

Tutorial 18: Simple Vertex and Pixel Shaders

by Rick Wong

Content

This tutorial will explain the basic implementation of Vertex and Pixel Shaders without using the `ID3DXEffect` interface. In this example, we will draw a quad with a texture applied to it and we will add a simple sepia effect.

main.cpp - Initializing

Below you see the variables that we will use. If you read this tutorial, you should already be familiar with the Direct3D vertex buffer and texture objects. If you are not, please refer to previous tutorials. There are two variables required for the use of vertex shaders: a vertex declaration and a vertex shader object. The vertex declaration is simply a more detailed version of the Flexible Vertex Format. The `IDirect3DVertexShader9` encapsulates a vertex shader in Direct3D. If you look closely, you might notice that I declared a third variable, namely the `ID3DXConstantTable` object. This little helper is provided by the D3DX Library and aids us in communicating with the shaders. An example would be setting a time variable or even a whole matrix! The `IDirect3DPixelShader9` encapsulates a pixel shader in D3D and the `ID3DXBuffer` is used to load the shaders from a file.

```
int WINAPI WinMain(HINSTANCE instance,
                  HINSTANCE previousInstance,
                  LPSTR cmdline,
                  int cmdShow)
{
    MSG message;
    HRESULT result;

    //variables for this tutorial
    LPDIRECT3DVERTEXBUFFER9      quadVB = NULL; //VertexBuffer
    LPDIRECT3DTEXTURE9          quadTexture = NULL; //Texture

    LPDIRECT3DVERTEXDECLARATION9 vertexDecl = NULL; //VertexDeclaration (NEW)
    LPDIRECT3DVERTEXSHADER9      vertexShader = NULL; //VS (NEW)
    LPD3DXCONSTANTTABLE          constantTable = NULL; //ConstantTable (NEW)

    LPDIRECT3DPIXELSHADER9      pixelShader = NULL; //PS (NEW)

    LPD3DXBUFFER                 code = NULL; //Temporary buffer (NEW)
```

The following code will set up the data for a textured quad. You can find the `D3DVERTEX` struct in *d3ddefs.h*. (If you don't understand this code, refer to the Draw Polygons tutorial.)

```
//set up vertex buffer
D3DVERTEX quad[4] = {{D3DXVECTOR3(-3.0f, -3.0f, 10.0f),
                    D3DXVECTOR2(0.0f, 1.0f)},
                    {D3DXVECTOR3(-3.0f, 3.0f, 10.0f),
                    D3DXVECTOR2(0.0f, 0.0f)},
                    {D3DXVECTOR3(3.0f, -3.0f, 10.0f),
                    D3DXVECTOR2(1.0f, 1.0f)},
                    {D3DXVECTOR3(3.0f, 3.0f, 10.0f),
                    D3DXVECTOR2(1.0f, 0.0f)}};

D3DVERTEX *ptr = NULL;
app.getDevice()->CreateVertexBuffer(sizeof(quad),
                                   D3DUSAGE_WRITEONLY,
                                   0,
                                   D3DPOOL_MANAGED,
```

```

        &quadVB,
        NULL);
quadVB->Lock(0, 0, (void**)&ptr, 0);
memcpy((void*)ptr, (void*)quad, sizeof(quad));
quadVB->Unlock();

```

The following code loads a texture from a file. You should be familiar with this part as well, otherwise refer to the Texture Mapping tutorial.

```

//set up texture
D3DXCreateTextureFromFile(app.getDevice(), "texture.png", &quadTexture);

```

Finally, we're there. We are about to load a vertex shader. But first we have to create a vertex declaration. The following array is composed of several `D3DVERTEXELEMENT9` structures, one for each attribute in the vertex buffer (position and one texture coordinate in our case). The first entry defines the vertex stream for the particular attribute. Since we only use the first vertex stream this sticks to zero. The second entry is the offset of an attribute in a vertex and is zero for the first. A position consists of three floating point values ($3 * 4 \text{ Bytes} = 12 \text{ Bytes}$) and therefore the texture coordinates get that offset. Third we have the data type of the attributes followed by `D3DDECLMETHOD_DEFAULT`, which is rarely changed. The next entry specifies the usage of the attribute and is followed by an usage index making it possible to have multiple occurrences for one usage (e.g.: multiple texture coordinate sets).

This data is then passed to Direct3D to make it a valid `IDirect3DVertexDeclaration9`. A complete list of valid entries for the vertex declaration can be found in the documentation.

```

//set up Vertex Shader (NEW)
D3DVERTEXELEMENT9 decl[] = {{0,
    0,
    D3DDECLTYPE_FLOAT3,
    D3DDECLMETHOD_DEFAULT,
    D3DDECLUSAGE_POSITION,
    0},
    {0,
    12,
    D3DDECLTYPE_FLOAT2,
    D3DDECLMETHOD_DEFAULT,
    D3DDECLUSAGE_TEXCOORD,
    0},
    D3DDECL_END()};
app.getDevice()->CreateVertexDeclaration(decl, &vertexDecl);

```

Now we have got the vertex declaration that is needed for the GPU to pass correct data to a vertex shader. It is time to actually load up a vertex shader! We will ask D3DX to compile the HLSL code into valid operations and then we pass it to Direct3D. We also get a constant table from D3DX, this is used later to communicate with the vertex shader.

```

result = D3DXCompileShaderFromFile("vertex.vsh", //filepath
    NULL, //macro's
    NULL, //includes
    "vs_main", //main function
    "vs_1_1", //shader profile
    0, //flags
    &code, //compiled operations
    NULL, //errors
    &constantTable); //constants

if(FAILED(result))
    MessageBox(app.getHandle(), "Invalid vs code", "Error", MB_OK);

```

```
app.getDevice()->CreateVertexShader((DWORD*)code->GetBufferPointer(),
                                   &vertexShader);
code->Release();
```

Okay, we are done initializing the vertex shader part. Now we move on to the pixel shader, which is initialized analogous to the vertex shader. Even easier, since there is no such thing like a pixel declaration. So we only have to create the pixel shader object.

```
//set up Pixel Shader (NEW)
result = D3DXCompileShaderFromFile("pixel.psh", //filepath
                                   NULL,       //macro's
                                   NULL,       //includes
                                   "ps_main",  //main function
                                   "ps_1_1",   //shader profile
                                   0,         //flags
                                   &code,     //compiled operations
                                   NULL,     //errors
                                   NULL);     //constants

if(FAILED(result))
    MessageBox(app.getHandle(), "Invalid ps code", "Error", MB_OK);

app.getDevice()->CreatePixelShader((DWORD*)code->GetBufferPointer(),
                                   &pixelShader);
code->Release();
```

main.cpp - Rendering

Now it is time for rendering, but we need to update the constants that are used in the shaders before. Constants are variables, declared in the shader code itself. As I said before, the `ID3DXConstantTable` object will be used for writing to those variables.

```
while(app.getRunning())
{
    if(PeekMessage(&message, 0, 0, 0, PM_REMOVE))
    {
        TranslateMessage(&message);
        DispatchMessage(&message);
    }

    if(app.checkDevice())
    {
        app.getDevice()->Clear(0,
                              NULL,
                              D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                              D3DCOLOR_XRGB(100,100,100),
                              1.0f,
                              0);
        app.getDevice()->BeginScene();

        //communicate with shader (NEW)
        D3DXMATRIXA16 matWorld, matView, matProj;
        app.getDevice()->GetTransform(D3DTS_WORLD, &matWorld);
        app.getDevice()->GetTransform(D3DTS_VIEW, &matView);
        app.getDevice()->GetTransform(D3DTS_PROJECTION, &matProj);

        D3DXMATRIXA16 matWorldViewProj = matWorld * matView * matProj;
        constantTable->SetMatrix(app.getDevice(),
                                 "WorldViewProj",
                                 &matWorldViewProj);
```

Three new lines here. Instead of `SetFVF()` we call `SetVertexDeclaration()` and pass the vertex

declaration we created earlier. `SetVertexShader()` and `SetPixelShader()` are completely new, you just have to pass the vertex and pixel shader objects. Now render and present, that's all.

```

    //render scene with shaders (NEW)
    app.getDevice()->SetVertexDeclaration(vertexDecl);
    app.getDevice()->SetVertexShader(vertexShader);
    app.getDevice()->SetPixelShader(pixelShader);
    app.getDevice()->SetStreamSource(0, quadVB, 0, sizeof(D3DVERTEX));
    app.getDevice()->SetTexture(0, quadTexture);
    app.getDevice()->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);

    app.getDevice()->EndScene();
    app.getDevice()->Present(0, 0, 0, 0);
}
}

```

main.cpp - Cleaning up

The following code should speak for itself. Never forget to release your objects!

```

//clean up shaders (NEW)
if(constantTable)
{
    constantTable->Release();
    constantTable = NULL;
}

if(pixelShader)
{
    pixelShader->Release();
    pixelShader = NULL;
}

if(vertexShader)
{
    vertexShader->Release();
    vertexShader = NULL;
}

if(vertexDecl)
{
    vertexDecl->Release();
    vertexDecl = NULL;
}

//clean up scene data
if(quadTexture)
{
    quadTexture->Release();
    quadTexture = NULL;
}

if(quadVB)
{
    quadVB->Release();
    quadVB = NULL;
}

return 0;
}

```

vertex.vsh - The vertex shader

Let's have a look at the vertex shader. It is written in HLSL. At the beginning the input and output structures have to be declared. This is basically the data that is coming in from the vertex buffer, gets processed in the shader and then leaves it. Here, it is just the position and one texture coordinate for both structures. Global variables are passed through the constant table. The `vs_main` function is very similar to C code. The output structure is filled and then returned.

```
// Vertex shader input structure
struct VS_INPUT
{
    float4 Position    : POSITION;
    float2 Texture     : TEXCOORD0;
};

// Vertex shader output structure
struct VS_OUTPUT
{
    float4 Position    : POSITION;
    float2 Texture     : TEXCOORD0;
};

// Global variables
float4x4 WorldViewProj;

// Name: Simple Vertex Shader
// Type: Vertex shader
// Desc: Vertex transformation and texture coord pass-through
//
VS_OUTPUT vs_main( in VS_INPUT In )
{
    VS_OUTPUT Out;                //create an output vertex

    Out.Position = mul(In.Position,
                       WorldViewProj); //apply vertex transformation
    Out.Texture  = In.Texture;      //copy original texcoords

    return Out;                   //return output vertex
}
```

pixel.psh - The pixel shader

The pixel shader code is similar to the vertex shader. Position and texture coordinate input comes from the vertex shader. Output is just the color of the final pixel. A texture is the only global variable. Inside the `ps_main` function, the texel at the specified coordinate is fetched and then blended (multiplied) with a certain color.

```
// Pixel shader input structure
struct PS_INPUT
{
    float4 Position    : POSITION;
    float2 Texture     : TEXCOORD0;
};

// Pixel shader output structure
struct PS_OUTPUT
{
    float4 Color       : COLOR0;
};
```

```
// Global variables
sampler2D Tex0;

// Name: Simple Pixel Shader
// Type: Pixel shader
// Desc: Fetch texture and blend with constant color
//
PS_OUTPUT ps_main( in PS_INPUT In )
{
    PS_OUTPUT Out;                                //create an output pixel

    Out.Color = tex2D(Tex0, In.Texture);          //do a texture lookup
    Out.Color *= float4(0.9f, 0.8f, 0.4, 1);      //do a simple effect

    return Out;                                    //return output pixel
}
```