SkinnedMesh Sample

⊟ Collapse All

The SkinnedMesh sample illustrates mesh animation with skinning using D3DX. Skinning is an animation technique that takes data organized in a skeletal-mesh hierarchy and applies geometry blending to transform mesh vertices. The geometry blending generates smooth surfaces with fewer artifacts.

Path

Source	*SDK root* \Samples\C++\Direct3D\SkinnedMesh
Executable	*SDK root* \Samples\C++\Direct3D\Bin\ *x86 or x64* \SkinnedMesh.exe

Sample Overview

Direct3D and D3DX can be used to animate meshes. Applications can load X files containing animation data, then control and render the animated meshes. This sample demonstrates five skinning techniques: fixed-function non-indexed, fixed-function indexed, software, assembly shader, and HLSL shader. Each technique has its advantages and disadvantages, and application developers should weigh these and choose the appropriate techniques for the needs of their application.

Skinning is a popular animation technique that is modeled like a human body. Skinning uses a set of interconnected bones (or frames) that form a hierarchy. Moving or rotating the bones cause the mesh surface to move or rotate. The mesh surface is analogous to the skin of the human body. Each point (or vertex) on the mesh surface is associated with a number of bones. Its position is determined by the position of the associated bones. For instance, consider the hierarchy in figure 1 that describes a human limb:

```
Clavicle
  Upper Arm
    Forearm
      Hand
        Finger0
        Finger1
        Finger2
        Finger3
        Finger4
```

Figure 1. The hierarchy of a human limb.

The skin at the elbow is influenced by the upper arm bone and the forearm bone. If the upper arm remains stationary and the forearm moves, the elbow skin position is affected; the same is true when holding the forearm stationary and moving the upper arm. It can be concluded that the skin at the elbow is associated with both the upper arm and forearm bones. Transforming vertices with multiple matrices are done with geometry blending. Each matrix has a blending weight ranging from 0 to 1, with the sum of all blending weights equal to 1. The blending operation is carried out by first transforming the vertex with each matrix, yielding several transformed vertex coordinates. Then, these coordinates are interpolated based on the corresponding blending weights. Vertices can have different blending weights even if they are influenced by the same bones. This geometry blending allows surfaces to stay smooth when animated.

Furthermore, any movement by an ancestor bone may affect the skin position. For instance, if both the upper arm and forearm bones remain stationary (that is, no rotation), and the clavicle bone moved, the skin between the upper arm bone and the forearm bone should change position. This explains why the child bones are modeled in a hierarchy to generate this dependency.

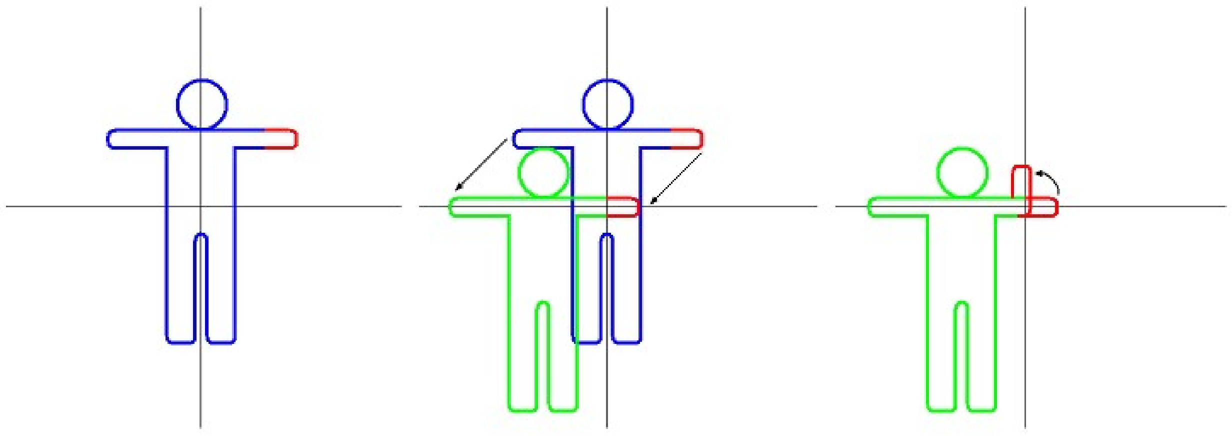

Figure 2. The effect of bone offset transformation on the character's left forearm.

- Left: The character as it is stored on the disk.
- Middle: The bone offset transformation changes the vertex coordinates from the mesh space to bone space.
- Right: Now that vertices are in bone space, animation transformation can be performed. In this case, a rotation is applied to the forearm.

How the Sample Works

The first step in loading a mesh is to call **D3DXLoadMeshHierarchyFromXInMemory**. This call is similar to **D3DXLoadMeshHierarchyFromX**, in that they both load a mesh from an X file into memory. However, D3DXLoadMeshHierarchyFromXInMemory returns the mesh in the form of a hierarchy tree composed of bones (or frames). This form matches how the mesh is structured in the X file. In D3DX, an animatable mesh is composed from a frame hierarchy. At the very top, there is a root frame. The root frame has one or more child frames, each child frame has its own child frames, and so forth.

A frame in the hierarchy is represented by a structure derived from a **D3DXFRAME**. An application may store private information with each frame by deriving the format of each hierarchy node from D3DXFRAME. D3DXFRAME, as shown in figure 3, provides the bare minimum data required by the hierarchy. Each frame has a name, a transformation matrix, and a mesh container for the geometry data. Each frame also contains a link to a sibling and a child frame.

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

Figure 3. D3DXFRAME is the building block of a mesh hierarchy.

The SkinnedMesh sample derives D3DXFRAME and augments the struct with another matrix named a combination matrix which will be used when advancing the animation.

In a hierarchy, some frames will have valid container values, which are the mesh geometry data. When rendering the mesh, the container is drawn regardless of its location in the mesh hierarchy. Similar to a frame, an application should define its own mesh container type by deriving from a **D3DXMESHCONTAINER**, shown in figure 4.

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

Figure 4. A standard mesh container.

All data fields in a D3DXMeshContainer also exist for an ordinary mesh, except for Name and pSkinInfo. Name is a unique string that identifies the mesh container in the hierarchy and pSkinInfo holds bone matrices information. This information consists of the bones by which each vertex is influenced, and is important when setting up the mesh into attribute groups.

In addition to returning a mesh hierarchy, D3DXLoadMeshHierarchyFromX returns an **ID3DXAnimationController** interface associated with the mesh frames. The animation controller controls the flow of the animation available to the mesh; use it specify what animation sequences to play and how fast each sequence plays.

Because the two fundamental structures of a frame hierarchy are often derived and defined by the application itself, the application has to define functions that handle the allocation and deallocation of the frames and mesh containers. **ID3DXAllocateHierarchy** is an interface that describes how the application can define such allocation and deallocation functions. To use it, an application first defines a class derived from ID3DXAllocateHierarchy. Then, implement the four methods declared by the interface: **ID3DXAllocateHierarchy::CreateFrame**, **ID3DXAllocateHierarchy::DestroyFrame**, **ID3DXAllocateHierarchy::CreateMeshContainer**, and **ID3DXAllocateHierarchy::DestroyMeshContainer**. The create methods allocate a new instance of frame or mesh container struct, allocate other resources as necessary, initialize the struct fields, and return the object instance back to the caller.

The destroy methods should free all resources allocated during the creation of the frame or mesh container, including the frame or mesh container instance itself. In the SkinnedMesh sample, CreateMeshContainer also initializes the application-defined fields in the mesh container structure. The structure is shown in figure 5.

```
struct D3DXMESHCONTAINER_DERIVED: public D3DXMESHCONTAINER
{
    LPDIRECT3DTEXTURE9*  ppTextures;

    // SkinMesh info
    LPD3DXMESH           pOrigMesh;
    LPD3DXATTRIBUTERANGE pAttributeTable;
    DWORD                NumAttributeGroups;
    DWORD                NumInfl;
    LPD3DXBUFFER         pBoneCombinationBuf;
    D3DXMATRIX**          ppBoneMatrixPtrs;
    D3DXMATRIX*           pBoneOffsetMatrices;
    DWORD                 NumPaletteEntries;
    bool                  UseSoftwareVP;
    DWORD                 iAttributeSW;
};
```

Figure 5. A derived mesh container.

After retrieving the bone offset matrices from pSkinInfo, CreateNeshContainer calls GenerateSkinnedMesh to set up the mesh hierarchy for animating and rendering. GenerateSkinnedMesh divides the mesh vertices into groups based on the skinning method used, which makes rendering the mesh optimal.

After D3DXLoadMeshHierarchyFromX returns, the sample calls SetupBoneMatrixPointers. This function traverses the frame hierarchy to search for mesh containers. When it finds a mesh container, it calls SetupBoneMatrixPointersOnMesh to initialize the ppBoneMatrixPtrs member of the mesh container. This member is an array of matrix pointers, where each element in the array points to a particular frame matrix. This array provides a quick lookup for frame matrices by frame index. Because the hierarchy is structured as a tree, performing an index-to-matrix lookup is not a straightforward process. This matrix serves as a helper lookup table for efficiency.

Rendering the Mesh

Rendering the animated mesh is a two-step process: setting up the matrices and the actual rendering. To set up the matrices, **ID3DXAnimationController::AdvanceTime** is called first. This method takes a TimeDelta parameter that indicates how much to advance since the last call, then it updates the frame hierarchy's transformation matrices with matrices that correspond to the bone positions at that instance of time. These matrices contain transformation with respect to their parent frames, consisting of a scaling plus rotation plus translation transformation. Next, the sample calls UpdateFrameMatrices. This function updates the frame hierarchy's combined transformation matrix. The combined transformation matrix holds the product of all of the ancestor frames' matrices, including the frame's own matrix. Recall that bones are influenced by their parent bones, and the parent bones are influenced by the grandparent bones, and so forth. Combining all influencing matrices makes the combined transformation matrix absolute, or relative to the world, which is more suitable for rendering. UpdateFrameMatrices achieves this by recursively traversing the hierarchy tree. For each frame, it writes the product of the transformation matrix and the parent's matrix to the combined transformation matrix. Then it calls UpdateFrameMatrices on its sibling and first child nodes, passing its parent matrix to the sibling and passing its own combined transformation matrix as the parent matrix for the children.

The actual rendering, like all other samples, starts in OnFrameRender. The rendering of the mesh is done by the DrawFrame. This function takes a pointer to a frame node, draws the frame's mesh if one exists, then recursively calls itself again with the frame's sibling and children. The result is that all mesh containers in the frame hierarchy will be drawn when the top DrawFrame returns. When a frame holds a valid mesh container, DrawFrame calls the DrawMeshContainer, which is where all the mesh rendering takes place.

DrawMeshContainer renders the mesh in different ways:

- Rendering with Fixed-Function Non-Indexed Skinning
- Rendering with Fixed-Function Indexed Skinning
- Rendering with Shader-Based Skinning
- Rendering with Software Skinning

Rendering with Fixed-Function Non-Indexed Skinning

When this skinning technique is active, the device can load up to four world matrices at a time and transform the vertices influenced by those four matrices. After these vertices are rendered, a different set of matrices can be loaded, and vertices influenced by the second set of matrices can be rendered. This rendering process continues until all faces of the mesh are rendered. GenerateSkinnedMesh calls **ID3DXSkinInfo::ConvertToBlendedMesh** which divides a mesh's faces into attribute groups. Each attribute group identifies geometry influenced by a particular set of four matrices. At render time, each attribute group can be rendered with a single **IDirect3DDevice9::DrawIndexedPrimitive**.

For non-indexed skinning, DrawMeshContainer loops through each attribute group in the mesh and renders one attribute group at a time. The mesh is set up so that each attribute group is influenced by up to four matrices. Before the vertices can be sent to the device, the influencing matrices first need to be sent to the device. To set the matrices, **IDirect3DDevice9::SetTransform** is called with the transformation type D3DTS_WORLDMATRIX(i), where i ranges from a minimum of 0 to a maximum of 3. The matrices that the function sends to the device for a particular attribute group are obtained from pBoneCombinationBuf of the mesh container. This member points to a buffer containing an array of type **D3DXBONECOMBINATION**. Each **D3DXBONECOMBINATION** identifies the bone matrices influencing this attribute group. With this information, the sample knows which four matrices to load on the device to properly render an attribute group. Before actually setting the matrices, however, the matrices need to have the bone offset matrices applied to them. Recall that bone offset matrices transform the vertices from the mesh's default pose on the disk to the parent bone's frame of reference. Therefore, a matrix that gets sent to the device is the product of the bone matrix and the frame's combined transformation matrix. After loading the matrices, the faces of the attribute group can be rendered by calling **ID3DXBaseMesh::DrawSubset**.

Rendering with Fixed-Function Indexed Skinning

With the fixed-function indexed skinning, the device has a matrix palette. The size of the palette can be obtained from the device capability information. The application can load as many matrices as the palette could hold. Each vertex in the mesh has up to four indices to identify the matrices in the palette that influence the vertex. In the sample, the code uses a maximum 12-matrix palette. Using a larger palette is more efficient, but using a smaller palette makes the code compatible with more devices. To obtain a mesh suitable for indexed skinning, GenerateSkinnedMesh calls **ID3DXSkinInfo::ConvertToIndexedBlendedMesh** (which is similar to **ID3DXSkinInfo::ConvertToBlendedMesh**) but it takes a palette size and divides the mesh vertices into attribute groups to work with the matrix palette. Each attribute group is influenced by one specific set of matrices that can fit in the palette, and can be drawn with a single **IDirect3DDevice9::DrawIndexedPrimitive**.

The render code for indexed skinning looks extremely similar to the non-indexed case. GenerateSkinnedMesh loops through the attribute groups and sets up the matrices to send to the device, just as it would for non-indexed skinning. Here, each attribute group identifies the vertices influenced by a particular set of matrices in the palette. In the **D3DXBONECOMBINATION** array, each element identifies the bone matrices with which to load the palette when rendering this attribute group. After the matrices are set, this attribute group's faces are rendered with a call to **ID3DXBaseMesh::DrawSubset**.

Rendering with Shader-Based Skinning

Shader-based skinning is, in concept, similar to fixed-function indexed skinning. Instead of using the hardware-supported matrix palette, the application uses shader constants to load the matrices. Then, the vertex shader can read in the matrix values and perform the skinning transformation. The sample supports two types of shader-based skinning: assembly and HLSL. Virtually, the only difference between the two is how the shader is written. Similar to fixed-function indexed skinning, GenerateSkinnedMesh computes the number of constants required for the palette, then calls **ID3DXSkinInfo::ConvertToBlendedMesh** to obtain a mesh that is attribute-sorted by palette. It is worth noting that each matrix in the palette only requires three shader constants because the fourth column is always (0, 0, 0, 1) and does not need to be explicitly stored.

Shader-based skinning is a form of indexed skinning. Instead of loading the matrices with **IDirect3DDevice9::SetTransform**, they are loaded as shader constants. The sample computes the matrices to send to the device in exactly the same manner as in the fixed-function indexed case. When sending the matrices, the sample calls **IDirect3DDevice9::SetVertexShaderConstantF** when using an assembly shader, or **ID3DXBaseEffect::SetMatrixArray** when using an HLSL shader. After this is done and the shader is set up, the attribute group can be rendered with DrawSubset.

Rendering with Software Skinning

Software skinning, as the name suggests, performs the skinning work completely in software. The process takes an input mesh and the bone matrices, carries out the transformation and blending, then writes the result into another mesh. This resulting geometry represents the animation pose desired and can be sent to the device for rendering. The biggest drawback of software skinning is performance. Software skinning is inherently slow because it is done by software instead of the efficient hardware. However, software skinning is also the only technique that is guaranteed to work regardless of the device on which the application runs. (Note that other skinning techniques can also work on all devices by using software vertex processing. However, software skinning can be useful in certain situations, such as performing a hit test on the animated mesh.) To set up for software skinning, the sample allocates an array of matrices to function as the software matrix palette. During rendering, this matrix array will be filled with skinning matrices and used to animate the mesh.

Essentially, software skinning can be thought of as the indexed skinning with an infinitely large palette (or as large as the system memory allows) done completely in software. All matrices can be loaded simultaneously for rendering. In the sample, DrawMeshContainer computes all matrices for the mesh and stores the result in a matrix array. **ID3DXSkinInfo::UpdateSkinnedMesh** implements software skinning on a set of source vertices using the passed transformation matrices and fills a set of output vertices with animated coordinates. When it returns, the MeshData member of the mesh container holds a skinned and animated mesh; the entire mesh can now be drawn by calling DrawSubset. The code still has to draw each attribute group at a time if the mesh contains multiple materials.