Skinned Mesh Character Animation with Direct3D 9.0c

Frank Luna
www.moon-labs.com

Created on Monday, February 20, 2004
Update 1 on Friday, September 10, 2004

Real-Time character animation plays an important role in a wide variety of 3D simulation programs, and particularly in 3D computer games. This paper describes the data structures and algorithms used to drive a modern real-time character animation system. In addition, it presents a thorough examination of the D3DX 9.0c Animation API.

Section 1 describes the motion and data structural representation of a 3D character. Section 2 focuses on the datasets needed to describe an animation sequence. Section 3 examines an animation technique that works with rigid bodies and emphasizes the problems associated with this approach. Section 4 explains a new animation technique, vertex blending (also called skinned mesh animation), which does not suffer the problems of rigid body animation. Section 5 shows how to implement a skinned mesh character animation using the D3DX Animation API. Section 6 demonstrates how to play multiple distinct animation sequences. Section 7 explores how to create new animations from existing ones using the D3DX animation blending functionality. And finally, Section 8 explains how to execute code in parallel with an animation sequence, using the D3DX animation callback functionality.

1 An Overview of Character Mesh Hierarchies

Figure 1 shows a character mesh. The highlighted chain of bones in the figure is called a **skeleton**. A skeleton provides a natural underlying structure for driving a character animation system. The skeleton is surrounded by an exterior **skin**, which we model as 3D geometry (vertices and polygons). Each **bone** in the skeleton influences the shape and position of the skin, just like in real life; mathematically, bones are described by transformation matrices which transform the skin geometry appropriately. Thus, as we animate the skeleton, the attached skin is animated accordingly to reflect the current pose of the skeleton.

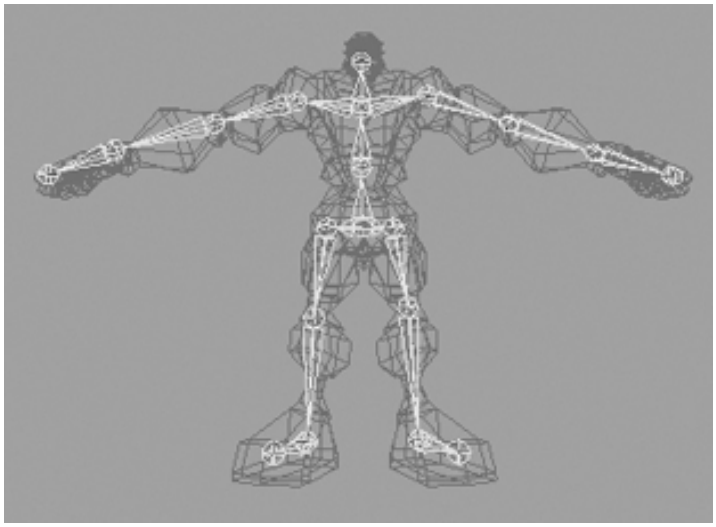

Figure 1: A Character mesh. The highlighted bone chain represents the character's skeleton. The dark colored polygons represent the character's skin.

1.1 Bones and Inherited Transforms

Initially, all bones start out in **bone space** with their joints coincident with the origin. A bone B has two associated transforms: 1) A **local transform L** and 2) a **combined transform C**. The local transform is responsible for rotating B in bone space about its joint (Figure 2*a*), and also for offsetting (translating) B relative to its immediate parent such that B's joint will connect with its parent bone (Figure 2*b*). (The purpose of this offset translation will be made clear in a moment.)

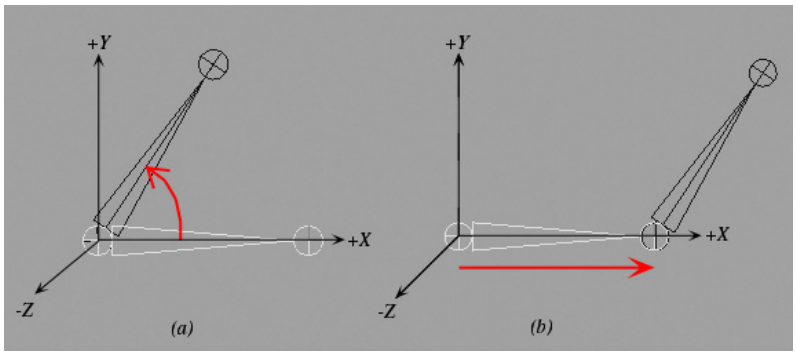

Figure 2: a) A bone rotates about its pivot joint in bone space. b) The bone is offset to make room for its parent bone.

In contrast to the local transform, the combined transform is responsible for actually posing the bone relative to the character in order to construct the character's skeleton, as Figure 3 shows. In other words, the combined transform transforms a bone from bone space to the character space. Therefore, it follows that the combined transform is the transform that is used to actually positions and shapes the skin in character space.

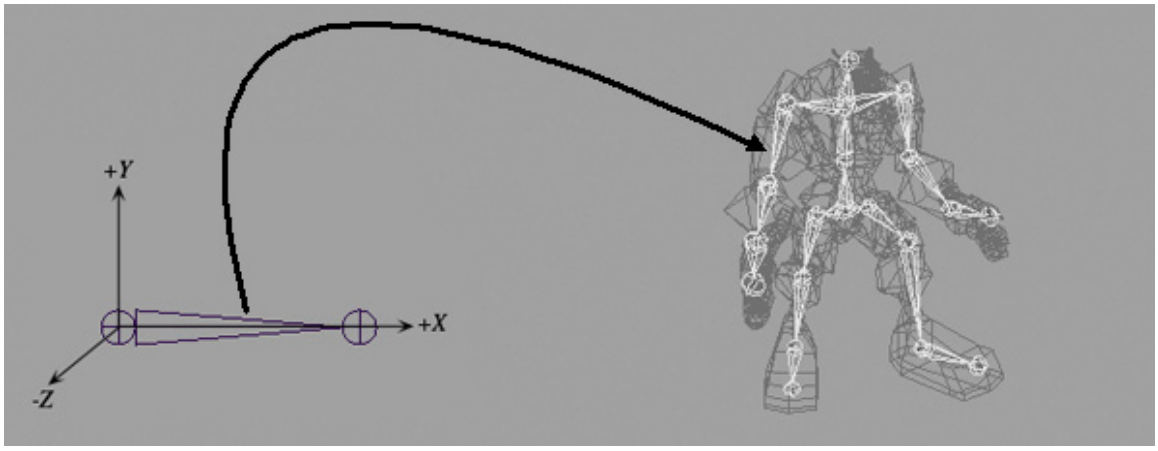

Figure 3: The combine transformation transforms the bone from bone space to character space. In this figure, the bone in bone space becomes the right upper-arm bone of the character.

So, how do we determine the combined transform? The process is not completely straightforward since bones are not independent of each other, but rather affect the position of each other. Ignoring rotations for the moment, consider the desired bone layout of an arm, as depicted in Figure 4.

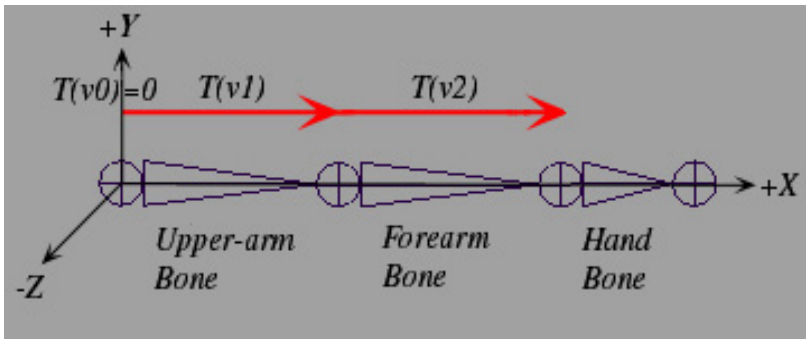

Figure 4: The skeleton of an arm. Observe how the combination of T(v0), T(v1) and T(v2) position the hand. Likewise, the combination of T(v0) and T(v1) position the forearm. And notice how T(v0) positions the upper-arm. (Actually T(v0) does nothing, since the upper-arm is the root bone it doesn't need to be translated, hence T(v0) = 0.)

Given an upper-arm, forearm, and hand bone in bone space, we need to find combined transforms for each bone that will position the bones in the configuration shown in Figure 4. Because the local transform of a bone offsets a bone relative to its parent, we can readily see from Figure 4 that a bone's position, relative to the character mesh, is determined by first applying its local translation transform, then by applying the local translation transform of *all* of its parents, in the order of youngest parent to eldest parent.

Now, consider the skeleton arm depicted in Figure 5. Physically, if we rotate the upper-arm about the shoulder joint, then the forearm and hand must necessarily rotate with it. Likewise, if we rotate the forearm, then just the hand must necessarily rotate with it. And of course, if we rotate the hand, then only the hand rotates. Thus we observe that a bone's position, relative to the character mesh, is determined by first applying its local

rotation transform, then by applying the local rotation transform of *all* of its parents, in the order of youngest parent to eldest parent.

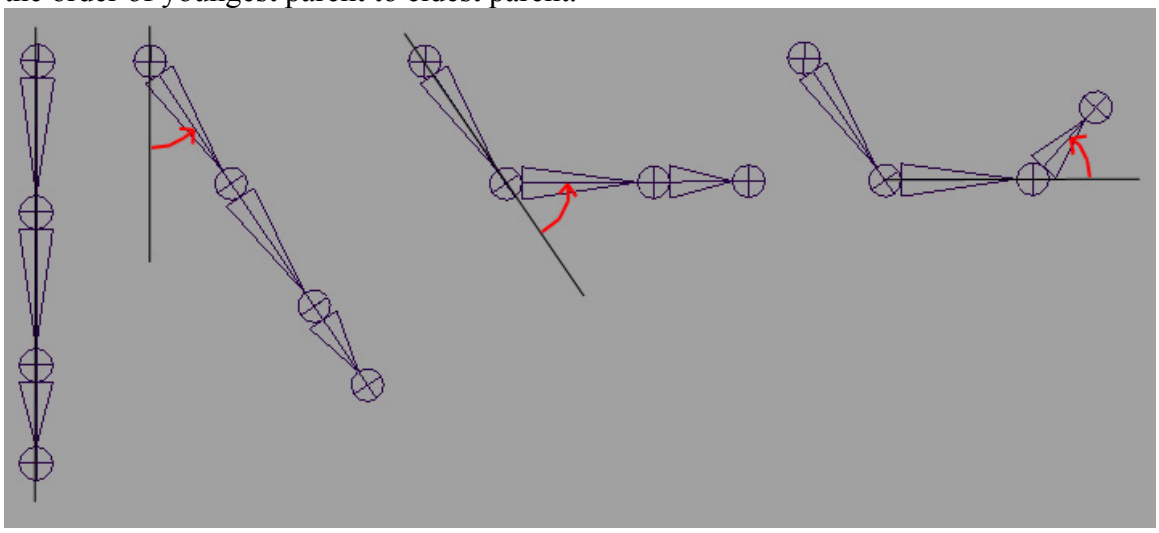

Figure 5: Hierarchy transforms. Observe that the parent transformation of a bone influences itself and all of its children.

Now that we see that both translations and rotations are inherited from each parent in a bone's **lineage**, we have the following: A bones **combined transform** is determined by first applying its local transform (rotation followed by translation), then by applying the local transform of its parent P', then by applying the local transform of its grandparent P', …, and finally by applying the local transform of its eldest parent $P^{(n)}$ (the root). Mathematically, the combined transformation matrix of the i^{th} bone $\mathbf{C}_i$ is given by:

(1) $$\mathbf{C}_i = \mathbf{L}_i \mathbf{P}_i,$$

where $\mathbf{L}_i$ is the local transformation matrix of the i^{th} bone, and $\mathbf{P}_i$ is the combined transformation matrix of the i^{th} bone's parent. Note that we multiply by the matrix $\mathbf{L}_i$ first, so that its local transform is applied first, in bone space.

1.2 D3DXFRAME

We now introduce a D3DX hierarchical data structure called `D3DXFRAME`. We will use this structure to represent the bones of the character. By assigning some pointers we can connect these bones to form the skeleton. For example, Figure 6 shows the pointer connection that form the bone hierarchy tree (skeleton) of the character showed in Figure 1.

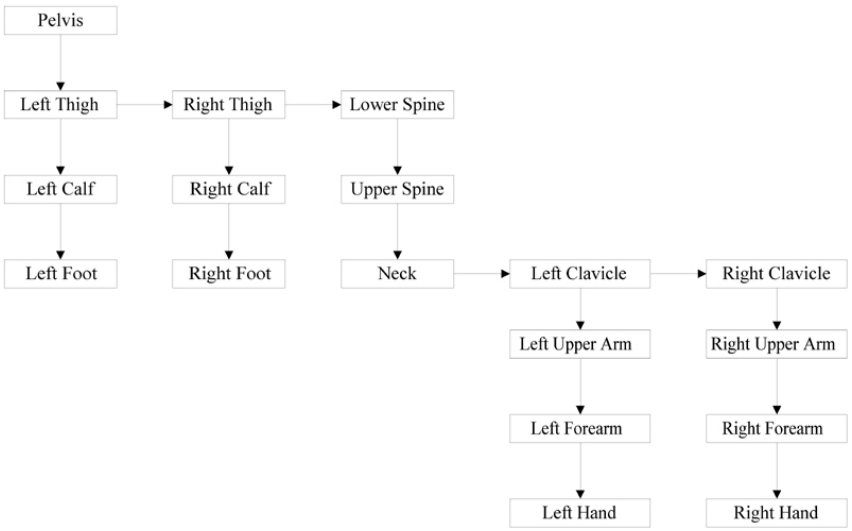

Figure 6: Tree hierarchy diagram of the skeleton of the character depicted in Figure 1. Down vertical arrows represent "first child" relationships, and rightward horizontal arrows represent "sibling" relationships.

Admittedly, in the context of character animation, the name `BONE` is preferred to `D3DXFRAME`. However, we must remember that `D3DXFRAME` is a generic data structure that can describe non-character mesh hierarchies as well. In any case, in the context of character animation we can use "bone" and "frame" interchangeably.

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

Table 1: `D3DXFRAME` data member descriptions.

Data Member	Description
Name	The name of the node.
TransformationMatrix	The local transformation matrix.
pMeshContainer	Pointer to a `D3DXMESHCONTIANER`. This member is used in the case that you want to associate a container of meshes with this frame. If no mesh container is associated with this frame, set this pointer to null. **We will ignore this member for now and come back to `D3DXMESHCONTIANER` in Section 5 of this paper.**
pFrameSibling	Pointer to this frame's sibling frame; one of two pointers used to connect this node to the mesh hierarchy—see Figure 6.
pFrameFirstChild	Pointer to this frame's first child frame; one of two pointers used to connect this node to the mesh hierarchy—see Figure 6.

The immediate problem with `D3DXFRAME` is that it does not have a combined transform member. To remedy this we extend `D3DXFRAME` as follows:

```
struct FrameEx : public D3DXFRAME
{
    D3DXMATRIX combinedTransform;
};
```

1.3 Generating the Combined Transforms in C++

We can compute the combined transform **C** for each node in the hierarchy by recursively traversing the tree top-down. The following C++ code implements this process:

```
void CombineTransforms(FrameEx* frame,
                       D3DXMATRIX& P) // parent's combined transform
{
    // Save some references to economize line space.
    D3DXMATRIX& L = frame->TransformationMatrix;
    D3DXMATRIX& C = frame->combinedTransform;

    C = L * P;

    FrameEx* sibling    = (FrameEx*)frame->pFrameSibling;
    FrameEx* firstChild = (FrameEx*)frame->pFrameFirstChild;

     // Recurse down siblings.
     if( sibling )
          combineTransforms(sibling, P);

     // Recurse to first child.
     if( firstChild )
          combineTransforms(firstChild , C);
}
```

And to start off the recursion we would write:

```
D3DXMATRIX identity;
D3DXMatrixIdentity(&identity);
CombineTransforms( rootBone, identity );
```

Because the root does not have a parent, we pass in an identity matrix for its parent's combined transform.

2 Keyframes and Animation

For this paper we will consider prerecorded animation data; that is, animations that have been predefined in a 3D modeler, or from a motion capture system. Note however, it is indeed possible to dynamically animate meshes at runtime using physics models, for example. Moreover, Section 7 describes a technique that enables us to create new animations by blending together existing animations.

The preceding section stated that as we animate a skeleton, the attached skin is animated accordingly, via the bone transform matrices, to reflect the current pose of the skeleton. The question then is: How do we animate a skeleton?

To keep things concrete we work with a specific example. Suppose that a 3D artist is assigned the job of creating a robot arm **animation sequence** that lasts for five seconds. The robots upper-arm should rotate on its shoulder joint 60° and the forearm should not move locally, during the time interval [0.0s, 2.5s]. Then, during the time interval (2.5s, 5.0s], the upper-arm should rotate on its shoulder joint –30° and the forearm should not move locally. To create this sequence, the artist roughly approximates this animation with three[1] key frames for the upper-arm bone[2], taken at the times the skeleton reaches critical poses in the animation; namely at times $t_0 = 0\,\text{s}$, $t_1 = 2.5\,\text{s}$, and $t_2 = 5\,\text{s}$, respectively—see Figure 7.

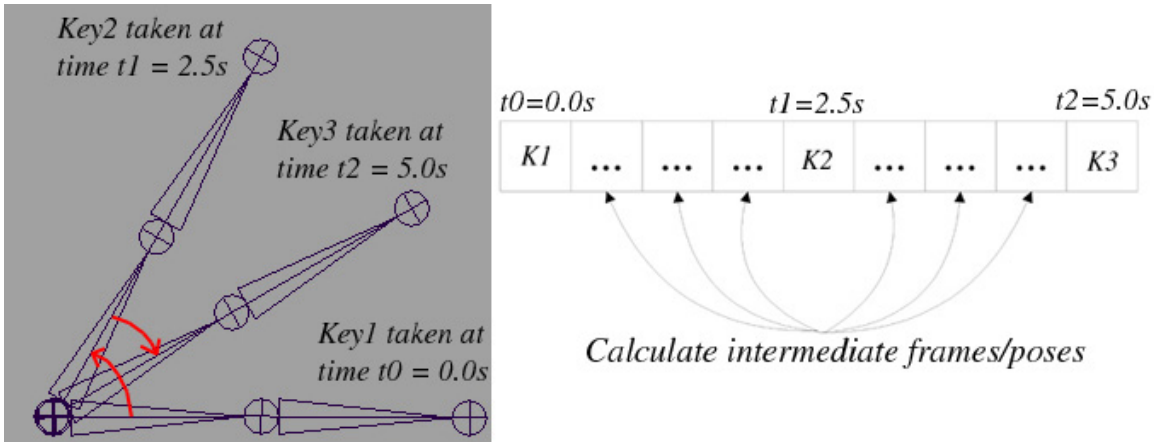

Figure 7: During [0.0s, 2.5s] the arm rotates 60° about the shoulder joint. During (2.5s, 5.0s] the arm rotates -30° about the shoulder joint.

A **key frame** is a *significant* pose of a *bone* in the skeleton at some instance in time. Each bone in the skeleton will typically have several key frames in an animation sequence. Usually, key frames are represented with a rotation quaternion, scaling vector, and translation vector.

Observe that the key frames define the *extreme* poses of the animation; that is to say, all the other poses in the animation lie in-between some pair of key frames. Now obviously three key frames per bone is not enough to smoothly represent a five second

[1] This is a trivial example, in practice many keyframes are required to approximate complex animations such as a human character running or swinging a sword.

[2] Since the forearm does not rotate about its local pivot joint, it does not need its own set of key frames. But, in the case that it did move about its local pivot joint, then the artist would have to define key frames for the forearm as well. In general, key frames will be defined for every bone that is animated.

animation sequence; that is, three frames per five seconds will result in an extremely choppy animation. However, the key idea is this: Given the key frames, the computer can calculate the correct intermediate bone poses between key frames at any time in the five-second sequence. By calculating enough of these intermediate poses (say sixty poses per second), we can create a smooth continuous animation. Figure 8 shows some of the intermediate poses the computer generated for our robot arm.

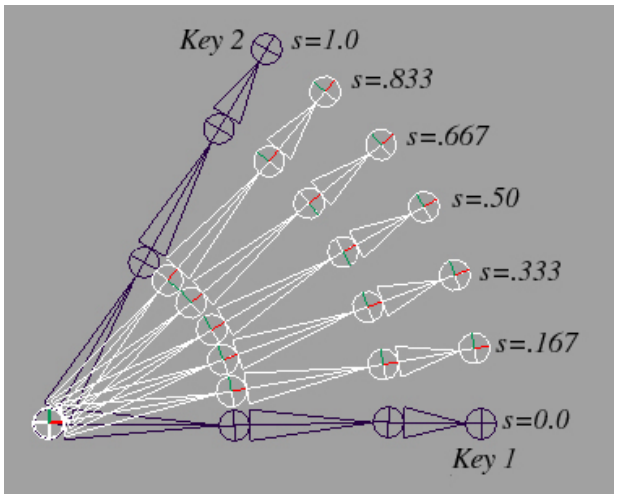

Figure 8: Key frame interpolation.

Returning to our original example shown in Figure 7, during the times [0.0s, 2.5s] the arm will animate from Key 1 to Key 2. Then during the times (2.5s, 5.0s], the arm will animate from Key 2 to Key 3.

2.1 Calculating Intermediate Poses

The intermediate poses are calculated by interpolating between key frames. That is, given key frames K_0 and K_1 we can calculate the intermediate poses by mathematically interpolating from the bone pose described by K_0 to the bone pose described by K_1. Figure 8 shows several intermediate poses calculated via interpolating between key frames K_0 and K_1, for different interpolation parameters taken for $s \in [0, \quad 1]$. We see that as the interpolation parameter *s* moves from zero to one, the intermediate pose moves from K_0 to K_1, respectively. Thus *s* acts like a percentage indicating how far to blend from one key frame to the other.

How do we interpolate between bones? Linear interpolation works for translations and scalings. Rotations in 3-space are a bit more complex; we must use quaternions to represent rotations and spherical interpolation to interpolate quaternion-based rotations correctly. The following D3DX functions perform these interpolation techniques: `D3DXVec3Lerp` and `D3DXQuaternionSlerp`.

Note: We never interpolate matrices because matrix rotations do not interpolate correctly—we must use quaternions to interpolate rotations correctly. Therefore, keyframe transformation data is usually always stored by a rotation quaternion, scaling vector, and translation vector (**RST-values**), and not as a matrix. After interpolating the RST-values (i.e., interpolating rotation quaternions, scaling vectors, and translation vectors) for all bones, we can proceed to build the bone-matrix for each bone out of the interpolated RST-values. Lastly, in the case a file format does present keyframe data as a matrix, we must decompose the matrix into RST-values in order to perform the interpolation.

Now that we know how to calculate intermediate poses, let us review the overall process. Given a time t to calculate the intermediate pose at, the first step is to find two key frames K_i and K_{i+1} taken at times t_0 and t_1, respectively, such that $t_0 \leq t \leq t_1$. These are the two key frames to interpolate between for the given time t. The second step is to transform $t \in [t_0, \quad t_1]$ to the range $t \in [0, \quad 1]$ so that it acts as a percent value indicating how far to interpolate from K_i to K_{i+1}. Next, we iterate over each bone and compute the interpolated RST-values for each bone. Finally, we iterate over the interpolated RST-values and *update the local transformation matrix* of each bone to reflect the current interpolated bone pose for this frame, based on those interpolated RST-values.

Given a time t to calculate the intermediate pose at, the following pseudocode interpolates between two key frames K_0 and K_1, of some bone, taken at times t_0 and t_1, respectively, that satisfy $t_0 \leq t \leq t_1$:

```
struct Keyframe
{
    float time;
    D3DXQUATERNION R;
    D3DXVECTOR3    S;
    D3DXVECTOR3    T;
};

void interpolateBone(Keyframe& K0, Keyframe& K1, D3DXMATRIX& L)
{
    // Transform to [0, 1]
    float t0 = K0.time;
    float t1 = K1.time;
    float lerpTime = (t - t0) / (t1 - t0);

    // Compute interpolated RST-values.
    D3DXVECTOR3 lerpedT;
    D3DXVECTOR3 lerpedS;
    D3DXQUATERNION lerpedR;
    D3DXVec3Lerp(       &lerpedT, &K0.T, &K1.T, lerpTime);
    D3DXVec3Lerp(       &lerpedS, &K0.S, &K1.S, lerpTime);
    D3DXQuaternionSlerp(&lerpedR, &K0.R, &K1.R, lerpTime);
```

```
    // Build and return the interpolated local
    // transformation matrix for this bone.
    D3DXMATRIX T, S, R;
    D3DXMatrixTranslation(&T, lerpedT.x, lerpedT.y, lerpedT.z);

    D3DXMatrixScaling(&S, lerpedS.x, lerpedS.y, lerpedS.z);
    D3DXMatrixRotationQuaternion(&R, &lerpedQ);

    L  = R * S * T;
}
```

Note: It is usually the case that the translation and scaling keys are constant, and therefore do not need to be interpolated.

The above code just interpolates one bone. Of course, in order to animate the entire skeleton, we must perform an interpolation *for each bone* in the skeleton. We call the skeleton of all these interpolated bones an **interpolated-skeleton**.

All that said, we will not have to calculate any intermediate poses ourselves, as that will be handled by the D3DX `ID3DXAnimationController` interface. Nonetheless, it is important to have a basic understanding of what is going on behind the scenes.

3 Rigid Body Animation; A Problem

We now know how to represent an animated character mesh with a hierarchy of bones (i.e., a skeleton), and also how to animate it with key frame interpolation. But we still have not discussed the details of the relationship between a bone and its skin. A straightforward relationship might be to have a separate mesh that corresponds to each bone in the skeleton. Moreover, it is convenient for each of these meshes to be modeled in the bone space of its corresponding bone, because then **the bones' corresponding combined transformations will correctly position the corresponding meshes to form the character.** Therefore, given recomputed combined transforms that reflect the current animation pose, we can render the character mesh by drawing each bone's corresponding mesh with its corresponding combined transform applied to it, as the following code illustrates:

```
void RecursRender(D3DXFRAME* node, D3DXMATRIX& parentTransform)
{
   D3DXMATRIX C = node->TransformationMatrix * parentTransform;

   _device->SetTransform(D3DTS_WORLD, &C);
   for(uint32 i = 0; i < node->pMeshContainer->NumMaterials; ++i)
   {
      D3DXMESHCONTAINER* mc = node->pMeshContainer;
      D3DMATERIAL9& mtrl = mc->pMaterials[i].MatD3D;
      _device->SetMaterial(&mtrl);
```

```
      node->pMeshContainer->MeshData.pMesh->DrawSubset(i);
   }

   if( node->pFrameSibling )
      recursRender(node->pFrameSibling, parentTransform);

   if( node->pFrameFirstChild )
      recursRender(node->pFrameFirstChild , C);
}
```

This technique, called *rigid body animation*, works in that we can indeed animate a character mesh in this way. However, Figure 9 illustrates a flaw it possesses that is considered unacceptable for contemporary games.

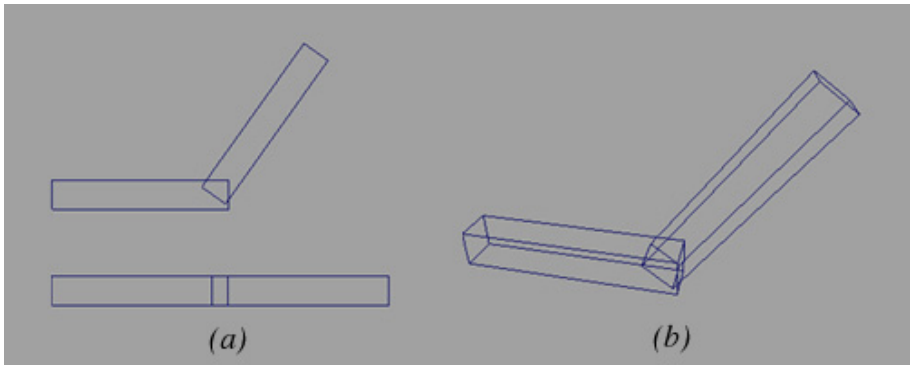

Figure 9: a) Observe the sharp unnatural bending that occurs when we rotate a bone using two separate meshes. b) A perspective view of the unnatural bending.

4 A Solution: Vertex Blending

The rigid body character animation model has a definite flaw by separating a character's skin into unconnected parts. We might do better by treating the character's skin as one continuous mesh. Let us examine a picture that shows ideally what we would like to happen—see Figure 10.

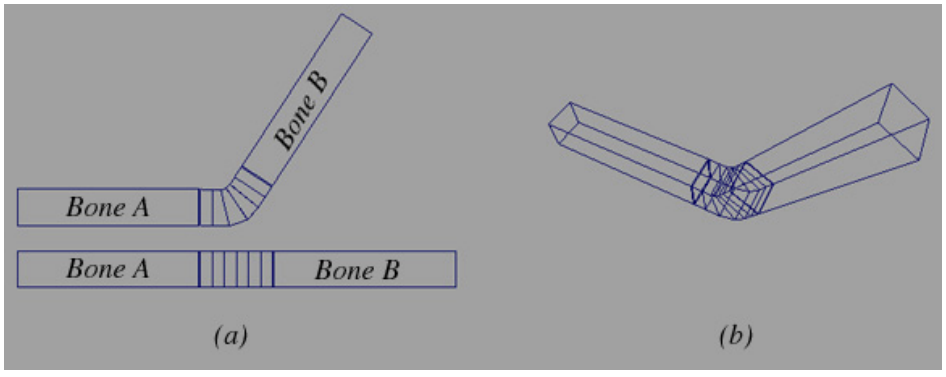

Figure 10: Note the skin is one continuous mesh that spans both bones. a) Observe that the vertices near the joint are influenced by both Bone A and Bone B to create a smooth transitional blend from bone A's position to bone B's position. b) A perspective view of vertex blending.

Observe that the skin is continuous and in some parts of the arm the skin compresses and in other parts it stretches. More specifically, the vertices near the joint seem to be influenced by both the upper-arm and the forearm; in other words, **the position of the vertices near the joint are determined by a weighted average of the two influential bone transformations.** This concept is the key idea of the **vertex blending** algorithm; that is, parts of the skin may be influenced by more than one bone.

4.1 The Offset Transform

Before continuing with the implementation details of vertex blending, let us handle one problem first. Recall that in the articulated character mesh model there was a corresponding mesh for each bone; moreover, that mesh started out in the bone space of its corresponding bone. It then followed that a bone's combined transform would position its corresponding mesh correctly in character space to build the character. However, in vertex blending, we define the character's skin as one continuous mesh in character space. Consequently, because the vertices are not in bone space, we cannot simply use the combined transforms of the bones.

To solve this, we introduce a new transform called the **offset transform**. Each bone in the skeleton has a corresponding offset matrix. An offset matrix transforms vertices, *in the bind pose*[3], from bind space to the space of the respective bone. Figure 11 summarizes our transformations.

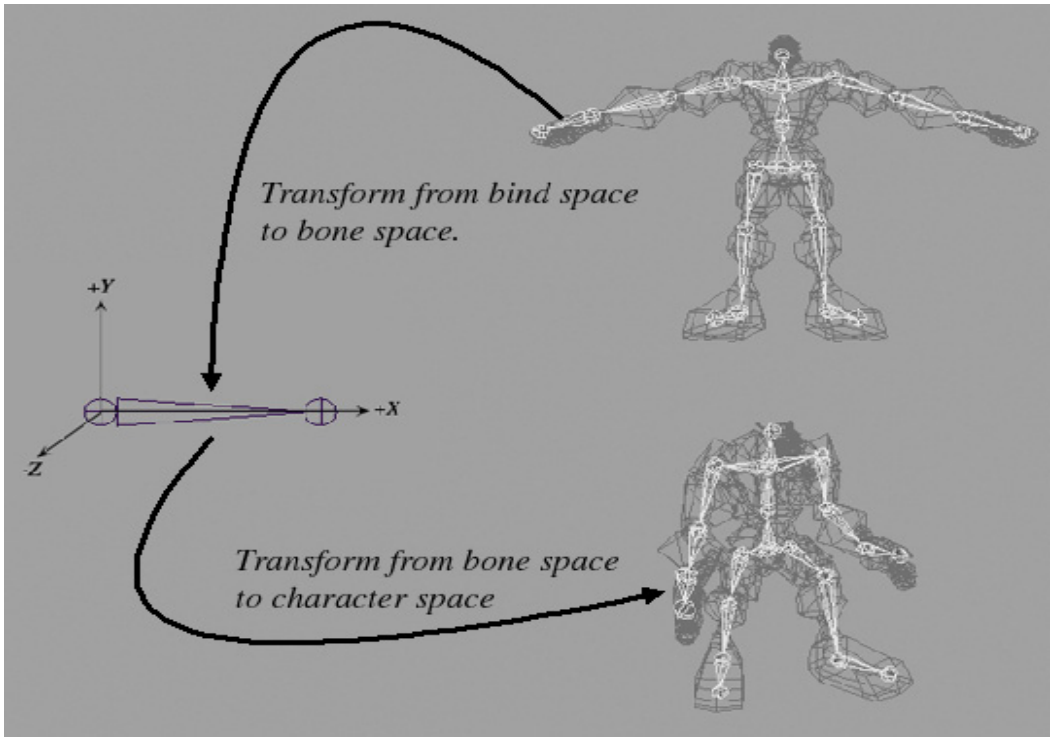

Figure 11: Transformation summary.

[3] The bind pose is the default layout of the character mesh geometry before applying any bone transformations (i.e., the bone transforms are identity matrices).

Thus, by transforming the vertices by the offset matrix of some bone B, we move the vertices to the bone space of B. But once we have the vertices in bone space of B we can use B's combined transform to position it back in character space in its current animated pose! So we now introduce a new transform, call it the **final transform**, which combines a bone's offset transform with its combined transform. Mathematically, the final transformation matrix of the i^{th} bone $\mathbf{F}_i$ is given by:

(2) $$\mathbf{F}_i = \mathbf{M}_i\mathbf{C}_i$$

where $\mathbf{M}_i$ is the offset matrix of the i^{th} bone, and $\mathbf{C}_i$ is the combined matrix of the i^{th} bone.

4.2 Vertex Blending Implementation Details

In practice, *Real Time Rendering* notes that we usually do not need more that four bone influences per vertex. Therefore, in our design we will consider a maximum of four influential bones per vertex.

So to implement vertex blending we now model the character mesh's skin as one continuous mesh. Each vertex contains up to four indices that index into a **matrix palette**; the matrix palette is an array of the final transformation matrices for each bone in the skeleton. Each vertex also has up to four weights that describe the respective amount of influence each of the four influencing bones has on the vertex. Thus we have the following vertex structure for vertex blending:

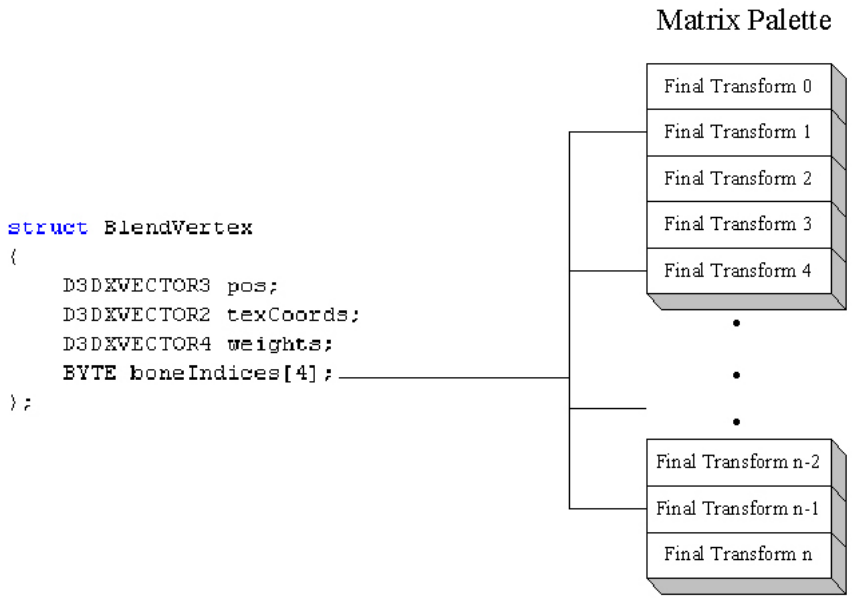

Figure 12: Observe how the four bone indices index into the matrix palette. The bone indices identify the bones that influence the vertex. Note that a vertex is not necessarily influenced by four bones; for instance, only two of the four indices might be used, thereby indicating that only two bones influence the vertex.

A continuous mesh whose vertices have this format is configured for vertex blending and we call it a **skinned mesh.**

Then the final position $\mathbf{v}'$ of any vertex $\mathbf{v}$ can be calculated with:

$$(3) \qquad \mathbf{v}' = \left(\sum_{i=0}^{n-1} w_i \mathbf{v}\mathbf{F}_i \right) = \left(w_0 \mathbf{v}\mathbf{F}_0 + w_1 \mathbf{v}\mathbf{F}_1 + \ldots + w_{n-1} \mathbf{v}\mathbf{F}_{n-1} \right).$$

Observe that in this equation we transform the given vertex $\mathbf{v}$ individually by all of the final bone transforms that influence $\mathbf{v}$. We then take a weighted average of these individually transformed points to compute the final position $\mathbf{v}'$. Hence, the final position is determined by all the influencing bones based on their weight.

Note: The sum of the blend weights should add to 1.0; that is 100%.

Lastly, we present a vertex shader written in the DirectX High Level Shading Language to perform the vertex blending calculations and output the appropriately blended vertex. We use vertex shader version 2.0 so that the number bone influences per vertex can be adjusted dynamically, which allows us to use the same shader for meshes that use two bone influences per vertex, three bone influences per vertex, or four bone influences per vertex. If your graphics card does not support vertex shader 2.0, then the sample programs will run in the REF device. See the following note for modifying the samples to use a lesser vertex shader version.

```
//////////////////////////////////////////////////////////////////////
//
// File: vertblendDynamic.txt
//
// Author: Frank Luna
//
// Desc: Vertex blending vertex shader.  Supports meshes with 2-4
//       bone influences per vertex.  We can dynamically set
//       NumVertInfluences so that the shader knows how many
//       weights it is processing per vertex.  In order to support
//       dynamic loops, we must use at least vertex shader
//       version 2.0.
//
//////////////////////////////////////////////////////////////////////

extern float4x4 WorldViewProj;
extern float4x4 FinalTransforms[35];
extern texture  Tex;
extern int NumVertInfluences = 2;//<--- Normally set dynamically.

sampler S0 = sampler_state
{
    Texture = <Tex>;
```

```
    MinFilter = LINEAR;
    MagFilter = LINEAR;
    MipFilter = LINEAR;
};

struct VS_OUTPUT
{
    float4 pos      : POSITION0;
    float2 texCoord : TEXCOORD;
    float4 diffuse  : COLOR0;
};

VS_OUTPUT VertexBlend(float4 pos         : POSITION0,
                      float2 texCoord    : TEXCOORD0,
                      float4 weights     : BLENDWEIGHT0,
                      int4   boneIndices : BLENDINDICES0)
{

    VS_OUTPUT output = (VS_OUTPUT)0;

    float4 p = float4(0.0f, 0.0f, 0.0f, 1.0f);
    float lastWeight = 0.0f;
    int n = NumVertInfluences-1;

    // This next code segment computes formula (3).
    for(int i = 0; i < n; ++i)
    {
        lastWeight += weights[i];
       p += weights[i]*mul(pos, FinalTransforms[boneIndices[i]]);
    }
    lastWeight = 1.0f - lastWeight;
    p += lastWeight * mul(pos, FinalTransforms[boneIndices[n]]);
    p.w = 1.0f;

    output.pos      = mul(p, WorldViewProj);
    output.texCoord = texCoord;
    output.diffuse  = float4(1.0f, 1.0f, 1.0f, 1.0f);

    return output;
}

technique VertexBlendingTech
{
    pass P0
    {
        vertexShader = compile vs_2_0 VertexBlend();
        Sampler[0] = <S0>;

        Lighting = false;
    }
}
```

Note: There are two problems that can occur for vertex blending when using a lesser vertex shader version than 2.0 (e.g., version 1.1). First, the lesser versions do not guarantee a sufficient number of constant registers to store the matrix palette. You can check the number of shader constants via `D3DCAPS9::MaxVertexShaderConst`. We note that vertex shader version 1.1 only guarantees 96 constant registers. Divide that by four and you have only enough memory for 24 *4x4* matrices. A solution to this problem is to split the skinned mesh into multiple meshes and render it in parts, such that the matrix palette of each part can fit into the constant registers. (`ID3DXSkinInfo::ConvertToIndexedBlendedMesh` can do this split.) Conversely, vertex shader 2.0 guarantees 256 constant registers, which is usually enough in practice, and so we do not need to split the mesh with version 2.0.

A second problem with vertex shader version 1.1 is that some of the older cards (e.g., Geforce 3) do not support the `D3DDECLTYPE_UBYTE4` shader input parameter declaration type, which is used to store the vertex bone indices. The solution here is to convert `D3DDECLTYPE_UBYTE4` to the `D3DDECLTYPE_D3DCOLOR` declaration type, which the Geforce 3 does support. Then we can extract the indices from the `D3DCOLOR` parameter in the shader using the HLSL intrinsic function `D3DCOLORtoUBYTE4`.

Finally, a third inconvenience with vertex shader version 1.1 is that it does not support dynamic branching. That is, we cannot dynamically update shader variables used in loops and conditional statements. Consequently, we would not be able to use one general shader to handle meshes with a different number of bone influences per vertex. And in practice we most definitely will have various character meshes, some requiring two bone influences per vertex, some three, and some four. Clearly it is convenient to have one generalized shader that handles all four cases.

5 Implementation Details in Direct3D 9.0

In this section, we put the theory of the previous sections into practice by animating a skinned mesh, using vertex blending with the D3DX library; the corresponding sample application this discussion refers to is called "d3dx_skinnedMesh." But before examining the implementation details, let us recall what the theory requires from us in order to implement vertex blending animation, so that we have a basic idea of the task at hand.

First and foremost we need a hierarchy of bones and a skinned mesh. Additionally, in order to compute the final transform for each bone (based on the current animated pose), we need the offset transformation matrix and the *updated* combined transformation matrix, for each bone. For animation, we require keyframes for each bone; this keyframe data will be loaded from the .X file; moreover, we need functionality to interpolate bones between keyframes. Given the preceding data and functionality we can animate a skinned mesh using the vertex blending algorithm. The subsequent sections describe the details for obtaining, building, and using this data and functionality.

5.1 D3DXMESHCONTAINER

It was mentioned in Section 1.2 that the D3DXFRAME structure contains a pointer to a D3DXMESHCONTAINER structure, which allowed us to associate a container (linked list) of meshes with a frame. At the time we did not elaborate on D3DXMESHCONTAINER, so we do that now. The D3DXMESHCONTAINER structure is defined as follows:

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

Table 2: D3DXMESHCONTAINER data member descriptions.

Data Member	Description
Name	The name of the mesh.
MeshData	A D3DXMESHCONTAINER is a general structure and can be an ID3DXMesh, ID3DXPMesh, or ID3DXPatchMesh. The D3DXMESHDATA structure specifies what type of mesh this is, and contains a valid pointer to that type of mesh.
pMaterials	Pointer to an array of D3DXMATERIAL structures.
pEffects	Pointer to a D3DXEFFECTINSTANCE structure, which contains effect file information. We will not be loading any effect data from the .X file, and therefore we can ignore this variable.
NumMaterials	The number of elements in the material array pMaterials points to.
pAdjacency	Pointer to the adjacency info of the mesh.
pSkinInfo	Pointer to an ID3DXSkinInfo interface, which contains information needed for performing vertex blending. That is, it contains offset matrices for each bone, vertex weights, and vertex bone indices. The important methods of ID3DXSkinInfo will be discussed later on at the time they are used.
pNextMeshContainer	Pointer to the next mesh in the container. If this value is null, then we are at the end of the mesh container.

5.2 ID3DXAnimationController

The `ID3DXAnimationController` interface is responsible for animation. For each animation sequence, it stores all the key frames for each bone[4]. It also contains the functionality for interpolating between key frames, and some more advance features like animation blending and animation callbacks.

To render a smooth animation we must incrementally update the character from its current pose at time t in the animation sequence to its next pose, some Δt seconds later ($t + \Delta t$). By making Δt small enough, the illusion of a smooth continuous animation is achieved. The `ID3DXAnimationController::AdvanceTime` method does exactly this. Using key frame interpolation, it updates the bones of a character from its current pose at time t in the animation sequence, to the next pose in the animation sequence at time $t + \Delta t$. The method prototype is:

```
HRESULT ID3DXAnimationController::AdvanceTime(
    DOUBLE TimeDelta,
    LPD3DXANIMATIONCALLBACKHANDLER pCallbackHandler);
```

where `TimeDelta` is Δt and we will ignore `pCallbackHandler` for now by passing in null for it. Note that when the animation sequences reaches the end, the current track timer resets and loops back to the beginning of the animation sequence by default.

Once the animation controller has interpolated the bones to reflect the updated pose, we need access to them. Where do we get the interpolated bones? **An important fact about the animation controller is that it contains pointers to the `D3DXFRAME::TransformationMatrix` variables of all the frames in the hierarchy.** This is significant because when the animation controller interpolates the bones, it writes the interpolated bones to their respective `D3DXFRAME::TransformationMatrix` variable. **Thus, we have direct access to the interpolated bones at any time via the `D3DXFRAME::TransformationMatrix` variables, stored in the frame hierarchy.**

5.3 ID3DXAllocateHierarchy

In order to create and destroy a mesh hierarchy using the D3DX functions, we must implement an `ID3DXAllocateHierarchy` interface, which consists of four abstract methods. In doing so, we are able to define how meshes and frames are created and destroyed, thereby giving the application some flexibility in the construction and destruction process. For example, **in our implementation of `CreateMeshContainer`, we elect to ignore meshes that are not skinned meshes.**

The following code shows the child class we define to implement the `ID3DXAllocateHierarchy` interface:

```
class AllocMeshHierarchy : public ID3DXAllocateHierarchy
{
public:
    HRESULT STDMETHODCALLTYPE CreateFrame(
        THIS  PCSTR Name,
```

[4] The key frames for all the bones in a skeleton for a distinct animation sequence are stored in what is called an **Animation Set.**

```
        D3DXFRAME** ppNewFrame);

    HRESULT STDMETHODCALLTYPE CreateMeshContainer(
        PCSTR Name,
        const D3DXMESHDATA* pMeshData,
        const D3DXMATERIAL* pMaterials,
        const D3DXEFFECTINSTANCE* pEffectInstances,
        DWORD NumMaterials,
        const DWORD *pAdjacency,
        ID3DXSkinInfo* pSkinInfo,
        D3DXMESHCONTAINER** ppNewMeshContainer);

    HRESULT STDMETHODCALLTYPE DestroyFrame(
        THIS_ D3DXFRAME* pFrameToFree);

    HRESULT STDMETHODCALLTYPE DestroyMeshContainer(
        THIS_ D3DXMESHCONTAINER* pMeshContainerBase);
};
```

Table 4: `ID3DXAllocateHierarchy` abstract method descriptions.

Function	Description
CreateFrame	Given the frame name as input, create and return a newly allocate `D3DXFRAME` through `ppNewFrame`.
CreateMeshContainer	Given all the parameters, except the last, as valid input values, create and return a newly allocated `D3DXMESHCONTAINER` through `ppNewMeshContainer`.
DestroyFrame	Free any memory or interfaces `pFrameToFree` owns, and delete `pFrameToFree`.
DestroyMeshContainer	Free any memory or interfaces `pMeshContainerBase` owns, and delete `pMeshContainerBase`.

The implementation of these functions is straightforward, so it will not be discussed nor shown here. However, you can refer to this paper's corresponding sample code for the complete implementation of `AllocMeshHierarchy`.

5.4 D3DXLoadMeshHierarchyFromX and D3DXFrameDestroy

After we have implemented an `ID3DXAllocateHierarchy` interface, we can use the following D3DX function to load the mesh hierarchy from an .X file:

```
HRESULT WINAPI D3DXLoadMeshHierarchyFromX(
    LPCSTR Filename,
    DWORD MeshOptions,
    LPDIRECT3DDEVICE9 pDevice,
    LPD3DXALLOCATEHIERARCHY pAlloc,
    LPD3DXLOADUSERDATA pUserDataLoader,
    LPD3DXFRAME* ppFrameHeirarchy,
```

```
    LPD3DXANIMATIONCONTROLLER* ppAnimController);
```

Table 3 summarizes the unobvious parameters of `D3DXLoadMeshHierarchyFromX`.

Table 3: `D3DXLoadMeshHierarchyFromX` abridged parameter descriptions.

Parameter	Description
pAlloc	A pointer to an instance of a class that implements the `ID3DXAllocateHierarchy` interface. By implementing this interface the application can define how the hierarchy and its components are created and destroyed.
pUserDataLoader	A pointer to an instance of a class that implements `ID3DXLoadUserData` interface. By implementing this interface the application can load custom data templates from .X files. Since we are using standard .X file templates, we ignore this parameter.
ppFrameHierarchy	Returns a pointer to the root of the loaded mesh hierarchy.
ppAnimController	Returns a pointer to an allocated `ID3DXAnimationController` interface instance that contains all the animation data from the .X file.

And to destroy the frame hierarchy we can use the `D3DXFrameDestroy` function. The following example function calls show `D3DXLoadMeshHierarchyFromX` and `D3DXFrameDestroy` in use:

```
AllocMeshHierarchy allocMeshHierarchy;
D3DXLoadMeshHierarchyFromX(inputFilename.c_str(),
    D3DXMESH_MANAGED, _device, &allocMeshHierarchy,
    0,/* ignore user data */
    & _root, & _animCtrl);
.
.
.
if( _root )
{
    AllocMeshHierarchy allocMeshHierarchy;
    D3DXFrameDestroy(_root, &allocMeshHierarchy);
    _root = 0;
}
```

5.5 Finding the One and Only Mesh

For simplification purposes, we make the following assumption: **We assume that the input .X file contains exactly one skinned mesh.** This assumption is not particularly limiting; we note, for example, that the DirectX SDK's *tiny.x* file contains one and only one skinned mesh. From this assumption, and from the fact that we ignored non-skinned mesh types in `CreateMeshContainer`, we can infer that there exists

exactly one frame in the hierarchy that has a pointer to a `D3DXMESHCONTAINER` with a valid `pMesh` variable; additionally, because we only read in skinned meshes, that mesh container also contains skinning info (i.e., a non-null `pSkinInfo` pointer).

So let us now find the one and only mesh container. The following method recursively searches the hierarchy for the frame that has the one and only mesh:

```
D3DXFRAME* SkinnedMesh::findNodeWithMesh(D3DXFRAME* frame)
{
	if( frame->pMeshContainer )
		if( frame->pMeshContainer->MeshData.pMesh != 0 )
			return frame;

	D3DXFRAME* f = 0;
	if(frame->pFrameSibling)
		if( f = findNodeWithMesh(frame->pFrameSibling) )
			return f;

	if(frame->pFrameFirstChild)
		if( f = findNodeWithMesh(frame->pFrameFirstChild) )
			return f;

	return 0;
}
```

And the subsequent code that starts off the recursion, saves a local pointer to the mesh container, and saves a member pointer to the skin info:

```
// Find the one and only mesh in the tree hierarchy.
D3DXFRAME* f = findNodeWithMesh(_root);
if( f == 0 ) THROW_DXERR(E_FAIL);
D3DXMESHCONTAINER* meshContainer = f->pMeshContainer;
_skinInfo = meshContainer->pSkinInfo;
_skinInfo->AddRef();
```

Note that we just save a pointer to the mesh container, and moreover we do not take responsibility for freeing it. Because the mesh stays in the hierarchy, it will be freed when we destroy the hierarchy. Conversely, since we `AddRef` the skin info object, we do take responsibility for releasing that interface in the `SkinnedMesh::deleteDeviceObjects` method.

5.6 Converting to a Skinned Mesh

So far we have a pointer to the one and only mesh container (which contains the one and only mesh). However, at this point, the vertex format of the mesh does not include vertex weights or bone index data, both of which are needed for vertex blending. Therefore, we must convert the mesh to an **indexed-blended-mesh** or what is also known as a **skinned mesh**, which does have the necessary vertex format for vertex blending.

The following method, where we pass in a copy of the pointer to the one and only mesh for the parameter, does this conversion:

```
void SkinnedMesh::buildSkinnedMesh(ID3DXMesh* mesh)
{
      DWORD        numBoneComboEntries = 0;
      ID3DXBuffer* boneComboTable      = 0;

      THROW_DXERR( _skinInfo->ConvertToIndexedBlendedMesh(
            mesh,
            D3DXMESH_MANAGED | D3DXMESH_WRITEONLY,
            SkinnedMesh::MAX_NUM_BONES_SUPPORTED,
            0, // ignore adjacency in
            0, // ignore adjacency out
            0, // ignore face remap
            0, // ignore vertex remap
            &_maxVertInfluences,
            &numBoneComboEntries,
            &boneComboTable,
            &_skinnedMesh) )

      // We do not need the bone table, so just release it.
      ReleaseCOM(boneComboTable);
}
```

The key function called in `buildSkinnedMesh` is the `ID3DXSkinInfo::ConvertToIndexedBlendedMesh`. Recall from Section 5.1 that the `ID3DXSkinInfo` interface contains the offset matrices for each bone, vertex weights, and vertex bone indices. Hence, `ID3DXSkinInfo` is the interface capable of converting an input mesh into a skinned mesh.

Most of the parameters of `ID3DXSkinInfo::ConvertToIndexedBlendedMesh` are self explanatory from the example given. However, a few of them deserve some elaboration. The two parameters related to the bone combination table can be ignored since we do not use bone combination table in this paper. The value returned through `_maxVertInfluences` specifies the maximum number of bones that influence a vertex in the skinned mesh. Lastly, observe that we save the resulting skinned mesh into the member variable `_skinnedMesh` of the `SkinnedMesh` class.

5.7 Building the Combined Transform Matrix Array

Because we eventually need to set the matrix palette array (Section 4.2) to the vertex shader, it is convenient to have the combined transforms in an array format. But, we do not need nor want copies of the combined transforms because that would duplicate memory and would mean that we would have to update our array copy whenever the combined transformation matrices in the hierarchy change (and they change every frame). By using an array of pointers to the combined transformation matrices we avoid

duplication and have direct access to the updated combined transformation matrices. The subsequent function saves pointers to the combined transformation matrices matrices:

```
void SkinnedMesh::buildCombinedTransforms()
{
    for(UINT i = 0; i < _numBones; ++i)
    {
        // Find the frame that corresponds with the ith
        // bone offset matrix.
        const char* boneName = _skinInfo->GetBoneName(i);
        D3DXFRAME* frame = D3DXFrameFind(_root, boneName);
        if( frame )
        {
            FrameEx* frameEx = static_cast<FrameEx*>( frame );
            _combinedTransforms[i] = &frameEx->combinedTransform;
        }
    }
}
```

Observe that we store the pointers to the combined transformations such that the i^{th} pointer corresponds with the i^{th} offset matrix. Thus, given the i^{th} bone, we can obtain its i^{th} offset matrix and i^{th} combined transformation matrix. However, there is another reason for configuring the matrix array layout in this way. Recall that `_skinInfo` converted the source mesh to an indexed-blended-mesh. It follows then that the bone indices for the vertices are relative to the `_skinInfo` bone array. Therefore, it is important that we base the combined transformation matrix array relative to the offset matrix array since that is the array the bone indices of the vertices are relative to.

5.8 Initialization Summarized

Let us summarize the initialization steps taken to create and prepare a skinned mesh for rendering. We first implemented an `ID3DXAllocateHierarchy` interface so that we can use the `D3DXLoadMeshHierarchyFromX` and `D3DXFrameDestroy` function to create and destroy the bone hierarchy, respectively. Next, we called `D3DXLoadMeshHierarchyFromX` to actually load the animated character mesh data from the .X file. We then searched the hierarchy for the one frame that contained the character's skin data (i.e., mesh container). Fourth, because the vertex format of the mesh stored in the mesh container was not a skinned mesh (i.e., it did not have vertex weights of bone indices), we had to convert it to a skinned mesh using the `ID3DXSkinInfo::ConvertToIndexedBlendedMesh` method. Lastly, we built an array of pointers to the combined transformation matrices of the bones, so that we have a fast access data structure from which to access the combined transforms. We are now ready for animating and rendering the skinned mesh.

5.9 Runtime Tasks; Animating the Character Mesh

After initialization we have the offset matrices, pointers to the combined transforms of each bone, an animation controller to interpolate the bones to the current pose in the animation sequence, and a skinned mesh configured for vertex blending. That is to say, we have all the data necessary to animate and render the skinned mesh.

We can break the animation and rendering tasks into five steps:

1. Interpolate the bones to the current pose using the `ID3DXAnimationController::AdvanceTime` method. Recall that the animation controller has pointers to the hierarchy frame transformation matrices (`D3DXMATRIX::TransformationMatrix`). The animation controller updates these matrices to reflect the pose, at the current time, of the animation sequence by interpolating between keyframes.

2. Now that the frames are updated to the current pose, recurs down the tree computing the combined transformation matrices for each bone out of the interpolated `D3DXMATRIX::TransformationMatrix`es, like we showed how to do in Section 1.3.

3. For each bone, fetch the i^{th} offset matrix and i^{th} combined transformation matrix, and concatenate them to build the final transformation matrix of the i^{th} bone.

4. Pass the final transformation matrix array, which includes all the transformations to correctly transform the skin to the current pose of the character, to the vertex shader.

5. Finally, render the character mesh in its current pose.

The respective code for the first four steps is given by the `SkinnedMesh::frameMove` method:

```
void SkinnedMesh::frameMove(float deltaTime,
                            D3DXMATRIX& worldViewProj)
{
      _animCtrl->AdvanceTime(deltaTime, 0);

      D3DXMATRIX identity;
      D3DXMatrixIdentity(&identity);
      combineTransforms(static_cast<FrameEx*>(_root), identity);

      D3DXMATRIX offsetTemp, combinedTemp;
      for(UINT i = 0; i < _numBones; ++i)
      {
            offsetTemp   = *_skinInfo->GetBoneOffsetMatrix(i);
            combinedTemp = *_combinedTransforms[i];
            _finalTransforms[i] = offsetTemp * combinedTemp;
      }

       effect->SetMatrix( hWorldViewProj, &worldViewProj);
```

```
        _effect->SetMatrixArray(_hFinalTransforms,
              &_finalTransforms[0], _finalTransforms.size());
}
```

And the respective code for the fifth step is given by the `SkinnedMesh::render` function:

```
void SkinnedMesh::render()
{
      _effect->SetTechnique(_hTech);
      UINT numPasses = 0;
      _effect->Begin(&numPasses, 0);

      for(UINT i = 0; i < numPasses; ++i)
      {
            _effect->BeginPass(i);

            // Draw the one and only subset.
            _skinnedMesh->DrawSubset(0);

            _effect->EndPass();
      }
      _effect->End();
}
```

For our sample application we used the DirectX SDK's tiny.x character mesh model; Figure 13 shows a screenshot. And this concludes our discussion of the "d3dx_skinnedMesh" sample application. The subsequent sections examine some additional features of the `ID3DXAnimationController` interface.

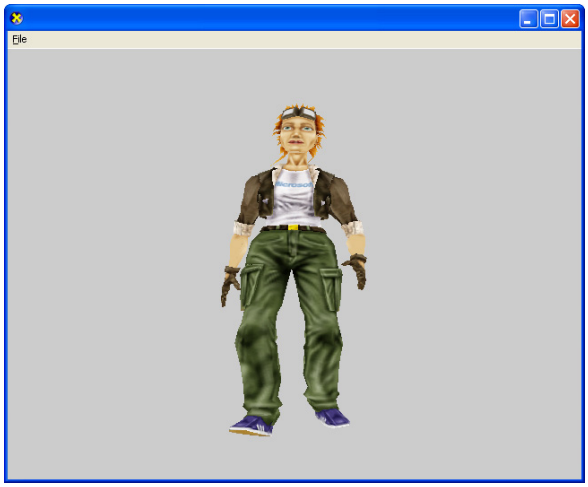

Figure 13: Screenshot from "d3dx_skinnedMesh."

Note: It is probably a good idea to familiarize yourself with the "d3dx_skinnedMesh" sample code before reading the rest of this paper.

6 Multiple Animations

An animated character can contain the data for several animation sequences, which we recall the D3DX Animation API calls *animation sets*. For example, a character may have a walking sequence, a running sequence, a gun firing sequence, a jumping sequence, and a death sequence. In this section, we develop an application called "d3dx_multiAnimation," which demonstrates how to switch between the multiple animation sequences a character contains. We note that the sample developed here is much more simplified than the DirectX SDK 9.0 Summer Update's version, called "MultiAnimation."

Before we begin, we first need an .X file that contains multiple animation sequences (*tiny.x* only has one animation set). In addition to *tiny.x* the DirectX SDK 9.0 Summer Update ships with *tiny_4anim.x*, which includes the same mesh as *tiny.x* file, but contains four animation sets. The four animation sets contain the data for a waving sequence, a jogging sequence, a walking sequence, and a loitering sequence—see Figure 14. The file *tiny_4anim.x* can be found in the Direct3D SDK sample "MultiAnimation."

Figure 14: Quadrant I shows the jogging animation; quadrant II shows the walking animation; quadrant III shows the loitering animation; quadrant IV shows the wave animation.

Note: In the sample program "d3dx_multiAnimation," you can switch to the next animation sequence using the 'n' key. Also, the arrow keys allow you to orbit around the

scene.

Now that we have a multiple animation set .X file, we can load *tiny_4anim.x* the same exact way we loaded *tiny.x* in the last section, and all of the animation sets will be loaded into the animation controller. Once the animation set data is loaded, we can use the animation controller's interface to switch between the animation sequences.

To switch from one animation sequence to another we simply must overwrite the current sequence with a new sequence, and we can do that with the following method:

```
HRESULT ID3DXAnimationController::SetTrackAnimationSet(
    UINT Track,
    LPD3DXANIMATIONSET pAnimSet);
```

`Track` is an index that identifies the track we want to set the new animation set `pAnimSet` to. **Animation tracks** will be explained more in the next section, for now just understand that we are only playing one animation at a time, and therefore will only be using the first track—track zero. So by setting a new animation set (which describes an animation sequence) to track zero, we are overwriting the old animation set with the new one, and thus changing the current animation sequence (i.e., `ID3DXAnimationController::AdvanceTime` will animate the bones using the currently set animation sequence).

But where do we obtain pointers to the animation sets of the animation controller? We do that with this next method:

```
HRESULT ID3DXAnimationController::GetAnimationSet(
    UINT Index,
    LPD3DXANIMATIONSET *ppAnimSet);
```

The method returns the animation set specified by `Index` through the output parameter `ppAnimSet`. And the number of animation sets the animation controller has can be obtained through the `ID3DXAnimationController::GetNumAnimationSets` method.

Finally, when switching between animation sequences we might want to reset the **global animation time**, so that the animation starts at the beginning. We can do that with the `ID3DXAnimationController::ResetTime` method.

Let us summarize this section by examining the implementation details of the "d3dx_multiAnimation" sample program. For modularity purposes, we derive a class specific to *tiny_4anim.x* from `SkinnedMesh`. This class adds functionality to switch between the four animation sequences *tiny_4anim.x* supports:

```
// Note that we hardcode these index values and they are
// specific to tiny_4anim.x.  They were found by actually
// examining the .x file.
enum ANIMATIONSET_INDEX
{
      WAVE_INDEX    = 0,
```

```
      JOG_INDEX    = 1,
      WALK_INDEX   = 2,
      LOITER_INDEX = 3
};

class Tiny_X : public SkinnedMesh
{
public:
      void playWave();
      void playJog();
      void playWalk();
      void playLoiter();
};
```

For brevity we only show the implementation to one of these functions here:

```
void Tiny_X::playLoiter()
{
      ID3DXAnimationSet* loiter = 0;
      _animCtrl->GetAnimationSet(LOITER_INDEX, &loiter);
      _animCtrl->SetTrackAnimationSet(0, loiter);
      _animCtrl->ResetTime();
}
```

In the actual application driver file for "d3dx_multiAnimation" we check a counter every frame that is updated via user input. This counter controls which animation sequence to play. The significant code follows:

```
HRESULT MultiAnimDemoApp::FrameMove()
{
      ...

      switch( _animationIndex )
      {
      case LOITER_INDEX:
            _tinyxMesh.playLoiter();
            break;
      case WALK_INDEX:
            _tinyxMesh.playWalk();
            break;
      case JOG_INDEX:
            _tinyxMesh.playJog();
            break;
      case WAVE_INDEX:
            _tinyxMesh.playWave();
            break;
      }

      _tinyxMesh.frameMove(m_fElapsedTime, _worldViewProj);
```

```
        ...
}
```

So for example, if `animationIndex=LOITER_INDEX`, then `_tinyxMesh.playLoiter()` is called, which overwrites the current animation sequence with the loiter animation sequence. Then when `ID3DXAnimationController::AdvanceTime` is called inside `_tinyxMesh.frameMove`, the bones will be animated as the loiter animation specifies, thereby playing the loiter animation.

7 Animation Blending

Suppose your game character has a running sequence and a gun firing sequence. It is probably the case that you would like your character to be able to fire his gun as he is running. Now obviously you could have your 3D artist create a separate running-firing sequence, but why not save some content production time and memory by letting the computer create the mixed sequence by mathematically blending between the two animations? This is, in fact, what **animation blending** allows you to do: It allows you to take two existing animation sequences and mix them together to create a new sequence. Figure 15 shows the four blended animation sequences we create in this sections corresponding application program "d3dx_blendAnim."

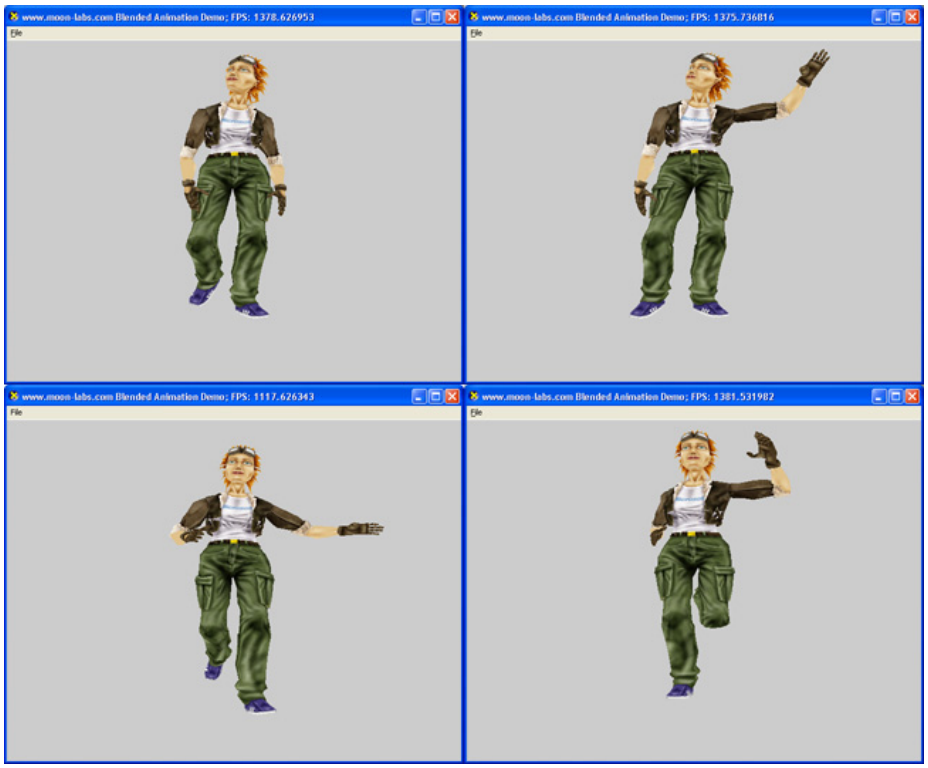

Figure 15: Quadrant I shows a blend of the loitering and wave animation; quadrant II shows a blend of the loitering and walking animation; quadrant III shows a blend of the walking and waving animation; quadrant IV shows a blend of the running and waving animation.

Note: In the sample program "d3dx_blendAnim," you can switch to the next blended animation using the 'n' key. Also, the arrow keys allow you to orbit around the scene.

Conveniently, the D3DX Animation API readily supports animation blending. Recall from the previous section that the animation controller has several different tracks that you can attach an animation set to, but thus far we only have been using the first track—track zero. The key idea to using animation blended, with D3DX, is that **the animation controller *automatically* blends all the *enabled* track animations together.** Therefore, to perform animation blending, all we need to do is attach several animation sets to several different tracks, and enable the tracks. Then when `ID3DXAnimationController::AdvanceTime` animates the bones, it will do so using a blend of all the animation tracks.

Of particular importance is that just adding animation sets to different animation tracks is not enough. The tracks must be enabled, too. A track can be enabled or disabled using the following method:

```
HRESULT ID3DXAnimationController::SetTrackEnable(
    UINT Track,
    BOOL Enable // True to enable and false to disable.
);
```

When enabling a new animation track we should also define its playing speed. We can do that with the following method:

```
HRESULT ID3DXAnimationController::SetTrackSpeed(
    UINT Track,
    FLOAT Speed);
```

Typically `Speed` will be set to equal one, however, you can achieve other effects, such as a "fast forward" or "slow motion" effect by moving `Speed` away from the value one.

Now it would be advantageous if we could specify how much weight each track contributes to the final blended animation. For example, you may want one track to contribute 30% and another track to contribute 70%, to the final blended animation. Unsurprisingly, the D3DX animation API supports this functionality with the following method:

```
HRESULT ID3DXAnimationController::SetTrackWeight(
    UINT Track,
    FLOAT Weight);
```

This method sets the contribution weight track `Track` contributes to the final blended animation. Note that all the track weights should sum to one, usually.

On the topic of weight contributions, D3DX provides an even finer control. We can identify the priority of an animation track as either `D3DXPRIORITY_HIGH` or

D3DXPRIORITY_LOW. During the blending, all high priority tracks will be blended separately and all low priority tracks will be blended separately. Then the result of these two separate blends will be blended together to produce the final blended animation. You can set the priority of a track using this next method:

```
HRESULT ID3DXAnimationController::SetTrackPriority(
    UINT Track,
    D3DXPRIORITY_TYPE Priority
);
```

Let us summarize this section by examining the implementation details of the "d3dx_blendAnim" sample program. Here we modify the derived class Tiny_X from the previous section by adding four new methods that play blended animations:

```
class Tiny_X : public SkinnedMesh
{
public:

      ...

      void playLoiterWaveBlend();
      void playLoiterJogBlend();
      void playJogWaveBlend();
      void playWalkWaveBlend();
};
```

For brevity we only show the implementation to one of these functions here:

```
void Tiny_X::playJogWaveBlend()
{
      ID3DXAnimationSet* jog  = 0;
      ID3DXAnimationSet* wave = 0;

      _animCtrl->GetAnimationSet(JOG_INDEX, &jog);
      _animCtrl->GetAnimationSet(WAVE_INDEX, &wave);

      _animCtrl->SetTrackAnimationSet(0, jog);
      _animCtrl->SetTrackAnimationSet(1, wave);

      _animCtrl->SetTrackWeight(0, 0.4f);
      _animCtrl->SetTrackWeight(1, 0.6f);

      _animCtrl->SetTrackEnable(0, true);
      _animCtrl->SetTrackEnable(1, true);

      _animCtrl->SetTrackSpeed(0, 1.0f);
      _animCtrl->SetTrackSpeed(1, 1.0f);

      _animCtrl->SetTrackPriority(0, D3DXPRIORITY_HIGH);
```

```
        _animCtrl->SetTrackPriority(1, D3DXPRIORITY_HIGH);

        _animCtrl->ResetTime();
}
```

And just like the previous sample "d3dx_multiAnimation," in the actual application driver file for "d3dx_blendAnim" we check a counter every frame that is updated via user input. This counter controls which *blended* animation sequence to play.

Note: The list of `ID3DXAnimationController` methods we have discussed is by no means exhaustive. And although we have discussed the primary ones, there are some others that may be useful. See the DirectX SDK documentation for a complete list.

8 Animation Callbacks

Often it is the case that we would like to execute some code in response to an **animation callback.** An animation callback is triggered at some specified time in an animation sequence where we would like to execute some code that parallels the animation. The SDK sample "MultiAnimation," for example, plays footstep sounds at the particular times *tiny_4anim.x* feet land on the ground. Alternatively, we might play a gunfire sound effect to parallel a gun firing animation sequence. As another example, when a character engages in a particular fighting technique, we may want to zoom in the camera for a close up shot to increase action intensity. The point is this: We will probably want to execute some code (it can be anything) that is tightly coupled to the animation sequence. Moreover, since the code is tightly coupled to the animation, you would rather it not be executed from the main application loop. Instead, it is more organized that the animation itself executes the code at the appropriate time. This task can be accomplished by animation callbacks.

The corresponding sample program for this section is called "d3dx_animCallbacks," and it executes a code block that rotates the camera ninety-degree, in response to the animation. Furthermore, we use *tiny.x* as the animation model for this sample. The following subsections explain how this is done.

8.1 Callback Handlers

A **callback handler** is a function that contains some code that is to be executed at some time during the animation sequence. We define this function by deriving a new class from `ID3DXAnimationCallbackHandler` and implementing the one and only method `HandleCallback`. For example, the derived class in "d3dx_animCallbacks" is defined like so:

```
class TinyXCallbackHandler : public ID3DXAnimationCallbackHandler
{
public:
        HRESULT CALLBACK HandleCallback(
                THIS_ UINT Track, LPVOID pCallbackData);
```

```
};
```

After a callback handler is implemented we must hook it up with the animation controller so that the controller knows which callback handler to call in the case that an animation callback is triggered. To do this, we pass a pointer to an instance of our derived class to the second parameter of `AdvanceTime`. For example:

```
TinyXCallbackHandler callbackHandler;
_animCtrl->AdvanceTime(deltaTime, &callbackHandler);
```

Observe that we can switch between several different callback handlers by deriving several classes from `ID3DXAnimationCallbackHandler` (each with its own unique implementation of `HandleCallback`) and passing pointers of the various instances to `AdvanceTime`.

8.2 Callback Keys

An animation sequence that supports animation callbacks will contain an array of callback keys. A **callback key** defines the time (relative to the animation sequence) an animation callback is to be triggered (i.e., the time a callback handler should be executed), and additionally, the callback key contains the data that is to be passed into the callback handler as an input parameter. The D3DX callback key structure is defined like so:

```
typedef struct _D3DXKEY_CALLBACK {
    FLOAT Time; // Time callback handler should be executed
    LPVOID pCallbackData; // Input data to callback handler
} D3DXKEY_CALLBACK, *LPD3DXKEY_CALLBACK;
```

Notice that the idea of callback keys is similar to key frames. Whereas key frames define the motion of the animation sequence, callback keys define code segments that are to be handled in parallel.

In “d3dx_animCallbacks“ our input data is simply a pointer to the camera’s theta coordinate:

```
struct TinyXCallbackInfo
{
      float* theta;
};
```

You can, of course, pass whatever data you like in the input structure. For instance, you can include conditional flags in your input data structure so that the callback handler can branch to different code blocks based on the various conditions set.

8.3 Handling the Callback

Handling an animation callback is straightforward; given the input data, execute whatever code you desire. In "d3dx_animCallbacks" we are simply going to rotate the camera ninety degrees every time the callback is called:

```
HRESULT TinyXCallbackHandler::HandleCallback(
    UINT Track,
    LPVOID pCallbackData)
{
    // Given your callback data, execute any desired code.
    TinyXCallbackInfo* data = (TinyXCallbackInfo*)pCallbackData;

    // rotate camera 90 degrees
    *(data->theta) += D3DX_PI * 0.5f;

    return D3D_OK;
}
```

8.4 Setting up the Callback Keys

We have discussed callback keys and how to handle them, but so far we have not discussed how to actually add the callback keys to an animation sequence. And if no callback keys are added, then no animation callbacks can be triggered. The following annotated method shows how this is done. Note that this method uses D3DX functions and types that have not been explained in this paper. In most cases the type/method name implies what they represent/do. If it does not, then see the DirectX SDK documentation for details.

```
void Tiny_X::setupCallbackKeyframes(float* theta)
{
      // Remark: 'theta' is a pointer to the camera's theta
      // coordinate.

      // Grab the current animation set for 'tiny.x'
      // (we know there is only one.)
      ID3DXKeyframedAnimationSet* animSetTemp = 0;
      _animCtrl->GetAnimationSet(
            0, (ID3DXAnimationSet**)&animSetTemp);

      // Compress it.
      ID3DXBuffer* compressedInfo = 0;
      animSetTemp->Compress(D3DXCOMPRESS_DEFAULT,
            0.5f, 0, &compressedInfo);

      // Setup two callback keys.
      UINT numCallbacks = 2;
      D3DXKEY_CALLBACK keys[2];

      // Make static so it does not pop off the stack.
      static TinyXCallbackInfo CallbackData;
```

```
        CallbackData.theta = theta;

        // GetSourceTicksPerSecond() returns the number of
        // animation key frame ticks that occur per second.
        // Callback keyframe times are tick based.
        double ticks = animSetTemp->GetSourceTicksPerSecond();

        // Set the first callback key to trigger a callback
        // half way through the animation sequence.
        keys[0].Time = float(animSetTemp->GetPeriod()/2.0f*ticks);
        keys[0].pCallbackData = (void*)&CallbackData;

        // Set the second callback key to trigger a callback
        // at the end of the animation sequence.
        keys[1].Time = animSetTemp->GetPeriod()*ticks;
        keys[1].pCallbackData = (void*)&CallbackData;

        // Create the ID3DXCompressedAnimationSet interface
        // with the callback keys.
        ID3DXCompressedAnimationSet* compressedAnimSet = 0;
        D3DXCreateCompressedAnimationSet(animSetTemp->GetName(),
              animSetTemp->GetSourceTicksPerSecond(),
              animSetTemp->GetPlaybackType(),  compressedInfo,
              numCallbacks, keys,   &compressedAnimSet);

        compressedInfo->Release();

        // Remove the old (non compressed) animation set.
        _animCtrl->UnregisterAnimationSet(animSetTemp);
        animSetTemp->Release();

        // Add the new (compressed) animation set.
        _animCtrl->RegisterAnimationSet(compressedAnimSet);

        // Hook up the animation set to the first track.
        _animCtrl->SetTrackAnimationSet(0, compressedAnimSet);
        compressedAnimSet->Release();
}
```

9 Summary

This concludes the tutorial on skinned mesh character animation. In review, we have learned how to represent the hierarchical skeleton of a character mesh; we have learned how a bone inherits the transforms of its parents; we have learned how an animation sequence can be described by a list of key frames for each bone; we have learned about rigid body animation and its disadvantages; we have learned about skinned mesh animation (vertex blending) and its advantages over rigid body animation; we have learned how to implement vertex blending using the D3DX Animation API; we have learned how to control multiple animation sequences using the D3DX Animation API; we have learned how to blend multiple animation sequences together to create new

sequences using animation blending; and finally, we have learned how to execute code in parallel with an animation using animation callbacks.

10 References

Akenine-Möller, Tomas, and Eric Haines. *Real-Time Rendering*. 2nd ed. Natick, Mass.: A K Peters, Ltd., 2002.

Freidlin, Benjamin. "DirectX 8.0: Enhancing Real-Time Character Animation with Matrix Palette Skinning and Vertex Shaders." *MSDN Magazine*, June 2001. http://msdn.microsoft.com/msdnmag/issues/01/06/Matrix/default.aspx

Lander, Jeff. "Slashing Through Real-Time Character Animation." *Game Developer Magazine*, April 1998. http://www.darwin3d.com/gamedev/articles/col0498.pdf

Lander, Jeff. "Skin Them Bones: Game Programming for the Web Generation." *Game Developer Magazine*, May 1998. http://www.darwin3d.com/gamedev/articles/col0598.pdf

Lander, Jeff. "Over My Dead, Polygonal Body." *Game Developer Magazine*, October 1999. http://www.darwin3d.com/gamedev/articles/col1099.pdf

Microsoft Corporation. Microsoft DirectX 9.0c SDK Documentation. Microsoft Corporation, 2003.