareVP = false;
    Flags |= D3DXMESH_MANAGED;
}
```

At this point, we have everything we need to create the indexed blended mesh. The following code shows the call to `ConvertToIndexedBlendedMesh` which creates a skinned mesh that has both weights and indices stored at the vertex level. It also returns the bone combination table describing which subsets use which matrices.

```
ID3DXMesh * pSourceMesh = pMeshContainer->MeshData.pMesh;
pMeshContainer->MeshData.pMesh=NULL;
pMeshContainer->pSkinInfo->ConvertToIndexedBlendedMesh
(
    pSourceMesh,
    Flags,
    pMeshContainer->NumPaletteEntries,
    pMeshContainer->pAdjacency,
    NULL, NULL, NULL,
    &pMeshContainer->NumInfl,
    &pMeshContainer->NumAttributeGroups,
    &pMeshContainer->pBoneCombinationBuf,
    &pMeshContainer->MeshData.pMesh
);

    pSourceMesh->Release();
} // end if skin
} //end function
```

When this function returns, we will have a mesh stored in our hierarchy that is ready to be rendered using the pipeline indexed skinning techniques discussed previously.

11.9.5 ID3DXSkinInfo::ConvertToIndexedBlendedMesh

Let us finish off our discussion on creating indexed skinned meshes by looking at the ConvertToIndexedBlendedMesh method of the ID3DXSkinInfo object. This function is very similar to the ConvertToBlendedMesh function used to create the non-indexed skinned mesh in the previous section. The exception is one extra parameter where we specify to the function the number of matrix palette entries we have available for our device. This allows the function to manipulate the mesh such that no single subset uses more matrix influences than this.

```
HRESULT ConvertToIndexedBlendedMesh (  
  
    LPD3DXMESH      pMesh,  
    DWORD           Options,  
    DWORD           paletteSize,  
    CONST LPDWORD   pAdjacencyIn,  
    LPDWORD         pAdjacencyOut,  
    DWORD           *pFaceRemap,  
    LPD3DXBUFFER    *ppVertexRemap,  
    DWORD           *pMaxVertexInfl,  
    DWORD           *pNumBoneCombinations,  
    LPD3DXBUFFER    *ppBoneCombinationTable,  
    LPD3DXMESH      *ppMesh  
);
```

All of the parameters are the same as the ConvertToBlendedMesh call with the exception of the third parameter (paletteSize). Please refer back to our earlier discussions if you need a refresher on these other parameters.

DWORD paletteSize

This is the number of world matrices that are available for use on the current device. If we were to pass in a value of 64 for this parameter, then this function would subdivide the mesh into different subsets such that no single subset needs to be rendered using more than 64 matrices.

We have now covered everything needed to load and store the indexed skinned mesh. With the exception of one or two places, the code did not change much from the non-indexed case. When D3DXLoadMeshHierarchyFromX returns, the hierarchy will be loaded and will contain all of our indexed skinned meshes in their respective mesh containers.

Once again you are reminded that animating the hierarchy is identical in all of the skinning cases we have covered. We will call the animation controller's AdvanceTime method and then traverse the hierarchy to build the absolute bone matrices for each frame.

11.9.6 Transforming and Rendering the Skin

When we are ready to render our indexed skinned mesh we take the same approach as always – we traverse the hierarchy looking for mesh containers and call their drawing functions as they are encountered. This part is no different than what we have seen in the previous sections. So let us now look at what the DrawMeshContainer method would look like in the indexed skinning technique and then our work is done. This version of DrawMeshContainer is actually the smallest of all the techniques studied.

The first thing that this function does is cast the passed mesh container and frame to our derived class formats.

```
void DrawMeshContainer (LPD3DXMESHCONTAINER pMeshContainerBase,
                      LPD3DXFRAME pFrameBase)
{
    D3DXMESHCONTAINER_DERIVED*pMeshContainer=
        (D3DXMESHCONTAINER_DERIVED*)pMeshContainerBase;

    D3DXFRAME_MATRIX *pFrame = (D3DXFRAME_MATRIX*)pFrameBase;

    UINT NumBlend;
    UINT iAttrib;
    LPD3DXBONECOMBINATION pBoneComb;
    UINT iMatrixIndex;
    D3DXMATRIXA16 matTemp;
    IDirect3DDevice9 * pDevice;
    pMeshContainer->MeshData.pMesh->GetDevice (&pDevice);
```

After retrieving a copy of our device, we check to see if the mesh stored in this mesh container is a skinned mesh. If so, then we check to see if the mesh needs to be rendered using software vertex processing and enable it if necessary.

```
    if (pMeshContainer->pSkinInfo != NULL)
    {
        if (pMeshContainer->UseSoftwareVP)
        {
            pDevice->SetSoftwareVertexProcessing (TRUE);
        }
    }
```

Next we will set the D3DRS_VERTEXBLEND render state so that we inform the pipeline about how many weights our vertices will have. This tells the pipeline that the N+1 weight in our vertex structure is where the matrix indices are stored. We have this information stored in our mesh container's NumInfl member because it was returned by ConvertToIndexedBlendedMesh. If NumInfl is set to 1 then it means that although we wish to perform indexed blending, there are no weights in the vertices. In this case each vertex will be influenced by one matrix with an assumed weight of 1.0. Every vertex in a subset may still be influenced by a different matrix in the palette, even in this case when the vertex does not have weights (although it would still have the weight being used as matrix indices); we are still performing

indexed matrix transformations and a given triangle could be transformed by up to three matrices (one per vertex).

```
if (pMeshContainer->NumInfl == 1)
    pDevice->SetRenderState(D3DRS_VERTEXBLEND, D3DVBF_0WEIGHTS);
```

If NumInfl is not 1 then we can just subtract one from it to tell us the number of weights that our vertices use (remember that the number of weights is always one less than the number of influences). So for any cases other than NumInfl = 1 we can use this fact to map directly to the enumerated type for vertex blending.

```
else
    pDevice->SetRenderState(D3DRS_VERTEXBLEND, pMeshContainer->NumInfl - 1);
```

As a reminder, the flags of the D3DVERTEXBLEND_FLAGS enumerated type are assigned the following values:

```
D3DVBF_DISABLE = 0,
D3DVBF_1WEIGHTS = 1,
D3DVBF_2WEIGHTS = 2,
D3DVBF_3WEIGHTS = 3,
D3DVBF_TWEENING = 255,
D3DVBF_0WEIGHTS = 256
```

Having set the D3DRS_VERTEXBLEND render state to correctly describe our vertex format and the weights we wish to use for blending, our next job is to inform the pipeline that we wish to perform indexed vertex blending instead of the standard non-indexed blending. We do this using a device render state that we have not yet seen, called D3DRS_INDEXEDVERTEXBLENDENABLE. It will be set to TRUE or FALSE to enable or disable indexed vertex blending, respectively. So before we render our skin, we must enable it:

```
pDevice->SetRenderState(D3DRS_INDEXEDVERTEXBLENDENABLE, TRUE);
```

Next, we get a pointer to the mesh container's bone combination table which will contain all of the bone indices used by each subset.

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

For each subset we loop through the BoneId array of its bone combination structure. We have stored in the mesh container the size of the matrix palette being used by this mesh, so this is exactly how large the BoneId array of each subset will be. We fetch each matrix index from the subset's BoneId array and, assuming it does not contain UINT_MAX, this is a matrix slot that will be used by this subset. We use the index to fetch the bone matrix from the mesh container and combine it with its corresponding bone offset matrix and set the resulting matrix at its correct slot in the device matrix palette.

```
for (iAttrib = 0; iAttrib < pMeshContainer->NumAttributeGroups; iAttrib++)
{
    for ( iPaletteEntry = 0;
```

```

        iPaletteEntry < pMeshContainer->NumPaletteEntries;
        ++iPaletteEntry )
    {
        iMatrixIndex = pBoneComb[iAttrib].BoneId[iPaletteEntry];
        if (iMatrixIndex != UINT_MAX)
        {
            D3DXMatrixMultiply( &matTemp,
                                &pMeshContainer->pBoneOffsetMatrices[iMatrixIndex],
                                pMeshContainer->ppBoneMatrixPtrs[iMatrixIndex] );

            pDevice->SetTransform( D3DTS_WORLDMATRIX( iPaletteEntry ),
                                   &matTemp );
        }
    }
}

```

By the time we have exited the `iPaletteEntry` loop above, we will have set all of the matrices used by the current subset in their proper device palette slots. This might be only two matrices or might be over two hundred depending on how many triangles are in the subset and how many matrices need to be used to transform them.

We can now set the textures and material used by the subset (not shown here for simplicity).

```
//Set Textures and Material of subset here
```

Finally, we render the current subset with the correct palette of matrices that we have just set up.

```

        pMeshContainer->MeshData.pMesh->DrawSubset( iAttrib );
    } // end current attribute 'iAttrib'

} // end If skin Info != NULL

else // this is just a regular mesh so draw normally
{
    pDevice->SetTransform(D3DTS_WORLD, &pFrame->mtxCombined);
    DrawMesh ( pMeshContainer->MeshData.pMesh)
}

pDevice->Release();
} // end function

```

As with all other versions of this function, the last section of code is executed if the mesh stored in this mesh container is not a skinned mesh. In that case it is rendered normally, as discussed in previous lessons.

11.9.7 Indexed Skinning Summary

Indexed Skinning Advantages

- Significantly faster than all other skinning techniques because effective batch rendering is maintained when the hardware has a sizable matrix palette.
- Supported in software vertex processing mode if no hardware support exists.
- Possible 256 matrix influences per subset and 12 influences per triangle.

Indexed Skinning Disadvantages

- Not as widely supported as non-indexed, even on fairly recent 3D hardware.
- Still limited to a maximum of 4 matrix influences per vertex, whereas the software skinning technique has no limitation.

Conclusion

We have now covered all three skinning techniques in isolation and have even looked at a lot of basic code to implement them. For a complete implementation description you should now turn to your workbook that accompanies this lesson. It will integrate all three skinning methods into a single mesh class that will make for a nice plug-in to our application framework.

We are not quite finished with our skinning discussions just yet. Our focus in this lesson was almost entirely on how to load and store skins that were pre-generated and stored in X files. In the next chapter, we will use many of the ideas we learned about in this lesson to do things a little differently -- we will create skins, skeletons, and even animations all procedurally. The result will be a solid understanding of how the entire process works, and from a practical perspective, a nice tree class that we can use to liven up our outdoor scenes. Make sure you are comfortable with all of the code we cover in the workbook for this chapter as it will be expected that you understand it when you work on the lab projects in the next chapter.