HLSLWithoutFX10 Sample

Collapse All

HLSLwithoutEffects demonstrates using HLSL to write vertex and pixel shaders without using the effect interfaces. HLSL is a language that closely resembles C syntax and constructs.

Path

Source	*SDK root*\Samples\C++\Direct3D10\HLSLWithoutFX10
Executable	*SDK root*\Samples\C++\Direct3D10\Bin\x86 or x64\HLSLWithoutFX10.exe

How the Sample Works

The scene that this sample renders consists of a square grid of triangles lying on the XZ plane. In each frame, the vertices of this grid will move up or down along the Y direction based on a function of their distance to the origin and time. The vertices are also lit using another function of the distance and time. The time is incremented for each frame. Because the Y coordinate and color of the vertices are generated in each frame, they do not need to be stored. Therefore, the vertex declaration only contains a D3DXVECTOR2 for the X and Z coordinates.

During initialization, the sample calls D3DX10CompileShaderFromFile to read the shader function from a file and compile it into binary shader. After this process, an ID3D10Blob contains the shader binary. Then, the sample calls CreateVertexShader or CreatePixelShader with the shader code to obtain a ID3D10Device object that can be passed to the device.

In FrameMove, the sample uses the ID3D10Buffer interface to set the shader's global variables mViewProj and fTime in order to update the view + projection transformation and time parameter for the shader. At render time, the sample calls VSSetShader and PSSetShader to set the vertex and pixel shaders to be used for any subsequent rendering calls. GSSetShader is used to set the geometry shader to NULL, or unused. When DrawIndexed is called after that, the vertex shader will be invoked once for every vertex processed. The pixel shader will be invoked for every pixel rasterized.

To ensure that the structure defining the shader constants in the application code (VS_CONSTANT_BUFFER) lines up with the constant buffer in the shader (cbuffer cb0), manual packing of constants is used. By default Direct3D 10 attempts to pack as many variables into float4s as possible. This is similar to using #pragma pack(16) in Visual Studio, except that the HLSL compiler will actually pack multiple variables into a single float4. For example, two float2 variables would, by default, take up 4 float positions in HLSL, but 8 positions in Visual Studio. To ensure that variables are packed a certain way, the packoffset keyword can be used to manually discern where each variables is placed in the constant buffer. In the shader, the usage of the packoffset keyword is as follows.

```
//-----------------------------------------------------------------------------
// Global variables
//-----------------------------------------------------------------------------
cbuffer cb0
{
    row_major float4x4 mWorldViewProj : packoffset(c0);     // World * View * Projection transformation
    float fTime :                       packoffset(c5.y);   // Time parameter. This keeps increasing
                                                            // Notice, that this parameter is placed in c5.y.
                                                            // If it were packed by the default packing rules,
                                                            // it would be placed in c4.x
};
```

packoffset(c0) places the first entry of the mWorldViewProj matrix at c0. This ensures that the elements of the matrix cover constants c0, c1, c2, and c3. fTime is placed in c5.y. The default packing rules would place the fTime variable in c4.x. This methodology can be used to arrange variables in the constant buffer into very specific patterns. For example, putting the fTime variable at c4.z, would leave a space of 2 floats between the mWorldViewProj and fTime. Likewise, the application would have to change where it puts fTime in the constant buffer to ensure that the fTime variable is in the place that the shader expects.

In the sample, the vertex shader is written in a text file called HLSLwithoutEffects10.vsh. In this file, there are two global variables and a shader function called Ripple. Ripple takes a float2 as input (for X and Z of the vertex) and outputs two float4 representing the screen-space position and vertex color. The Y coordinate is generated using this formula:

```
Y = 0.1 * sin( 15 * L * sin(T) );
```

L is the distance between the vertex and the origin before the Y adjustment, so it has the value of sqrt(X2 + Z2), because the vertices are lying on the XZ plane. T is the fTime global variable.

The color of the vertex will be some shade of gray based on this formula:

```
C = 0.5 - 0.5 * cos( 15 * L * sin(T) );
```

C will be the value used for all red, green, blue, and alpha and will range from 0 to 1, giving the vertex a color of black to white, respectively. The result looks like ripples with width changing back and forth between narrow and wide and with proper shading based on the slope.