Chapter 6. HLSL/DirectX

Pedro Sander

DirectX® 9 High Level Shading Language

Jason Mitchell
ATI Research

Introduction

One of the most empowering new components of DirectX 9 is the High Level Shading Language (HLSL). Using this standard high level language, shader writers are able to think at the algorithm level while implementing shaders, rather than worry about meddlesome hardware. The HLSL also has all of the usual advantages of a high level language such as easy code reuse, improved readability and optimizations provided by the compiler.

Assembly Language and Compile Targets

Unlike OpenGL, in which high level shader code is passed directly to the driver, the Direct3D HLSL is really independent of Direct3D itself and actually a convenient service of the D3DX library. When shaders were first added to DirectX in version 8.0, several virtual shader machines were defined—each roughly corresponding to a particular graphics processor produced by each of the top 3D graphics hardware vendors. In DirectX 8.0 and DirectX 8.1, programs written to these shader models (named vs_1_1 and ps_1_1 through ps_1_4) were relatively short and were generally written by developers directly in the appropriate assembly language. As shown on the left side of Figure 1, the application would pass this human-readable assembly language code to the D3DX library via `D3DXAssembleShader()`and get back a binary representation of the shader which would in turn be passed to Direct3D via `CreatePixelShader()` or `CreateVertexShader()`.

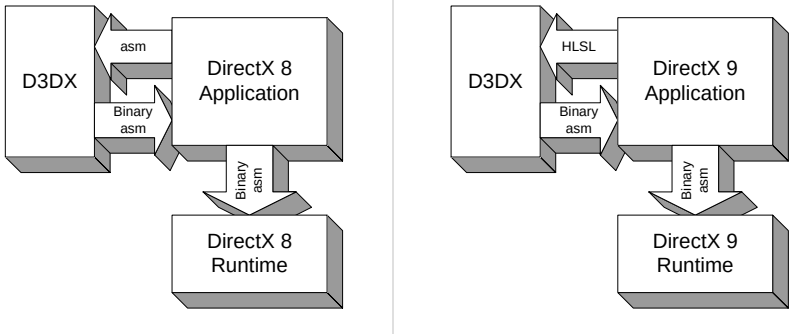

Figure 1 – Use of D3DX for Assembly and Compilation in DirectX 8 and DirectX 9

As shown on the right side of Figure 1, the situation in DirectX 9 is very similar in that the application passes an HLSL shader to D3DX via the `D3DXCompileShader()` API and gets back a binary representation of the compiled shader which is in turn passed to Direct3D via `CreatePixelShader()` or `CreateVertexShader()`. The binary asm code generated is a function only of the compile target chosen, not the specific graphics device in the user's or developer's system. That is, the binary asm which is generated is vendor-neutral and will be the same no matter where you compile or run it. In fact, the Direct3D runtime itself does not know anything about HLSL, only the binary assembly shader models. This is nice because it means that the HLSL compiler can be updated independent of the Direct3D runtime. In fact, Microsoft has already released updates to the compiler since the ship of DirectX 9 in December 2002.

In addition to the development of the HLSL compiler in D3DX, DirectX 9.0 also introduced additional assembly-level shader models to expose the functionality of the latest generation of 3D graphics hardware. Application developers can feel free to work directly in the assembly languages for these new models (vs_2_0, vs_3_0, ps_2_0 and ps_3_0) but we expect most developers to move wholesale to HLSL for shader development.

Hardware Realities

Of course, just because you can write an HLSL program to express a particular shading algorithm doesn't mean it will run on a given piece of hardware. As we discussed earlier, an application calls D3DX to compile an HLSL shader to binary asm via the `D3DXCompileShader()` API. One of the parameters to this API entrypoint is a parameter which defines which of the assembly language models (or *compile targets*) the HLSL compiler should use to express the final shader code. If an application is doing HLSL shader compilation at run time (as opposed to offline), the application could examine the capabilities of the Direct3D device and select the compile target to match. If the algorithm expressed in the HLSL shader is too complex to execute on the selected compile target, compilation *will fail*. What this means is that while HLSL is a huge benefit to shader development, it does not free developers from the realities of shipping games to a target audience which owns graphics devices of varying capabilities. As a developer, you still have to manage a tiered approach to your visuals, writing better shaders for better graphics cards and more basic versions for older cards. With well-written HLSL, however, this burden can be eased significantly.

Compilation Failure

As mentioned above, failure of a given HLSL shader to compile for a particular compile target is an indication that the shader is too complex for the compile target. This can mean that the shader either requires too many resources or it requires some capability, such as dynamic branching, that is not supported by the chosen compile target. For example, an HLSL shader could be written to access a given texture map six times in a shader. If this shader is compiled for the ps_1_1 compile target, compilation will fail since the ps_1_1 model supports only four textures. Another common source of

compilation failure is exceeding the maximum instruction count of the chosen compile target. An algorithm expressed in HLSL may simply require too many instructions to be executed by a given compile target.

It is important to note that the choice of compile target does not restrict the HLSL syntax that a shader writer can use. For example, a shader writer can use 'for' loops, subroutines, 'if-else' statements etc. and still compile for targets which don't natively support looping, branching or 'if-else' statements. In such cases, the compiler will unroll loops, inline function calls and execute both branches of an 'if-else' statement, selecting the proper result based upon the original value used in the 'if-else' statement. Of course, if the resulting shader is too long or otherwise exceeds the resources of the compile target, compilation will fail.

A Simple Example

Let's have a look at one HLSL vertex shader and one HLSL pixel shader taken from an application which renders simple procedural wood. The first HLSL shader shown below is a simple vertex shader:

```
float4x4 view_proj_matrix;
float4x4 texture_matrix0;

struct VS_OUTPUT
{
   float4 Pos     : POSITION;
   float3 Pshade  : TEXCOORD0;
};


VS_OUTPUT main (float4 vPosition : POSITION)
{
   VS_OUTPUT Out = (VS_OUTPUT) 0;

   // Transform position to clip space
   Out.Pos = mul (view_proj_matrix, vPosition);

   // Transform Pshade
   Out.Pshade = mul (texture_matrix0, vPosition);

   return Out;
}
```

The first two lines of this shader declare a pair of 4×4 matrices called `view_proj_matrix` and `texture_matrix0`. Following these global-scope matrices, a structure is declared. This `VS_OUTPUT` structure has two members: a `float4` called `Pos` and a `float3` called `Pshade`.

The main function for this shader takes a single `float4` input parameter and returns a `VS_OUTPUT` structure. The `float4` input `vPosition` is the sole input to the shader while the returned `VS_OUTPUT` struct defines this vertex shader's output. For now, don't worry about the `POSITION` and `TEXCOORD0` keywords following these parameters and structure members. These are called *semantics* and their meaning will be discussed later in this chapter.

Looking at the actual code body of the main function, you'll see that an intrinsic function called `mul` is used to multiply the input `vPosition` vector by the `view_proj_matrix` matrix. This intrinsic is very commonly used in vertex shaders to perform vector-matrix multiplication. In this case, `vPosition` is treated as a column vector since it is the second parameter to `mul`. If the `vPosition` vector were the first parameter to `mul`, it would be treated as a row vector. The `mul` intrinsic and other intrinsics will be discussed in more detail later in the chapter. Following the transformation of the input position `vPosition` to clip space, `vPosition` is multiplied by another matrix called `texture_matrix0` to generate a 3D texture coordinate. The results of both of these transformations have been written to members of a `VS_OUTPUT` structure, which is returned. A vertex shader must always output a clip-space position at a minimum. Any additional values output from the vertex shader are interpolated across the rasterized polygon and are available as inputs to the pixel shader. In this case, the 3D `Pshade` is passed from the vertex to the pixel shader via an interpolator.

Below, we see a simple HLSL procedural wood pixel shader. This pixel shader, which is written to work with the vertex shader we just described, will be compiled for the ps_2_0 target.

```
float4 lightWood; // xyz == Light Wood Color
float4 darkWood;  // xyz == Dark Wood Color
float  ringFreq;  // ring frequency

sampler PulseTrainSampler;

float4 hlsl_rings (float4 Pshade : TEXCOORD0) : COLOR
{
    float scaledDistFromZAxis = sqrt(dot(Pshade.xy, Pshade.xy)) * ringFreq;

    float blendFactor = tex1D (PulseTrainSampler, scaledDistFromZAxis);

    return lerp (darkWood, lightWood, blendFactor);
}
```

The first few lines of this shader are the declaration of a pair of floating-point 4-tuples and one scalar `float` at global scope. Following these variables, a sampler called `PulseTrainSampler` is declared. Samplers will be discussed in more detail later in the chapter but for now you can just think of a sampler as a window into video memory with associated state defining things like filtering, and texture coordinate addressing modes. With variable and sampler declarations out of the way, we move on to the body of the shader code. You can see that there is one input parameter called `Pshade`, which is interpolated across the polygon. This is the value that was computed at each vertex by the vertex shader above. In the pixel shader, the Cartesian distance from the shader-space *z* axis is computed, scaled and used as a 1D texture coordinate to access the texture bound to the `PulseTrainSampler`. The scalar color that is returned from the `tex1D()` sampling function is used as a blend factor to blend between the two constant colors (`lightWood` and `darkWood`) declared at global scope of the shader. The 4D vector result of this blend is the final output of the pixel shader. All pixel shaders must return a 4D RGBA color at a minimum. We will discuss additional optional pixel shader outputs later in the chapter.

Shader Inputs

Vertex and pixel shaders have two types of input data, *varying* and *uniform*. The varying input is the data that is unique to each execution of a shader. For a vertex shader, the varying data (i.e. position, normals, etc.) comes from the vertex streams. The uniform data (i.e. material color, world transform, etc.) is constant for multiple executions of a shader. If you are familiar with the assembly models, uniform data is specified in constant registers, and varying data in the 'v'/'t' registers in vertex and pixel shaders.

Uniform input

Uniform data can be specified by two methods in HLSL. The most common method is to declare global variables and use them within the vertex or pixel shaders. Any use of a global variable within a shader will result in the addition of the variable to a list of uniform variables required by the shader. The second method is to mark an input parameter of the top-level shader function as uniform. This marking specifies that the given variable should be added to the list of uniform variables used by the shader. Both of these methods are illustrated in the following code snippet:

```
// Declare a global uniform variable
// Appears in constant table under name 'UniformGlobal'
float4 UniformGlobal;

// Declare a uniform input parameter
// Appears in constant table under name '$UniformParam'
float4 main( uniform float4 UniformParam ) : POSITION
{
   return UniformGlobal * UniformParam;
}
```

Varying input

Varying data is specified by marking the input parameters of the top-level shader function with an input semantic. All top-level shader inputs must either be marked as varying by using semantics or marked with the keyword 'uniform' indicating that the value is constant for the execution of the shader. If a top-level shader input is not marked with a semantic or 'uniform' keyword, then the shader will fail to compile.

The input semantic is a name used to link the given shader input to an output of the previous stage of the graphics pipeline. For example, the input semantic `POSITION0` is used by vertex shaders to specify where the position data from the vertex buffer should be linked.

Pixel and vertex shaders have different sets of input semantics due to the different parts of the graphics pipeline that feed into each shader unit. Vertex shader input semantics describe the per vertex information to be loaded from a vertex buffer into a form that can be consumed by the vertex shader (i.e. positions, normals, texture

coordinates, colors, tangents, binormals, etc). These input semantics directly map to the combination of the D3DDECLUSAGE enum and UsageIndex that is used to describe vertex data elements in a vertex buffer.

Pixel shader input semantics describe the information that is provided per pixel by the rasterization unit. This data is generated by interpolating between the outputs of the vertex shader for each vertex of the current primitive. The basic pixel shader input semantics link the input color and texture coordinate information to input parameters.

Input semantics can be assigned to shader input by two methods. The first method is by appending a colon, ':', and the input semantic name to the input parameter declaration. The second method is to define an input structure with input semantics assigned to each element of the input structure.

Input semantic example:

```
// Declare an input structure with a semantic binding
struct InStruct
{
      float4 Pos1 : POSITION1
};

// Declare the Pos variable as containing position data
float4 main( float4 Pos : POSITION0, InStruct In ) : POSITION
{
      return Pos * In.Pos1;
}

// Declare the Col variable as containing the interpolated COLOR0 value
float4 mainPS( float4 Col : COLOR0 ) : COLOR
{
      return Col;
}
```

Shader Outputs

Vertex and pixel shaders provide output data to the subsequent graphics pipeline stage. Output semantics are used to specify how data generated by the shader should be linked to the inputs of the next stage. For example, the output semantics for a vertex shader are used to link the outputs with the interpolators in the rasterizer to generate the input data for the pixel shader. The pixel shader outputs are the values provided to the alpha blending unit for each of the render targets or the depth value to be written to the depth buffer.

Vertex shader output semantics are used to link the shader both to the pixel shader and the rasterizer stage. The POSITION output is a required output from each vertex shader that is consumed by the rasterizer and not exposed to the pixel shader. TEXCOORD*n* and COLOR*n* denote outputs that are made available to the pixel shader post interpolation.

Pixel shader output semantics bind the output colors of a pixel shader with the correct render target. The colors output from the pixel shader are linked to the alpha blend stage, which determines how the destination render targets are modified. The `DEPTH` output semantics can be used to change the destination depth value at the current raster location. NOTE: `DEPTH` and multiple render targets (also known as "MRT") are only supported with some shader models.

The syntax for output semantics is identical to the syntax for specifying input semantics. The semantics can be either specified directly on parameters declared as 'out' parameters, or assigned during the definition of a structure that is either returned as an out parameter or the return value of the function.

The following code snippets illustrate the variety of ways in which data can be output from HLSL shaders:

```
// Declare an output structure with a semantic binding
struct OutStruct
{
      float2 Tex2 : TEXCOORD2
};

// Declare the Tex0 out parameter as containing TEXCOORD0 data
float4 main(out float2 Tex0 : TEXCOORD0, out OutStruct Out ) : POSITION
{
      Tex0 = float2(1.0, 0.0);
      Out.Tex2 = float2(0.1, 0.2);
      return float4(0.5, 0.5, 0.5, 1);
}

// Declare the Col variable as containing the interpolated COLOR0 value
float4 mainPS( out float4 Col1 : COLOR1) : COLOR
{
      // write out to render target 1 using out parameter
      Col1 = float4(0.0, 0.0, 0.0, 0.0);

      // write to render target 0 using the declared return destination
      return float4(1.0, 0.9722, 0.3333334, 0);
}
```

```
struct PS_OUT
{
      float4 Color: COLOR;
      float  Depth: DEPTH;
};

//
// Three different ways to output from a pixel shader:
//

PS_OUT PSFunc1() { ... }

void PSFunc2(out float4 Color : COLOR,
             out float Depth : DEPTH)
{
  ...
}

void PSFunc3(out PS_OUT Out)
{
  ...
}
```

An Example Shader

Now that we've discussed the language itself and how it connects with the rest of the graphics pipeline via inputs and outputs, we will discuss an example shader called NPR Metallic. We call it this since it was designed to look like a metallic surface which would exist in a world rendered in a cel-animation style (Figure 2).

Figure 2 – NPR Metallic

First, let's look at the NPR Metallic vertex shader written in HLSL:

```
float4x4 view_proj_matrix;

float4 view_position;
float4 light0;
float4 light1;
float4 light2;

struct VS_OUTPUT
{
   float4 Pos   : POSITION;
   float3 View  : TEXCOORD0;
   float3 Normal: TEXCOORD1;
   float3 Light1: TEXCOORD2;
   float3 Light2: TEXCOORD3;
   float3 Light3: TEXCOORD4;
};

VS_OUTPUT main( float4 inPos    : POSITION,
                float3 inNorm   : NORMAL )
{
   VS_OUTPUT Out = (VS_OUTPUT) 0;

   // Output transformed vertex position:
   Out.Pos = mul( view_proj_matrix, inPos );

   Out.Normal = inNorm;

   // Compute the view vector:
   Out.View = normalize( view_position - inPos );

   // Compute vectors to three lights from the current vertex position:
   Out.Light1 = normalize (light0 - inPos);   // Light 1
   Out.Light2 = normalize (light1 - inPos);   // Light 2
   Out.Light3 = normalize (light2 - inPos);   // Light 3

   return Out;
}
```

NPR Metallic Vertex Shader

The first thing that we see in this vertex shader is the declaration of a matrix and a set of floats at global scope: `view_proj_matrix`, `view_position`, `light0`, `light1`, and `light2`. These are all implicitly uniform variables which are externally settable by the API and modifiable in the shader itself.

Following these global variables, we see the definition of a structure called `VS_OUTPUT`, which is also the return type of our main function. This means that this vertex shader will output five 3D texture coordinates in addition to the required 4D position.

Looking at the main function, we see that the vertex shader takes a 4D vector as input position, a 3D vector as input normal and a 2D vector as a texture coordinate. The input position, `inPos`, is transformed by the `view_proj_matrix` using the `mul()` intrinsic, while the normal, `inNorm`, is passed through to the output untouched.

Finally, 3D vectors from the object space vertex position to the three lights and the view position are all computed. These 3D vectors are passed to the `normalize()` intrinsic to guarantee that they are of unit length. These normalized 3D vectors are all output from the vertex shader as 3D texture coordinates which will be interpolated across the polygon.

To reinforce the earlier discussion about compile targets and assembly models, we will compile this shader and have a look at the assembly output. First, we have written the above code into a file called NPRMetallic.vhl. Next, we can compile it on the commandline with fxc:

```
fxc -nologo -T vs_1_1 -Fc -Vd NPRMetallic.vhl
```

Because this vertex shader does not require flow control, we select the vs_1_1 compile target. We also set the flags to generate a code file and disable validation. A portion of the generated code file is shown below:

```
// Parameters:
//     float4 light0;
//     float4 light1;
//     float4 light2;
//     float4 view_position;
//     float4x4 view_proj_matrix;
//
// Registers:
//     Name             Reg   Size
//     ---------------- ----- ----
//     view_proj_matrix c0       4
//     view_position    c4       1
//     light1           c5       1
//     light2           c6       1
//     light0           c7       1

    vs_1_1
    dcl_position v0
    dcl_normal v1
    mul r0, v0.x, c0
    mad r2, v0.y, c1, r0
    mad r4, v0.z, c2, r2
    mad oPos, v0.w, c3, r4
    add r1, -v0, c4
    dp4 r1.w, r1, r1
    rsq r1.w, r1.w
    mul oT0.xyz, r1, r1.w
    add r8, -v0, c7
    dp4 r8.w, r8, r8
    rsq r8.w, r8.w
    mul oT2.xyz, r8, r8.w
    add r3, -v0, c5
    add r10, -v0, c6
    dp4 r3.w, r3, r3
    rsq r3.w, r3.w
    mul oT3.xyz, r3, r3.w
    dp4 r10.w, r10, r10
    rsq r10.w, r10.w
    mul oT4.xyz, r10, r10.w
    mov oT1.xyz, v1
```

At the top of the code file, we see the parameters to this vertex shader. That is, we see the global scope variables which will need to be set from the API for this shader to work properly in a given application. The next section shows the hardware registers to which these parameters must be loaded by the application for the assembly shader to work properly. Next, we have the shader code itself, which as been compiled to 21 assembly instructions. We won't go through all of the code, but you should take note of the `dcl_position` and `dcl_normal` statements, which are a direct result of the `POSITION` and `NORMAL` semantics on the inputs to the shader's main function. Additionally, note the storage of final results in the `oPos`, `oT0`, `oT1`, `oT2`, `oT3` and `oT4` registers. This is caused by the return type of the function being a structure whose members are tagged with the corresponding semantics. While not strictly necessary, knowing how to use fxc to generate assembly code from HLSL and how to read through it can be beneficial at some stages of development, particularly when trying to write more optimal HLSL.

Now that we have used the vertex shader to transform the geometry into clip space and define the values which will be interpolated across the polygons, we can move on to the pixel shader which will make use of all of these interpolated quantities.

```
float4 Material;

sampler Outline;

float4 main( float3 View:   TEXCOORD0,
             float3 Normal: TEXCOORD1,
             float3 Light1: TEXCOORD2,
             float3 Light2: TEXCOORD3,
             float3 Light3: TEXCOORD4 ) : COLOR
{
   // Normalize input normal vector:
   float3 norm = normalize (Normal);

   float4 outline = tex1D(Outline, 1 - dot (norm, normalize(View)));

   float lighting = (dot (normalize (Light1), norm) * 0.5 + 0.5) +
                    (dot (normalize (Light2), norm) * 0.5 + 0.5) +
                    (dot (normalize (Light3), norm) * 0.5 + 0.5);

   return outline * Material * lighting;
}
```

NPR Metallic Pixel Shader

As before, we see that this shader has declared some variables at global scope. In this case, we have a 4D vector `Material` which defines material values for the object to be rendered, and a single sampler `Outline` which we will use to access a special texture used for outlining the object. The five 3D texture coordinates computed in the vertex shader are the inputs to the main function of this pixel shader and define the view vector, the normal vector and three light vectors.

Since the texture coordinates are *linearly* interpolated across the polygon, it is possible for them to contain non-normalized values at a given pixel. Thus, this shader first renormalizes the interpolated normal vector using the `normalize()` intrinsic. Subsequently, the outline texture is sampled using the dot product of the normalized

normal and view vectors. The lighting is then computed by summing a series of scaled and biased dot products of the normal with normalized light vectors.

In the last line of this pixel shader, we return the product of the variables `outline`, `Material` and `lighting`. The first two of these are 4D vectors while the last is a scalar. If you recall from our earlier discussion of type casting, the multiplication of the scalar by a vector temporarily promotes the scalar to a vector whose components are all equivalent to the original scalar. That is, the following two expressions are equivalent:

```
return outline * Material * lighting;

return outline * Material * float4(lighting, lighting, lighting, lighting);
```

Thus, the end result is that all of the channels are multiplied by the scalar `lighting`, giving us the final result you see in Figure 2.

As we did with the NPRMetallic vertex shader, we will generate a code file for the pixel shader using fxc:

```
fxc -nologo -T ps_2_0 -Fc -Vd NPRMetallic.phl
```

This compilation uses the same flags as before but is compiled for the ps_2_0 target. The resulting 29 instruction shader is shown below:

```
// Parameters:
//     float4 Material;
//     sampler Outline;
//
// Registers:
//     Name         Reg   Size
//     ------------ ----- ----
//     Material     c0       1
//     Outline      s0       1

    ps_2_0
    def c1, 1, 0, 0, 0.5
    dcl t0.xyz
    dcl t1.xyz
    dcl t2.xyz
    dcl t3.xyz
    dcl t4.xyz
    dcl_2d s0
    dp3 r0.w, t1, t1
    rsq r2.w, r0.w
    mul r9.xyz, r2.w, t1
    dp3 r9.w, t0, t0
    rsq r9.w, r9.w
    mul r4.xyz, r9.w, t0
    dp3 r9.w, r9, r4
    add r11.xy, -r9.w, c1.x
    texld r6, r11, s0
    dp3 r9.w, t2, t2
    rsq r9.w, r9.w
    mul r1.xyz, r9.w, t2
    dp3 r9.w, r1, r9
    mad r9.w, r9.w, c1.w, c1.w
```

```
    dp3 r8.w, t3, t3
    rsq r10.w, r8.w
    mul r5.xyz, r10.w, t3
    dp3 r0.w, r5, r9
    mad r9.w, r0.w, c1.w, r9.w
    add r9.w, r9.w, c1.w
    dp3 r2.w, t4, t4
    rsq r11.w, r2.w
    mul r1.xyz, r11.w, t4
    dp3 r8.w, r1, r9
    mad r10.w, r8.w, c1.w, r9.w
    add r5.w, r10.w, c1.w
    mul r6, r6, r5.w
    mul r0, r6, c0
    mov oC0, r0
```

As before, the variables (in this case the constant `Material` and the sampler `Outline`) are listed at the top of the file. These must be set properly by the application via the API in order for the shader to function correctly.

After the `ps_2_0` instruction, there is a `def` instruction of some magic constants. This `def` instruction is a free instruction which appears in the actual assembly instruction stream that defines constants which will be used by the subsequent ALU operations. This kind of constant definition is generally the result of literal values appearing in the HLSL shader, as in the following statements taken from the NPRMetallic pixel shader:

```
...
1 - dot (norm, normalize(View)
...
dot (normalize (Light1), norm) * 0.5 + 0.5
...
```

Following this constant definition, there are five 3D texture coordinate declarations of the form `dcl tn.xyz`. As in the vertex shader, these are a result of the semantics of the input parameters to this HLSL shader's main function. Following the texture coordinate declarations, there is a sampler declaration `dcl_2d s0`. This indicates that a 2D texture must be bound to sampler zero. This may seem odd since the `tex1D()` intrinsic was used in the HLSL shader. This discrepancy exists since there is no such thing as a 1D texture in the Direct3D API or shader assembly language. The `tex1D()` intrinsic is actually just a way for the HLSL shader writer to indicate to the compiler that only one component of the texture coordinate needs to be populated, shaving off an assembly instruction in some cases.

Now that you are familiar with some of the correspondence between HLSL and assembly code, we will discuss optimization strategies so that you can be sure that you are writing the best HLSL possible.

Optimization

While the DirectX HLSL compiler has an excellent optimizer built in, there are things you can do as an HLSL coder to help shave off a few more cycles here and there. While this is probably more of an academic exercise in the long term, it may make the difference between being able to target a legacy 1.*x* shader model using HLSL or not.

The most important thing to remember about writing high performance shaders is that the compiler is required to do what you ask it to. That is, if you write your shader to require a certain number of math operations or a particular value in an output component, then it will need to perform those operations. The compiler is smart about removing dead code, but it cannot know about values that do not ultimately matter due to circumstances outside of a given shader. For example, if the pixel shader is not using the second texture coordinate, the vertex shader probably shouldn't compute it. The HLSL compiler, of course, has no way of knowing this when you compile the vertex shader. Additionally, you may know that you will always use an $n \times 1$ function lookup texture at a given sampler and hence it is not necessary to compute the 2nd texture coordinate for use in the sampling intrinsic. If you use the `tex2D()` intrinsic, however, the HLSL compiler will require you to compute the 2nd texture coordinate even though it is ultimately unnecessary. The compiler is designed to build an assembly program that does exactly what you asked without making any visual quality vs performance tradeoffs.

Another extremely important objective for high performance shaders is to make sure that a computation only runs at the required frequency. If you can get away with doing a calculation per-vertex rather than per-pixel, then do so. These types of operations are where the biggest wins often come from. The same optimization is true for operations on values which are uniform (i.e. operations that do not change for the entire execution of the shader). An example of this would be pre-multiplying the world ambient color value by an object's material ambient value and passing their product to the shader instead of redundantly calculating the product per vertex or per pixel.

The following sections go into some detail on how language features are mapped into assembly constructs. While it is not necessary to understand how to write vertex or pixel shader assembly, it can be quite helpful to understand the basic limitations and efficiencies of the assembly models. Understanding key assembly features is essential to generating compact and efficient shaders.

Matrix Datatype Usage

One of HLSL's more obvious departures from the C standard is the introduction of vector and matrix datatypes. The datatypes were added to enable easier writing of code and to enable intrinsic functions to work properly, but correct usage of the datatypes allows for better code generation as well. The usage of vector types enables the compiler to more easily use all of the capabilities of the vector instructions. The compiler will

automatically vectorize scalar operations when possible, but in general it is better to write your HLSL code in a vector-friendly form for best performance.

Although you can implement shaders with arrays of vectors instead of matrices, the recommended way to store a matrix is with a matrix datatype. By using a matrix datatype, the compiler has the choice to store internal matrices in either column major or row major order depending on how the matrix is used. This optimization can be quite useful for situations in which a matrix is generated in either a pixel or vertex shader. As mentioned earlier, for input matrices, the compiler always uses either column major or row major storage format based on a compiler flag, with column major being the default method.

Integer Datatype Usage

The `int` datatype is important to understand and use correctly in HLSL. It is very easy to generate extra instructions by using the `int` datatype in places that it should not be used. The `int` datatype was added to HLSL to make relative addressing familiar as well as efficient. A problem with using `float` datatypes for addressing purposes without truncation rules is that incorrect access to arrays can occur. For example, if the index variable is 2.5 and a `float4x4` matrix is being accessed, half of matrix 2 and half of matrix 3 will be used instead of truncating to access matrix 2 before accessing the matrix. In order to fix this, all floats that are used for accessing arrays must be rounded before being multiplied by the size of each element. This can be an expensive operation, since correct C rounding rules are not easily accomplished using the available instructions.

In order to avoid unwanted rounding or truncation operations, the `int` datatype was added to mark values as being integer values. In order to properly avoid treating input data incorrectly as floating point data, all inputs that are going to be used as integers should be defined as `int`s. For example, matrix palette indices read from a vertex stream component should be marked as `int`s. Declaring an input as `int` is a “free” operation in that no truncation will be performed and the value is assumed to be an integer value. If the input is not declared as an `int`, the shader will not do what you expect. If, on the other hand, you cast a `float` to `int` in your shader or use a `float` for addressing purposes, a truncation will happen. Casting non-`int` intermediate values to `int` will also result in truncation overhead.

```
OutPos = mul(Pos, WorldArray[Index]);
```

```
// Index declared as float
frc r0.w, r1.w
add r2.w, -r0.w, r1.w
mul r9.w, r2.w, c61.x
mova a0.x, r9.w
m4x4 oPos, v0, c0[a0.x]
```

```
// Index declared as int
mul r0.w, c60.x, r1.w
mova a0.x, r0.w
m4x4 oPos, v0, c0[a0.x]
```

Code generated with float index vs integer index

Flow Control and Performance

Most current vertex and pixel shader hardware does not support flow control. The hardware is designed to run a shader linearly, executing each instruction once. Newer hardware supports limited forms of flow control: *static branching*, *predicated instructions*, *static looping*, *dynamic branching*, and *dynamic looping*. Since HLSL can be compiled down to any or all of the models that support various degrees of flow control, it must be taken into consideration when writing shaders designed to run on more restricted models. As mentioned earlier, no restrictions are placed on the syntax of HLSL based on the compile target, but compile time errors will occur if a shader is impossible to implement on the compile target used.

Loops are a form of flow control that occur quite often in shaders. Some hardware allows for either static or dynamic looping, but most require linear execution. On the models that do not support looping, all loops must be unrolled. While this can be an expensive operation, it can be used to generate excellent code with minimal effort. A good example is the DepthOfField sample from the DirectX 9 SDK that uses unrolled loops even for ps_1_1 shaders. In order to write high performance shaders, you should keep this in mind, either for using the compiler to do the work of unrolling for you, or realizing when shaders will become unbounded and perform poorly or exceed instruction limits.

Using ‘if’ statements can have large performance implications due to the lack of support for branching in most assembly-level shader models. In models that do not support any form of branching, both sides of an ‘if’ must be executed and the output chosen based on which side of the ‘if’ would have been taken. Having come from the CPU programming world, this form of execution is a bit different than most HLSL shader writers would expect. Common optimizations that you would use on a CPU to avoid expensive operations will not work as expected on shader models that don’t support branches, since both the expensive path and the cheap path will be executed. Some shader models support different levels of branching: *predicated instructions*, *static if blocks*, and *dynamic if blocks*.

Example using if in vs_1_1:

```
if (Value > 0)
    Position = Value1;
else
    Position = Value2;
```

Assembly Generated:

```
// calculate lerp value based on Value > 0
mov r1.w, c2.x
slt r0.w, c3.x, r1.w
// lerp between Value1 and Value2
mov r7, -c1
add r2, r7, c0
mad oPos, r0.w, r2, c1
```

The most common branching support in current hardware shading models is *static branching*. Static branching is a capability in a shader model that allows for blocks of code to be switched on or off based on a Boolean shader constant. This is a very convenient method for enabling/disabling potentially expensive code paths based on the type of object currently being rendered. Between Draw calls, you can decide the various features you want to support with the current shader and then set the Boolean flags required to get that behavior. The best part about this method is that any instructions that are 'disabled' by the Boolean constant are completely skipped during execution. The disadvantage is that you can only change which if blocks are enabled/disabled at a low frequency (i.e. between draw calls). In contrast, using the execute-both-sides approach, it is possible to dynamically choose between the outputs of the two paths dynamically at a per-pixel or per-vertex level.

The most familiar branching support is *dynamic branching*. The dynamic branching support offered by some shader models is very similar to that offered by a standard CPU. The performance hit is the cost of the branch plus the cost of the instructions on the side of the branch taken. This execution cost is comparable to what most people are familiar with optimizing for in CPU-side code. The problem with this form of branching is that it is not available on most hardware and is currently only available for vertex shaders. Optimizing shaders that work with these models is very similar to optimizing code that runs on a CPU.

Importance of input type declarations

The *type* of an input to a shader is used differently than you might expect. The method in which data is loaded into the registers either from a vertex buffer into a vertex shader or from the vertex shader output to the pixel shader input registers is well-defined in the Direct3D spec. That is, shader input values are always expanded into a vector of four floats. This means that the datatype declaration is more of a hint than a specification of how the data is loaded into the shader. Taking advantage of this provides a couple of optimization opportunities.

A common optimization used by shader assembly writers is to take advantage of the way in which data is expanded when loaded into registers. For example, in vertex shaders, the w component will be set to 1.0 if no w component is present in the vertex buffer. The y and z components will be set to 0.0 if not present in the vertex buffer. The most common place for this to be useful is for the position in vertex shaders. It is very common to need the w component to be set to 1.0 when multiplying by the World matrix, but the vertex buffer typically only contains x, y and z components. If the position input parameter is declared as a `float3`, then an extra instruction to copy a 1.0 into the w component would be required. If the parameter were declared as a `float4`, then the w component would be set to 1.0f by the hardware loading the input registers. The compiler cannot do this type of optimization automatically since this optimization requires knowledge of what data is in the vertex buffer.

Another optimization is to make sure and declare all input parameters with the appropriate type for their usage in the shader. For example, if the incoming data is integer and the data is going to be used for addressing purposes, then it is important to

declare the parameter as an `int` to avoid truncation. The subtle issue with declaring inputs as `int`s is that the values in the input should truly be integer values. Otherwise, the generated code might not run correctly due to the optimizations the compiler will make based upon the assumption that the input data is truly integer data.

Acknowledgements

Thanks to ATI's 3D Application Research Group for providing the sample HLSL shaders. Thanks to Craig Peeper, Dan Baker and Loren McQuade of Microsoft for their feedback and specifically their contributions to the section on optimizations.