Skinning10 Sample

Collapse All

This sample demonstrates multiple ways to perform skinning on Direct3D 10 hardware.

Path

Source	*SDK root* \Samples\C++\Direct3D10\Skinning10
Executable	*SDK root* \Samples\C++\Direct3D10\Bin\ *x86 or x64* \Skinning10.exe

How the Sample Works

This sample focuses on different methodologies of indexing transformation matrices through bone indices that are defined within the mesh vertex stream. It does not cover skeletal animation or hierarchal transformations.

The Skinning10 sample demonstrates 4 different methods of indexing bone transformation matrices for GPU skinning. In addition, it demonstrates how stream out, without a geometry shader, can be used to reduce the cost of vertex processing when the skinned model will be used for multiple rendering passes.

Constant Buffer Skinning

Of all 4 methods described here, *constant buffer* skinning is the most like the traditional Direct3D 9-style GPU-based skinning. Transformation matrices are loaded into shader constants, and the vertex shader indexes these based upon the bone index in the vertex stream. Like most Direct3D 9 implementations, the number of bone indices per vertex is limited to 4. Unlike most Direct3D 9 implementations, the skinning method that is used in this sample does not create a palette of matrices, nor does it divide the mesh into sections by vertices that share like indices. This is because Direct3D 10 allows constant buffers to be much larger than Direct3D 9 constant buffers. Only for meshes with several hundred bones would splitting the bone matrices into palettes be a necessity. The code for indexing the bones in the shader is as follows:

```
mret = g_mConstBoneWorld[ iBone ];
```

Texture Buffer Skinning

Texture buffer (or tbuffer) skinning looks basically the same as constant buffer skinning. In fact, the bones are referenced the same way in HLSL.

```
mret = g_mTexBoneWorld[ iBone ];
```

The only difference is that the constant buffer that holds the transformation matrices is declared as a texture buffer instead of a normal constant buffer. Declaring a buffer as a tbuffer allows it to be accessed like a texture, which may increase performance for applications that require a more random access of the texture.

Texture Skinning

A logical extension of tbuffer-based skinning is to store the transformation matrices for the bones in a floating point texture. Depending on the texture-fetch performance of the video hardware, as well as whether the shader is ALU- or texture-fetch bound, this method could yield a surprising increase in skinning speed.

```
iBone *= 4;
float4 row1 = g_txTexBoneWorld.Load( float2(iBone,     0) );
float4 row2 = g_txTexBoneWorld.Load( float2(iBone + 1, 0) );
float4 row3 = g_txTexBoneWorld.Load( float2(iBone + 2, 0) );
float4 row4 = g_txTexBoneWorld.Load( float2(iBone + 3, 0) );

mret = float4x4( row1, row2, row3, row4 );
```

As can be seen here, the act of fetching transformations by way of 4 texture fetches may be faster than fetching the same transformation from a constant buffer, depending on the hardware.

Buffer Skinning

The final method of indexing transformation matrices is through *buffer skinning*. In buffer skinning, the matrices are loaded into an **ID3D10Buffer** object which is then bound as a shader resource. The matrix data is loaded from the buffer using the **Load** command.

```
 iBone *= 4;
float4 row1 = g_BufferBoneWorld.Load( iBone );
float4 row2 = g_BufferBoneWorld.Load( iBone + 1 );
float4 row3 = g_BufferBoneWorld.Load( iBone + 2 );
float4 row4 = g_BufferBoneWorld.Load( iBone + 3 );

mret = float4x4( row1, row2, row3, row4 );
```

Speeding up Multiple Passes with Stream Out

One traditional problem with GPU skinning was that if multiple passes needed to be performed on an object, the object needed to be skinned once for each pass. Take, for example, a scene with 3 shadow-mapped light sources and a skinned character; to generate shadows for each of the lights, the character needs to be drawn 3 times — once from the point of view of each light. (Note that amplifying and streaming geometry to multiple render targets by using the geometry shader could also help alleviate this problem.) This would require that the vertex shader skin the character 3 times. The unfortunate reality is that this is two times too many — we really want to skin the character once and save this transformed character to use for the remainder of the frame. That way, if we had a scene that required 10 passes, the character wouldn't need to be skinned 10 times.

Fortunately, Stream Out allows us to do just that. Direct3D 10 allows the sample to skin the character once and stream the results out to a buffer in video memory. Then, for the remainder of the passes, the application just uses the pre-skinned vertices. For a character that may take 10 passes to render, this is 9-fold savings in vertex processing.

This sample does not render multiple passes, but instead renders the character multiple times. It's the same problem, just stated differently. When Stream Out is disabled, the application skins the character each time that it renders it. However, when Stream Out is enabled, it skins the character once, saves the results via Stream Out, and renders the multiple characters by using the pre-skinned geometry.