Beginning Direct3D Game Programming:

9. Shader Programming

jintaeks@gmail.com
Division of Digital Contents, DongSeo University.
May 2016
Vertex shaders are executable programs that process vertex data.

In the fixed function pipeline, Vertex processing takes primitive vertex data from model space and converts it into vertex data in projection space.
Vertex Processing

- Vertex processing converts data from model space to projection space.
Vertex Blending

- Blending combines one or more sets of vertex data to animate vertex data.
- Vertex blending uses one set of vertex data and one or more transforms (each with a corresponding weight).
After vertex processing, primitive processing and pixel processing convert per-vertex data into the final pixels that will be rendered.
■ **Clipping.** Clipping or removing geometry that’s outside of the viewing frustum to maximize rendering efficiency.

■ **Homogeneous divide.** Converting the x,y,z vertex data to nonhomogeneous space before sending the data to the rasterizer.

■ **Viewport scaling.** Scaling 2-D projection space to 2-D screen space.

■ **Triangle set up.** Preparing triangle-attribute interpolators and converting per-vertex attributes into per-pixel attributes.
The output of primitive processing is vertex data interpolated per pixel.

Pixel processing **blends** several per-pixel data types and **texture samples** (if any) into output pixel colors.
Part 1

- **Sample texture.** Sample one or more textures.
- **Blend.** Blend per-pixel attributes, typically diffuse and specular color and/or texture samples.
Part 2

■ Alpha test. Apply an alpha test to see if the pixel color will affect the final pixel color.

■ Depth test. Update the depth buffer with the pixel depth if the pixel is visible.

■ Stencil test. Apply a stencil test to see if the pixel will affect the final pixel color.

■ Per-pixel fog. Apply a fog color to the pixel color.

■ Alpha blend. Apply pixel alpha to create a transparent or semitransparent blend between a source pixel and a frame buffer pixel.

■ Dither. Use a dithering algorithm to blend adjacent pixels for a more consistent color range.

■ Gamma. Apply a gamma correction to the final frame buffer pixel color.
■ **Texture data.** Texture resources such as a texture file or a render target.

■ **Sampler.** Used to sample textures, which means to use the texture coordinates to look up a texture color.
  
  – Texture filtering influences the quality of the information (typically color) that is sampled from a texture.
Programmable Shader

✓ **Assembly Shader** in DirectX 8.

```cpp
; Lighting calculation
;
; Transform normal by world0 matrix
// dp4 r0.x, v3, c4
// dp4 r0.y, v3, c5
// dp4 r0.z, v3, c6
m3x3 r0, v3, c4
; Transform normal by world1 matrix
// dp4 r1.x, v3, c8
// dp4 r1.y, v3, c9
// dp4 r1.z, v3, c10
m3x3 r1, v3, c8
; Linear interpolate the two normals r0 and r1 into r2
// vs_1_1 code
// mul r0, r0, v1.x  ; v0 * weight
// add r2, c1.x, -v1.x  ; r2 = 1 - weight
// mad r2, r2, r1, r0  ; normal = (1-weight)*v1 + v0*weight
lrp r2.xyz, v1.x, r0.xyz, r1
; Do the lighting calculation
dp3 r1.x, r2.xyz, c20  ; r1 = normal dot light
```
HLSL: High-Level Shading Language

✓ HLSL is a shading language developed by Microsoft for the Direct3D 9 API to augment the shader assembly language.

✓ HLSL is analogous to the GLSL shading language used with the OpenGL standard. It is very similar to the Nvidia Cg shading language.

✓ HLSL programs come in five forms:
  – pixel shaders (fragment in GLSL)
  – vertex shaders
  – geometry shaders
  – compute shaders
  – tessellation shaders (Hull and Domain shaders).
Vertex Shader Comparison

✓ **VS 2.0** = DirectX 9.0 original **Shader Model 2** specification.
✓ **VS 2.0a** = NVIDIA GeForce FX/PCX-optimized model, DirectX 9.0a.
✓ **VS 3.0** = **Shader Model 3.0**, DirectX 9.0c.
✓ **VS 4.0** = **Shader Model 4.0**, DirectX 10.
✓ **VS 4.1** = **Shader Model 4.1**, DirectX 10.1.
✓ **VS 5.0** = **Shader Model 5.0**, DirectX 11.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td># of instruction slots</td>
<td>128</td>
<td>256</td>
<td>256</td>
<td>≥ 512</td>
<td>4096</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max # of instructions executed</td>
<td>128</td>
<td>1024</td>
<td>65536</td>
<td>65536</td>
<td>65536</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction predication</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Temp registers</td>
<td>12</td>
<td>12</td>
<td>16</td>
<td>32</td>
<td>4096</td>
<td></td>
<td></td>
</tr>
<tr>
<td># constant registers</td>
<td>≥ 96</td>
<td>≥ 256</td>
<td>256</td>
<td>≥ 256</td>
<td>16×4096</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static flow control</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Dynamic flow control</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Dynamic flow control depth</td>
<td>No</td>
<td>No</td>
<td>24</td>
<td>24</td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Vertex texture fetch</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td># of texture samplers</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>4</td>
<td>128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Geometry instancing support</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Bitwise operators</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Native integers</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>
Vertex Shader Example

VS_OUTPUT RenderSceneVS( float4 vPos : POSITION, float3 vNormal : NORMAL, float2 vTexCoord0 : TEXCOORD0 )
{
    VS_OUTPUT Output;
    float3 vNormalWorldSpace;
    float4 vAnimatedPos = vPos;

    // Transform the position from object space to homogeneous projection space
    Output.Position = mul(vAnimatedPos, g_mWorldViewProjection);

    // Transform the normal from object space to world space
    vNormalWorldSpace = normalize(mul(vNormal, (float3x3)g_mWorld));

    // Compute simple directional lighting equation
    float3 vTotalLightDiffuse = float3(0,0,0);
    vTotalLightDiffuse += g_LightDiffuse * max( 0, dot( vNormalWorldSpace, g_LightDir ) );
VS_OUTPUT RenderSceneVS( float4 vPos : POSITION,
    float3 vNormal : NORMAL,
    float2 vTexCoord0 : TEXCOORD0 )
{

    ... 
    // Compute simple directional lighting equation
    float3 vTotalLightDiffuse = float3(0,0,0);
    vTotalLightDiffuse += g_LightDiffuse * max( 0, dot( vNormalWorldSpace, g_LightDir ) );

    Output.Diffuse.rgb = g_MaterialDiffuseColor * vTotalLightDiffuse +
                        g_MaterialAmbientColor * g_LightAmbient;
    Output.Diffuse.a = 1.0f;

    // Just copy the texture coordinate through
    Output.TextureUV = vTexCoord0;

    return Output;
}
Pixel Shader Comparison

✓ **PS 2.0** = DirectX 9.0 original **Shader Model 2** specification.
✓ **PS 2.0a** = NVIDIA GeForce FX/PCX-optimized model, DirectX 9.0a.
✓ **PS 2.0b** = ATI Radeon X700, X800, X850, FireGL X3-256, V5000, V5100 and V7100 shader model, DirectX 9.0b.
✓ **PS 3.0** = **Shader Model 3.0**, DirectX 9.0c.
✓ **PS 4.0** = **Shader Model 4.0**, DirectX 10.
✓ **PS 4.1** = **Shader Model 4.1**, DirectX 10.1.
✓ **PS 5.0** = **Shader Model 5.0**, DirectX 11.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependent texture limit</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>Unlimited</td>
<td>8</td>
<td>Unlimited</td>
<td>Unlimited</td>
<td>Unlimited</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Texture instruction limit</td>
<td>4</td>
<td>6*2</td>
<td>32</td>
<td>Unlimited</td>
<td>Unlimited</td>
<td>Unlimited</td>
<td>Unlimited</td>
<td>Unlimited</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Position register</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Instruction slots</td>
<td>8+4</td>
<td>8+4</td>
<td>32 + 64</td>
<td>512</td>
<td>512</td>
<td>≥ 512</td>
<td>≥ 512</td>
<td>≥ 65536</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Executed instructions</td>
<td>8+4</td>
<td>6<em>2+8</em>2</td>
<td>32 + 64</td>
<td>512</td>
<td>512</td>
<td>65536</td>
<td>Unlimited</td>
<td>Unlimited</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Texture indirections</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>Unlimited</td>
<td>4</td>
<td>65536</td>
<td>Unlimited</td>
<td>Unlimited</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Interpolated registers</td>
<td>2 + 8</td>
<td>2 + 8</td>
<td>2 + 8</td>
<td>2 + 8</td>
<td>2 + 8</td>
<td>10</td>
<td>32</td>
<td>Unlimited</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Instruction predication</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Index input registers</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Temp registers</td>
<td>2</td>
<td>6</td>
<td>12 to 32</td>
<td>22</td>
<td>32</td>
<td>32</td>
<td>4096</td>
<td>4096</td>
<td>4096</td>
</tr>
<tr>
<td>Constant registers</td>
<td>8</td>
<td>8</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>224</td>
<td>16×4096</td>
<td>16×4096</td>
<td>16×4096</td>
</tr>
<tr>
<td>Arbitrary swizzling</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Gradient instructions</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Loop count register</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Pixel Shader Example

```plaintext
struct VS_OUTPUT
{
    float4 Position : POSITION; // vertex position
    float4 Diffuse : COLOR0;
    // vertex diffuse color (note that COLOR0 is clamped from 0..1)
    float2 TextureUV : TEXCOORD0; // vertex texture coords
};

PS_OUTPUT RenderScenePS( VS_OUTPUT In )
{
    PS_OUTPUT Output;

    // Lookup mesh texture and modulate it with diffuse
    Output.RGBColor = tex2D(MeshTextureSampler, In.TextureUV) * In.Diffuse;

    return Output;
}
```
A Microsoft DirectX effect enables the integration of vertex and pixel shaders with pipeline state to render objects.

Effects also provide a convenient way to write shaders for different hardware versions.

An effect can replace the vertex processing and part of the pixel processing performed by the graphics pipeline.
Effects and the 3D Pipeline

- Vertex and pixel processing can be performed by the fixed function pipeline, or can be implemented with programmable shaders.

- The input data tessellation, primitive processing, and data outputs are controlled by pipeline state. All this can be integrated into an effect.
Effects can save and restore states

✓ Shader state.
  – This includes creating and deleting shaders, setting shader constants, setting shader state, and rendering with shaders.

✓ Texture and sampler state.
  – This includes specifying texture files, initializing texture stages, creating sampler objects, and setting sampler state.

✓ Other pipeline state.
  – This includes states for setting transformations, lighting, materials, and rendering options.

✓ All rendering in an effect is done within a matching pair of ID3DXEffect::Begin and ID3DXEffect::End calls.
Effect State

✓ Effect state initializes the pipeline for processing pipeline data. Effect variables hold effect state values.

```csharp
technique TVertexShaderOnly_Asm
{
    pass P0
    {
        // lighting
        Lighting = FALSE;
        SpecularEnable = TRUE;
        // samplers
        Sampler[0] = (Sampler);
        // texture stages
        ColorOp[0] = MODULATE;
        ColorArg1[0] = TEXTURE;
        ColorArg2[0] = DIFFUSE;
        AlphaOp[0] = MODULATE;
    }
}
```
## Writing an Effect

- Writing an effect requires that you understand effect syntax, and generate the required state information.
- An effect is typically encapsulated into an effect file (.fx) but could also be written as a text string in an application.

<table>
<thead>
<tr>
<th>Global Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samplers and Structures</td>
</tr>
<tr>
<td>Vertex/Pixel Shader Functions</td>
</tr>
<tr>
<td>technique block</td>
</tr>
</tbody>
</table>
Simple Effect Example

// texture

texture Tex0 < string name = "tiger.bmp"; >;
sampler Sampler = sampler_state
{
    Texture = (Tex0);
    MipFilter = LINEAR;
    MinFilter = LINEAR;
    MagFilter = LINEAR;
};

float4x4 matWorldViewProj : WORLDVIEWPROJ;

struct VS_OUTPUT
{
    float4 Pos : POSITION;
    float2 Tex : TEXCOORD0;
};
VS_OUTPUT VS( float3 Pos : POSITION, float2 Tex : TEXCOORD0)
{
    VS_OUTPUT Out = (VS_OUTPUT)0;
    Out.Pos = mul(Pos, matWorldViewProj);
    Out.Tex = Tex;
    return Out;
}

technique TVertexShaderOnly_HLSL
{
    pass P0
    {
        // lighting
        Lighting = FALSE;
        SpecularEnable = TRUE;
        // samplers
        Sampler[0] = (Sampler);
        // texture stages
        ColorOp[0] = MODULATE;
    }
}
// texture stages
ColorOp[0] = MODULATE;
ColorArg1[0] = TEXTURE;
ColorArg2[0] = DIFFUSE;
AlphaOp[0] = MODULATE;
AlphaArg1[0] = TEXTURE;
AlphaArg2[0] = DIFFUSE;
ColorOp[1] = DISABLE;
AlphaOp[1] = DISABLE;

// shaders
VertexShader = compile vs_1_1 VS();
PixelShader = NULL;
}
Effect Example: BasicHLSL sample in DirectX Sdk

// Global variables

// Material's ambient color
float4 g_MaterialAmbientColor;

// Material's diffuse color
float4 g_MaterialDiffuseColor;

// Light's direction in world space
float3 g_LightDir;

// Light's diffuse color
float4 g_LightDiffuse;

// Light's ambient color
float4 g_LightAmbient;

// Color texture for mesh
texture g_MeshTexture;

// App's time in seconds
float g_fTime;

// World matrix for object
float4x4 g_mWorld;

// World * View * Projection matrix
float4x4 g_mWorldViewProjection;
Texture samplers, Vertex shader output structure

// Texture samplers

sampler MeshTextureSampler =

sampler_state
{
  Texture = <g_MeshTexture>;
  MipFilter = LINEAR;
  MinFilter = LINEAR;
  MagFilter = LINEAR;
};

// Vertex shader output structure

struct VS_OUTPUT
{
  float4 Position : POSITION;   // vertex position
  float4 Diffuse : COLOR0;      // vertex diffuse color (note that COLOR0 is clamped from 0..1)
  float2 TextureUV : TEXCOORD0; // vertex texture coords
};
Vertex shader

// This shader computes standard transform and lighting

VS_OUTPUT RenderSceneVS( float4 vPos : POSITION,
    float3 vNormal : NORMAL,
    float2 vTexCoord0 : TEXCOORD0,
    uniform int nUnused,
    uniform bool bTexture,
    uniform bool bAnimate )

{
    VS_OUTPUT Output;
    float3 vNormalWorldSpace;
    float4 vAnimatedPos = vPos;

    // Transform the position from object space to homogeneous projection space
    Output.Position = mul(vAnimatedPos, g_mWorldViewProjection);

    // Transform the normal from object space to world space
    vNormalWorldSpace = normalize(mul(vNormal, (float3x3)g_mWorld)); // normal (world space)
// Transform the normal from object space to world space
vNormalWorldSpace = normalize(mul(vNormal, (float3x3)g_mWorld)); // normal (world space)

// Compute simple directional lighting equation
float3 vTotalLightDiffuse = float3(0,0,0);
vTotalLightDiffuse += g_LightDiffuse * max(0, dot(vNormalWorldSpace, g_LightDir));

Output.Diffuse.rgb = g_MaterialDiffuseColor * vTotalLightDiffuse +
                    g_MaterialAmbientColor * g_LightAmbient;
Output.Diffuse.a = 1.0f;

// Just copy the texture coordinate through
if (bTexture )
    Output.TextureUV = vTexCoord0;
else
    Output.TextureUV = 0;

return Output;
// Pixel shader output structure

struct PS_OUTPUT
{
    float4 RGBColor : COLOR0; // Pixel color
};

// This shader outputs the pixel's color by modulating the texture's
// color with diffuse material color

PS_OUTPUT RenderScenePS( VS_OUTPUT In, uniform bool bTexture )
{
    PS_OUTPUT Output;

    // Lookup mesh texture and modulate it with diffuse
    if( bTexture )
        Output.RGBColor = tex2D(MeshTextureSampler, In.TextureUV) * In.Diffuse;
    else
        Output.RGBColor = In.Diffuse;

    return Output;
}
technique

// Renders scene to render target

//technique RenderSceneWithTexture1Light
{
    pass P0
    {
        VertexShader = compile vs_2_0 RenderSceneVS( 1, false, false );
        PixelShader  = compile ps_2_0 RenderScenePS( false ); // trivial pixel shader (could use FF instead if desired)
    }
}

//technique RenderSceneWithTexture2Light
{
    pass P0
    {
        VertexShader = compile vs_2_0 RenderSceneVS( 2, true, true );
        PixelShader  = compile ps_2_0 RenderScenePS( true ); // trivial pixel shader (could use FF instead if desired)
    }
}
Effects Contain One or More Techniques and Passes

```plaintext
technique T0
{
  pass P0
  { ...
  }
}

technique T1
{
  pass P0
  { ...
  }

  pass P1
  { ...
  }
}
```
Using an Effect: Creating an Effect

ID3DXEffect* g_pEffect = NULL;
DWORD dwShaderFlags = 0;

    dwShaderFlags |= D3DXSHADER_FORCE_VS_SOFTWARE_NOOPT;
dwShaderFlags |= D3DXSHADER_FORCE_PS_SOFTWARE_NOOPT;
dwShaderFlags |= D3DXSHADER_NO_PRESHADER;
// Read the D3DX effect file
WCHAR str[MAX_PATH];
DXUTFindDXSDKMediaFileCch( str, MAX_PATH, L"BasicHLSL.fx" );

D3DXCreateEffectFromFile(  
pd3dDevice,  
str,  
NULL, // CONST D3DXMACRO* pDefines,  
NULL, // LPD3DXINCLUDE pInclude,  
dwShaderFlags,  
NULL, // LPD3DXEFFECTPOOL pPool,  
&g_pEffect,  
NULL );
// Apply the technique contained in the effect
renderEffect->Begin(&cPasses, 0);

for (iPass = 0; iPass < cPasses; iPass++)
{
    renderEffect->BeginPass(iPass);

    // Only call CommitChanges if any state changes have happened
    // after BeginPass is called
    renderEffect->CommitChanges();

    // Render the mesh with the applied technique
    mesh->DrawSubset(0);

    renderEffect->EndPass();
}
renderEffect->End();
Use Semantics to Find Effect Parameters

✓ In effect file, the **semantic** is located following a colon (:) after the parameter name. For instance:

```plaintext
float4x4 matWorldViewProj : WORLDVIEWPROJ;
```

✓ The effect interface can use a semantic to get a handle to a particular effect parameter. For instance:

```plaintext
D3DHANDLE handle = m_pEffect->GetParameterBySemantic(NULL, "WORLDVIEWPROJ");
```
Add Parameter Information with Annotations

✓ **Annotations** are user-specific data that can be attached to any technique, pass, or parameter.
  – An annotation is a flexible way to add information to individual parameters.

✓ **Annotation declarations** are delimited by angle brackets. An annotation contains:
  – A data type.
  – A variable name.
  – An equals sign (=).
  – The data value.
  – A ending semicolon (;).

```
texture Tex0 < string name = "tiger.bmp"; >;
```
The annotation is attached to the texture object and specifies the texture file that should be used to initialize the texture object.

The annotation does not initialize the texture object, it is simply a piece of user information.

An application can read the annotation with `ID3DXBaseEffect::GetAnnotation`. 

```cpp
texture Tex0 < string name = "tiger.bmp"; >;
```
m_pEffect->SetTechnique( "RenderScene" );

m_pEffect->BeginParameterBlock();
D3DXVECTOR4 v4( Diffuse.r, Diffuse.g, Diffuse.b, Diffuse.a );
m_pEffect->SetVector( "g_vDiffuse", &v4 );
m_pEffect->SetFloat( "g_fReflectivity", fReflectivity );
m_pEffect->SetFloat( "g_fAnimSpeed", fAnimSpeed );
m_pEffect->SetFloat( "g_fSizeMul", fSize );
m_hParameters = m_pEffect->EndParameterBlock();
Practice: Build BasicHLSL sample project in Sdk

✓ Examine all statements related to 'g_pEffect'.

```c
ID3DXEffect* g_pEffect = NULL; // D3DX effect interface
```

✓ Examine all statements in the effect file related to 'g_pEffect'.
Fix HLSL error

![Visual Studio interface with Solution Explorer and Shader properties highlighted]
✓ Replace 'tiny.x' mesh with 'teapot.x'
✓ Disable the vertex animation in effect file.
✓ Modify diffuse color of lights and check the result.
References

MY BRIGHT FUTURE
동서대학교
DSU Dongseo University 동서대학교