Introduction to the DirectX® 9 High Level Shading Language

Craig Peeper
CraigP@microsoft.com
Development Lead
Microsoft Corporation

Jason L. Mitchell
JasonM@ati.com
3D Application Research Group Lead
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 details such as register allocation, register read-port limits, instruction co-issuing and so on. In addition to freeing the developer from hardware details, the HLSL also has all of the usual advantages of a high level language such as easy code reuse, improved readability and the presence of an optimizing compiler. Many of the chapters in this book and in the *ShaderX^2 - Shader Tips & Tricks* book will utilize shaders which are written in HLSL. As a result, it will be much easier for you to understand and work with those shaders after reading this introductory chapter.

In this chapter, we will outline the basic structure of the language itself as well as strategies for integrating HLSL shaders into your application.

A Simple Example

Before presenting an exhaustive description of the HLSL, let's first 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.

Assembly Language and Compile Targets

Now that we have seen a few HLSL shaders, we'll discuss briefly how the language relates to Direct3D, D3DX, assembly shader models and your application. Shaders were first added to Direct3D in DirectX 8. At that time, several virtual shader machines were defined—each roughly corresponding to a particular graphics processor produced by each of the top 3D graphics hardware vendors. For each of these virtual shader machines, an assembly language was designed. 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()`. For more on the details of the legacy assembly shader models, please refer to the many resources available online and offline, including *Shader X* and the DirectX SDK.

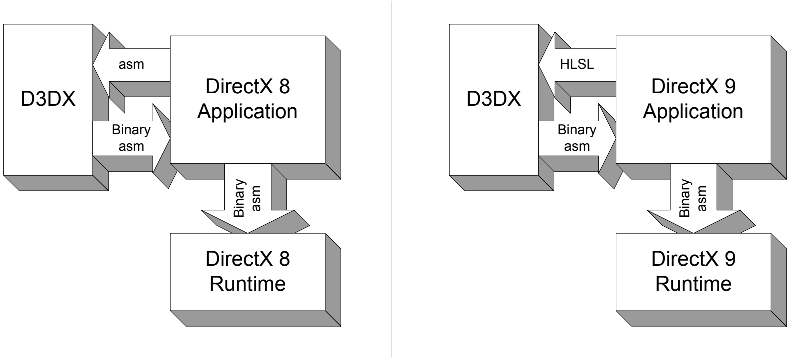

Figure 1 – Use of D3DX for Assembly and Compilation in DirectX 8 and DirectX 9

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, between press time and the release of the first printing of this book in late summer 2003, Microsoft plans to release a DirectX SDK Update which will contain an updated HLSL compiler.

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 game 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.

The Commandline Compiler - fxc

Rather than compile HLSL shaders using D3DX on the customer’s machine at application load time or at first use, many developers choose to compile their shaders from HLSL to binary asm before they even ship. This keeps their HLSL source away from prying eyes and also ensures that all of the shaders that their app will ever run have gone through their internal quality assurance process. A convenient utility which allows developers to compile shaders offline is the fxc commandline compiler which is provided in the DirectX 9.0 SDK. This utility has a number of convenient options that you can use to not only compile your shaders on the commandline but also generate disassembled code for the specified compile target. Studying the disassembled output can be very educational during development if you want to optimize your shaders or just generally get to know the virtual shader machine’s capabilities at a more detailed level. These commandline options are summarized in Table 1.

Option	**Description**
-T *target*	compile target (default: vs_2_0)
-E *name*	entrypoint *name* (default: main)
-Od	disable optimizations
-Vd	disable validation
-Zi	enable debugging information
-Zpr	pack matrices in row-major order
-Zpc	pack matrices in column-major order
-Fo *file*	output object file
-Fc *file*	output listing of generated code
-Fh *file*	output header containing generated code
-D *id* = *text*	define macro
-nologo	suppress copyright message

Table 1 – fxc Commandline Options.

Now that you understand the context in which the HLSL compiler can be used for shader development, we will discuss the actual mechanics of the language. As we progress, it is important to keep the notion of a *compile target* and the varying capabilities of the underlying assembly shader models in mind.

Language basics

Now that you have a sense of what HLSL vertex and pixel shaders look like and how they interact with the low-level assembly shaders, we'll discuss some of the details of the language itself.

Keywords

Keywords are predefined identifiers that are reserved for the HLSL language and cannot be used as identifiers in your program. Keywords marked with '*' are case insensitive.

```
asm *         bool            compile          const
decl*         do              double           else
extern        false           float            for
half          if              in               inline
inout         int             matrix *         out
pass *        pixelshader *   return           sampler
shared        static          string *         struct
technique *   texture *       true             typedef
uniform       vector *        vertexshader *   void
volatile      while
```

The following keywords are currently unused, but are reserved for potential future use:

```
auto                break        compile        const
char                class        case           catch
default             delete       const_cast     continue
explicit            friend       dynamic_cast   enum
mutable             namespace    goto           long
private             protected    new            operator
reinterpret_cast    short        public         register
static_cast         switch       signed         sizeof
throw               try          template       this
typename            unsigned     using          union
virtual
```

Datatypes

The HLSL has support for a variety of data types, from simple scalars to more complex types such as vectors and matrices.

Scalar Types

The language supports the following scalar datatypes:

Datatype	**Representable Values**
`bool`	true or false
`int`	32-bit signed integer
`half`	16-bit floating point value
`float`	32-bit floating point value
`double`	64-bit floating point value

If you are already familiar with the assembly-level programming models, you will know that graphics processors do not currently have native support for all of these datatypes. As a result, integers may need to be emulated using floating point hardware. This means that integer operations that go outside the range of integers that can be expressed as floats on these platforms are not guaranteed to function as expected. Additionally, not all target platforms have native support for half or double values. If the target platform does not, these will be emulated using float.

Vector Types

You will often find yourself declaring vector variables in your HLSL shaders. There are a variety of ways that these vectors can be declared, including the following:

`vector`	A vector of dimension 4; each component is of type float.
`vector` < *type*, *size* >	A vector of dimension *size*; each component is of scalar type *type*.

Vector Types

The most common way that you will see shader authors declare vectors, however, is by using the name of a type followed by an integer from 2 to 4. To declare a 4-tuple of floats, for example, you could use any of the following vector declarations:

```
float4 fVector0;
float  fVector1[4];
```

```
vector fVector2;
vector <float, 4> fVector3;
```

To declare a 3-tuple of `bool`s, for example, you could use any of the following declarations:

```
bool3 bVector0;
bool  bVector1[3];
vector <bool, 3> bVector2;
```

Once you have defined a vector, you may access its individual components by using the array access syntax or using a swizzle. In the swizzle case, the components must come from either the {*x*. *y*, *z*, *w*} or {*r*, *g*, *b*, *a*} name-space (but not both). For example:

```
float4 pos = {3.0f, 5.0f, 2.0f, 1.0f};
float  value0 = pos[0]; // value0 is 3.0f
float  value1 = pos.x;  // value1 is 3.0f
float  value2 = pos.g;  // value2 is 5.0f
float2 vec0   = pos.xy; // vec0 is {3.0f, 5.0f}
float2 vec1   = pos.ry; // INVALID because of bad swizzle
```

It should be noted that the ps_2_0 and lower pixel shader models do not have native support for arbitrary swizzles. Hence, concise high level code which uses swizzles can result in fairly nasty binary asm when compiling to these targets. You should familiarize yourself with the native swizzles available in these assembly models.

Matrix Types

Another very common type of variable you will find yourself using in HLSL shaders is matrices, which are 2D arrays of data. Like scalars and vectors, matrices may be composed of any of the basic datatypes: bool, int, half, float or double. Matrices may be of any size, but you will typically find shader writers using matrices with up to 4 rows and columns. You will recall that the example vertex shader shown at the beginning of the chapter declared two 4×4 float matrices at global scope:

```
float4x4 view_proj_matrix;
float4x4 texture_matrix0;
```

Naturally, other dimensions of matrices can be used. For example, we could declare a floating-point matrix with 3 rows and 4 columns in a variety of ways:

```
float3x4            mat0;
matrix<float, 3, 4> mat1;
```

Like vectors, the individual elements of matrices can be accessed using array or structure/swizzle syntax. For example, the following array indexing syntax can be used to access the top-left element of the matrix `view_proj_matrix`:

```
float fValue = view_proj_matrix[0][0];
```

There is also a structure syntax defined for access to and swizzling of matrix elements. For zero-based row-column position, you can use any of the following:

_m00, _m01, _m02, _m03
_m10, _m11, _m12, _m13
_m20, _m21, _m22, _m23
_m30, _m31, _m32, _m33

For one-based row-column position, you can use any of the following:

_11, _12, _13, _14
_21, _22, _23, _24
_31, _32, _33, _34
_41, _42, _43, _44

Matrices may also be accessed using array notation: For example:

```
float2x2 fMat = {3.0f, 5.0f,  // row 1
                 2.0f, 1.0f}; // row 2

float  value0 = fMat[0];      // value0 is 3.0f
float  value1 = fMat._m00;    // value1 is 3.0f
float  value2 = fMat._12      // value2 is 5.0f
float  value3 = fMat[1][1]    // value3 is 1.0f
float2 vec0   = fMat._21_22;  // vec0 is {2.0f, 1.0f}
float2 vec1   = fMat[1];      // vec1 is {2.0f, 1.0f}
```

Type Modifiers

There are a couple of optional type modifiers in the HLSL which you may want to use in your shaders. The familiar `const` type modifier is used to specify a variable whose value cannot be changed by the shader code. Using such a variable on the left sign of an assignment (i.e. as an *lval*) will result in a compilation error.

The `row_major` and `col_major` type modifiers can be used to specify the expected layout of a matrix within the hardware constant store. The `row_major` type modifier indicates that each row of the matrix will be stored in a single constant register. Likewise, using `col_major` indicates that each column of the matrix will be stored in a single constant register. Column-major is the default.

Storage Class Modifiers

Storage class modifiers inform the compiler about the intended scope and lifetime of a given variable. These modifiers are optional and may appear in any order as long as they appear before the variable type.

Like in C, a variable may be declared as `static` or `extern`. (These two modifiers are mutually exclusive.) At global scope, the `static` storage class modifier indicates that the variable is only to be accessed by the shader and not by the application via the API. Any non-static variable which is declared at global scope may be modified by the application through the API. Like in C, using the `static` modifier at *local* scope indicates that the variable contains data that is to persist between invocations of the declaring function.

The `extern` modifier can be used on a global variable to indicate that it can be modified from outside of the shader via the API. This is redundant, however, as this is the default behavior for variables declared at global scope.

The `shared` modifier is used to specify that a given global variable is to be shared between effects.

A variable which is `uniform` is assumed to have been set externally to the HLSL shader (i.e. via the `Set*ShaderConstant*()` API). Global variables are treated as if they were declared `uniform`. Such variables are not assumed to be `const`, however, as their values can be modified in the shader.

For example, say you declare the following variables at global scope:

```
extern float translucencyCoeff;
const  float gloss_bias;
static float gloss_scale;
float diffuse;
```

The variables `diffuse` and `translucencyCoeff` are settable by the `Set*ShaderConstant*()` API and can be modified by the shader itself. The `const` variable `gloss_bias` is settable by the `Set*ShaderConstant*()` API but cannot be modified in the shader code. Finally, the `static` variable `gloss_scale` is not settable by the `Set*ShaderConstant*()` API but can be modified within the shader only.

Initializers

As we have shown in some of the preceding examples, it is possible to initialize variables at declaration time in the same manner used in C. For example:

```
float2x2 fMat = {3.0f, 5.0f,  // row 1
                 2.0f, 1.0f}; // row 2
float4   vPos = {3.0f, 5.0f, 2.0f, 1.0f};
float fFactor = 0.2f;
```

Working with Vectors

In HLSL, there are a few gotchas to look out for when performing math on vectors. Fortunately, most of them are quite intuitive given that we are writing shaders for 3D graphics. For example, standard binary operators are defined to work *per component*:

```
float4 vTone = vBrightness * vExposure;
```

Assuming vBrightness and vExposure are both of type float4, this is equivalent to:

```
float4 vTone;
vTone.x = vBrightness.x * vExposure.x;
vTone.y = vBrightness.y * vExposure.y;
vTone.z = vBrightness.z * vExposure.z;
vTone.w = vBrightness.w * vExposure.w;
```

Note that this is *not* a dot product between the 4D vectors vBrightness and vExposure. Additionally, multiplying matrix variables in this way does not result in a matrix multiply. Dot products and matrix multiplies are applied via the intrinsic function mul() which we will discuss later in the chapter.

Constructors

Another language feature that you will often see in HLSL shaders is the constructor, which is similar to C++ but has some enhancements to deal with complex datatypes. Example uses of constructors include:

```
float3   vPos     = float3(4.0f, 1.0f, 2.0f);
float    fDiffuse = dot(vNormal, float3(1.0f, 0.0f, 0.0f));
float4   vPack    = float4(vPos, fDiffuse);
```

Constructors are commonly used when a shader writer wants to temporarily define a quantity with literal values (as in dot(vNormal, float3(1.0f, 0.0f, 0.0f)) above) or when a shader writer wants to explicitly pack smaller datatypes together (as in float4(vPos, fDiffuse) above). In this case, the float4 constructor takes in a float3 and a float and returns a float4 with the data packed together.

Type Casting

To aid in shader writing and in the efficiency of the generated code, it is a good idea to be familiar with HLSL's type casting behavior. Type casting often happens in order to promote or demote a given variable to match a variable to which it is being assigned. For example, in the following case, a literal float 0.0f is being cast to a float4 {0.0f , 0.0f , 0.0f , 0.0f } to initialize vResult.

```
float4   vResult = 0.0f;
```

Similar casting can occur when assigning a higher dimensional datatype like a vector or matrix to a lower dimensional datatype. In these cases, the extra data is effectively omitted. For example, we may write the following code:

```
float3   vLight;
float    fFinal, fColor;
fFinal = vLight * fColor;
```

In this case, `vLight` is cast to a float by using only the first component in the multiply with the scalar float `fColor`. In this case, `fFinal` is equal to `vLight.x * fColor`.

It is a good idea to be familiar with the following table of type casting rules:

Type of cast	**Casting Behavior**
Scalar-to-scalar	**Always valid**. When casting from bool type to an integer or floating point type, false is considered to be zero, and true is considered to be one. When casting from an integer or floating point type to bool, a zero value is considered to be false, and a nonzero value is considered to be true. When casting from a floating point type to an integer type, the value is rounded toward zero. This is the same truncation behavior as C.
Scalar-to-vector	**Always valid**. This cast operates by replicating the scalar to fill the vector.
Scalar-to-matrix	**Always valid**. This cast operates by replicating the scalar to fill the matrix.
Scalar-to-structure	This cast operates by replicating the scalar to fill the structure.
Vector-to-scalar	**Always valid**. This selects the first component of the vector
Vector-to-vector	The destination vector must not be larger than the source vector. The cast operates by keeping the left-most values, and truncating the rest. For the purposes of this cast, column matrices, row matrices, and numeric structures are treated as vectors.
Vector-to-matrix	The size of the vector must be equal to the size of the matrix.
Vector-to-structure	Valid if the structure is not larger than the vector, and all components of the structure are numeric.
Matrix-to-scalar	**Always valid**. This selects the upper-left component of the matrix.
Matrix-to-vector	The size of the matrix must be equal to the size of the vector.
Matrix-to-matrix	The destination matrix must not be larger than the source matrix, in both dimensions. The cast operates by keeping the upper-left values, and truncating the rest.
Matrix-to-structure	The size of the structure must be equal to the size of the matrix, and all components of the structure are numeric.
Structure-to-scalar	The structure must contain at least one member.
Structure-to-vector	The structure must be at least the size of the vector. The first

	components must be numeric, up to the size of the vector.
Structure-to-matrix	The structure must be at least the size of the matrix. The first components must be numeric, up to the size of the matrix.
Structure-to-object	The structure must contain at least one member. The type of this member must be identical to the type of the object.
Structure-to-structure	The destination structure must not be larger than the source structure. A valid cast must exist between all respective source and destination components.

Structures

As we showed in the first example shader above, it is often convenient to be able to define structures in HLSL shaders. For example, many shader writers will define an output structure in their vertex shader code and use this structure as the return type from their vertex shader's main function. (It is less common to do this with a pixel shader since most pixel shaders have only one `float4` output.) An example structure taken from the NPR Metallic shader that we will discuss later is shown below:

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

Structures may be declared for general use in an HLSL shader as well. They follow the type-casting rules outlined above.

Samplers

For each different texture map that you plan to sample in a pixel shader, you must declare a *sampler*. Recall the `hlsl_rings()` shader described earlier:

```
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);
}
```

In this shader, we declared a sampler called `PulseTrainSampler` at global scope and passed it as the first parameter to the `tex1D()` intrinsic function (we will discuss intrinsics in the next section). An HLSL sampler has a very direct mapping to the API concept of a sampler and, in turn, to the actual silicon in the 3D graphics processor which is responsible for addressing and filtering textures. A sampler must be defined for every texture map that you plan to access in a given shader, but you may use a given sampler multiple times in a shader. This usage is very common in image processing applications as discussed in the *ShaderX^2 - Shader Tips & Tricks* chapter "Advanced Image Processing with DirectX 9 Pixel Shaders" since the input image is often sampled multiple times with different texture coordinates to provide data to a filter kernel expressed in shader code. For example, the following shader uses the rasterizer to convert a height map to a normal map with a pair of Sobel filters.

```
sampler InputImage;

float4 main( float2 topLeft    : TEXCOORD0, float2 left        : TEXCOORD1,
             float2 bottomLeft : TEXCOORD2, float2 top         : TEXCOORD3,
             float2 bottom     : TEXCOORD4, float2 topRight    : TEXCOORD5,
             float2 right      : TEXCOORD6, float2 bottomRight : TEXCOORD7): COLOR
{
   // Take all eight taps
   float4 tl = tex2D (InputImage, topLeft);
   float4  l = tex2D (InputImage, left);
   float4 bl = tex2D (InputImage, bottomLeft);
   float4  t = tex2D (InputImage, top);
   float4  b = tex2D (InputImage, bottom);
   float4 tr = tex2D (InputImage, topRight);
   float4  r = tex2D (InputImage, right);
   float4 br = tex2D (InputImage, bottomRight);

   // Compute dx using Sobel operator:
   //
   //            -1 0 1
   //            -2 0 2
   //            -1 0 1
   float dX = -tl.a - 2.0f*l.a - bl.a + tr.a + 2.0f*r.a + br.a;

   // Compute dy using Sobel operator:
   //
   //            -1 -2 -1
   //             0  0  0
   //             1  2  1
   float dY = -tl.a - 2.0f*t.a - tr.a + bl.a + 2.0f*b.a + br.a;

   // Compute cross-product and renormalize
   float4 N = float4(normalize(float3(-dX, -dY, 1)), tl.a);

   // Convert signed values from -1..1 to 0..1 range and return
   return N * 0.5f + 0.5f;
}
```

This shader uses only one sampler, `InputImage`, but samples from it eight times using the `tex2D()` intrinsic function.

Intrinsics

As mentioned in the preceding section, there are a number of *intrinsics* built into the DirectX 9 High Level Shading Language for your convenience. Many intrinsics, such as mathematical functions, are provided for convenience, while others, such as the `tex1D()` and `tex2D()` functions mentioned above, are necessary for accessing texture data via samplers.

Math Intrinsics

The math intrinsics listed in the table below will be converted to micro operations by the HLSL compiler. In some cases, such as `abs()` and `dot()`, these intrinsics will map directly to single assembly-level operations while in other cases, such as `refract()` and `step()`, they will map to multiple assembly instructions. There are even a couple of cases, notably `ddx()`, `ddy()` and `fwidth()`, which are not supported for all compile targets. The math intrinsics are shown below:

Intrinsic	**Description**
abs(*x*)	Absolute value (per component).
acos(*x*)	Returns the arccosine of each component of *x*. Each component should be in the range [-1, 1].
all(*x*)	Test if all components of *x* are nonzero.
any(*x*)	Test is any component of *x* is nonzero.
asin(*x*)	Returns the arcsine of each component of *x*. Each component should be in the range [-π/2, π /2].
atan(x)	Returns the arctangent of *x*. The return values are in the range [-π /2, π /2].
atan2(*y*, *x*)	Returns the arctangent of *y*/*x*. The signs of *y* and *x* are used to determine the quadrant of the return values in the range [-π, π]. atan2 is well-defined for every point other than the origin, even if *x* equals 0 and *y* does not equal 0.
ceil(*x*)	Returns the smallest integer which is greater than or equal to *x*.
clamp(*x*, *min*, *max*)	Clamps *x* to the range [*min*, *max*].
clip(*x*)	Discards the current pixel, if any component of *x* is less than zero. This can be used to simulate clip planes, if each component of *x* represents the distance from a plane. This is the intrinsic you use when you want to generate an asm `texkill`
cos(*x*)	Returns the cosine of *x*
cosh(*x*)	Returns the hyperbolic cosine of *x*
cross(*a*, *b*)	Returns the cross product of two 3D vectors *a* and *b*.
D3DCOLORtoUBYTE4(*x*)	Swizzles and scales components of the 4D vector *x* to compensate for the lack of UBYTE4 stream component support in some hardware.
ddx(*x*)	Returns the partial derivative of *x* with respect to the screen-space *x*-coordinate.
ddy(*x*)	Returns the partial derivative of *x* with respect to the screen-space *y*-coordinate
degrees(*x*)	Converts *x* from radians to degrees
determinant(*m*)	Returns the determinant of the square matrix *m*

distance(*a*, *b*)	Returns the distance between two points *a* and *b*
dot(*a*, *b*)	Returns the dot product of two vectors *a* and *b*
exp(*x*)	Returns the base-e exponent e^x
exp2(a)	Base 2 Exp (per component)
faceforward(*n*, *i*, *ng*)	Returns -*n* * sign(dot(*i*, *ng*))
floor(*x*)	Returns the greatest integer which is less than or equal to *x*
fmod(*a*, *b*)	Returns the floating point remainder *f* of *a* / *b* such that *a* = *i* * *b* + *f*, where *i* is an integer, *f* has the same sign as *x*, and the absolute value of *f* is less than the absolute value of *b*.
frac(*x*)	Returns the fractional part *f* of *x*, such that *f* is a value greater than or equal to 0, and less than 1.
frexp(*x*, out *exp*)	Returns the mantissa and exponent of *x*. frexp returns the mantissa, and the exponent is stored in the output parameter *exp*. If *x* is 0, the function returns 0 for both the mantissa and the exponent.
fwidth(*x*)	Returns abs(ddx(*x*))+abs(ddy(*x*)).
isfinite(*x*)	Returns true if x is finite, false otherwise.
isinf(*x*)	Returns true if *x* is +INF or -INF, false otherwise
isnan(*x*)	Returns true if *x* is NAN or QNAN, false otherwise
ldexp(*x*, *exp*)	Returns *x* * 2^{exp}
len(*v*)	Vector length
length(*v*)	Returns the length of the vector *v*
lerp(*a*, *b*, *s*)	Returns *a* + *s*(*b* - *a*). This linearly interpolates between *a* and *b*, such that the return value is *a* when *s* is 0, and *b* when *s* is 1.
log(*x*)	Returns the base-e logarithm of *x*. If *x* is negative, the function returns indefinite. If *x* is 0, the function returns +INF.
log10(*x*)	Returns the base-10 logarithm of *x*. If *x* is negative, the function returns indefinite. If *x* is 0, the function returns +INF.
log2(*x*)	Returns the base-2 logarithm of *x*. If *x* is negative, the function returns indefinite. If *x* is 0, the function returns +INF.
max(*a*, *b*)	Selects the greater of *a* and *b*
min(*a*, *b*)	Selects the lesser of *a* and *b*
modf(*x*, out *ip*)	Splits the value *x* into fractional and integer parts, each of which has the same sign as *x*. The signed fractional portion of *x* is returned. The integer portion is stored in the output parameter *ip*.
mul(a, b)	Performs matrix multiplication between *a* and *b*. If *a* is a vector, it is treated as a row vector. If *b* is a vector, it is treated as a column vector. The inner dimension $a_{columns}$ and b_{rows} must be equal. The result has the dimension $a_{rows} \times b_{columns}$.
normalize(*v*)	Returns the normalized vector *v* / length(*v*). If the length of *v* is 0, the result is indefinite.
pow(*x*, *y*)	Returns x^y
radians(*x*)	Converts *x* from degrees to radians
reflect(*i*, *n*)	Returns the reflection vector *v*, given the entering ray direction *i*, and the surface normal *n*. Such that *v* = i - 2 * dot(*i*, *n*) * *n*
refract(*i*, *n*, *eta*)	Returns the refraction vector *v*, given the entering ray direction *i*, the surface

	normal n, and the relative index of refraction *eta*. If the angle between i and n is too great for a given *eta*, refract returns (0,0,0).
round(x)	Rounds x to the nearest integer.
rsqrt(x)	Returns 1 / sqrt(x).
saturate(x)	Clamps x to the range [0, 1].
sign(x)	Computes the sign of x. Returns -1 if x is less than 0, 0 if x equals 0, and 1 if x is greater than zero.
sin(x)	Returns the sine of x
sincos(x, out s, out c)	Returns the sine and cosine of x. sin(x) is stored in the output parameter s. cos(x) is stored in the output parameter c
sinh(x)	Returns the hyperbolic sine of x
smoothstep(*min*, *max*, x)	Returns 0 if $x < min$. Returns 1 if $x > max$. Returns a smooth Hermite interpolation between 0 and 1, if x is in the range [*min*, *max*].
sqrt(x)	Square root (per component).
step(a, x)	Returns ($x \geq a$) ? 1 : 0.
tan(x)	Returns the tangent of x
tanh(x)	Returns the hyperbolic tangent of x
transpose(m)	Returns the transpose of the matrix m. If the source is dimension $m_{rows} \times m_{columns}$, the result is dimension $m_{columns} \times m_{rows}$

Texture Sampling Intrinsics

There are sixteen texture sampling intrinsics used for sampling texture data into a shader. There are four types of textures (1D, 2D, 3D and cube map) and four types of loads (regular, with derivatives, projective and biased) with an intrinsic for each of the sixteen combinations:

Intrinsic	Description
tex1D(s, t)	**1D texture lookup**. s is a sampler. t is a scalar.
tex1D(s, t, *ddx*, *ddy*)	**1D texture lookup, with derivatives**. s is a sampler. t, *ddx*, and *ddy* are scalars.
tex1Dproj(s, t)	**1D projective texture lookup**. s is a sampler. t is a 4D vector. t is divided by its last component before the lookup takes place.
tex1Dbias(s, t)	**1D biased texture lookup**. s is a sampler. t is a 4D vector. The mip level is biased by t.w before the lookup takes place.
tex2D(s, t)	**2D texture lookup**. s is a sampler. t is a 2D texture coordinate.
tex2D(s, t, *ddx*, *ddy*)	**2D texture lookup, with derivatives**. s is a sampler. t, *ddx*, and *ddy* are 2D vectors.
tex2Dproj(s, t)	**2D projective texture lookup**. s is a sampler. t is a 4D vector. t is divided by its last component before the lookup takes place.
tex2Dbias(s, t)	**2D biased texture lookup**. s is a sampler. t is a 4D vector. The mip level is biased by t.w before the lookup takes place.
tex3D(s, t)	**3D volume texture lookup**. s is a sampler. t is a 3D texture coordinate.
tex3D(s, t, *ddx*, *ddy*)	**3D volume texture lookup, with derivatives**. s is a sampler. t, *ddx*, and *ddy* are

	3D vectors.
tex3Dproj(*s*, *t*)	**3D projective volume texture lookup**. *s* is a sampler. *t* is a 4D vector. *t* is divided by its last component before the lookup takes place.
tex3Dbias(*s*, *t*)	**3D biased texture lookup**. *s* is a sampler. *t* is a 4D vector. The mip level is biased by *t*.w before the lookup takes place.
texCUBE(*s*, *t*)	**Cubemap lookup**. *s* is a sampler. *t* is a 3D texture coordinate.
texCUBE(*s*, *t*, *ddx*, *ddy*)	**Cubemap lookup, with derivatives**. *s* is a sampler. *t*, *ddx*, and *ddy* are 3D vectors.
texCUBEproj(*s*, *t*)	**Projective cubemap lookup**. *s* is a sampler. *t* is a 4D vector. *t* is divided by its last component before the lookup takes place.
texCUBEbias(*s*, *t*)	**Biased cubemap lookup**. *s* is a sampler. *t* is a 4D vector. The mip level is biased by *t*.w before the lookup takes place.

The `tex1D()`, `tex2D()`, `tex3D()` and `texCUBE()` intrinsics are the most commonly used to sample textures. The texture loading intrinsics which take *ddx* and *ddy* parameters compute texture LOD using these explicit derivatives, which would typically have been previously calculated with the `ddx()` and `ddy()` math intrinsics. These are particularly important when writing procedural pixel shaders, but are not supported on ps_2_0 or lower compile targets.

The `tex*proj()` intrinsics are used to do projective texture reads, where the texture coordinates used to sample the texture are divided by the last component prior to accessing the texture. Of these, `tex2Dproj()` is the most commonly used, since it is necessary for projective shadow maps and similar effects.

The `tex*bias()` intrinsics are used to perform biased texture sampling, where the bias can be computed per-pixel. This is typically done to induce some over-blurring of the texture for a special effect. For example, as discussed in the *ShaderX*^{2} *- Shader Tips & Tricks* chapter "Motion Blur Using Geometry and Shading Distortion," the pixel shader used on the motion-blurred balls in the RADEON™ 9700 Animusic *Pipe Dream* demo uses the `texCUBEbias()` intrinsic to access the cubic environment map of the local scene:

```
…
   // Blur reflection by extension amount.
   float3 vCubeLookup = vReflection + i.Pos/fEnvMapRadius;
   float4 cReflection = texCUBEbias(tCubeEnv, float4(vCubeLookup, fBlur * fTextureBlur))
* vReflectionColor;
…
```

In this code snippet, `fBlur * fTextureBlur` is stored in the fourth component of the texture coordinate used in the `texCUBEbias()` call and determines the bias to be used when accessing the cube map.

Now that we have introduced some of the mechanics of the language, we will discuss how data is input to and output from HLSL shaders in DirectX 9.

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;
}
```

The uniform variables used by a shader are communicated back to the application via the *constant table*. The constant table a symbol table which defines how the uniform variables used by a shader must be loaded into the constant registers prior to shader execution. (NOTE: the uniform input function parameters appear in the constant table with a '$' pre-pended unlike the global variables. The '$' is required to avoid name collisions between "local" uniform inputs and global variables of the same name.)

The constant table contains the constant register locations of all uniform variables used by the shader. The table also includes the type information and the default value, if specified, for each constant table entry. The following is an example of what a constant table looks like when printed out. The constant table generated by the compiler is stored in a compact binary form. The API to interpret the table at runtime will be discussed later in the section on HLSL integration without the use of D3DX Effects.

Textual printout of constant table emitted by fxc.exe for a sample shader:

```
//
// Generated by Microsoft (R) D3DX9 Shader Compiler
//
//  Source: hemisphere.fx
//  Flags: /E:VS /T:vs_1_1
//

// Registers:
//
//     Name         Reg   Size
//     ------------ ----- ----
//     Projection   c0       4
//     WorldView    c4       3
//     DirFromLight c7       1
//     DirFromSky   c8       1
//     $bHemi       c18      1
//     $bDiff       c19      1
//     $bSpec       c20      1
//
//
// Default values:
//
//     DirFromLight
//       c7   = { 0.577, -0.577, 0.577, 0 };
//
//     DirFromSky
//       c8   = { 0, -1, 0, 0 };
```

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. You will see both of these styles used in the example shaders in this chapter and throughout the ShaderX books.

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;
}
```

Vertex Shader Input Semantics

Semantic	Description
POSITION*n*	Position
BLENDWEIGHT*n*	Blend weights
BLENDINDICES*n*	Blend indices
NORMAL*n*	Normal vector
PSIZE*n*	Point size
COLOR*n*	Color
TEXCOORD*n*	Texture coordinates
TANGENT*n*	Tangent
BINORMAL*n*	Binormal
TESSFACTOR*n*	Tessellation factor

Pixel Shader Input Semantics

Semantic	Description
COLOR*n*	Color
TEXCOORD*n*	Texture coordinates

n is an optional integer. As an example, PSIZE0, DIFFUSE1, etc.

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.

Vertex Shader Output Semantics

Semantic	Description
POSITION	Position
PSIZE	Point size
FOG	Vertex fog
COLOR*n*	Color (example: COLOR0)
TEXCOORD*n*	Texture coordinates (example: TEXCOORD0)

Pixel Shader Output Semantics

Semantic	Description
COLOR*n*	Color for render target *n*
DEPTH	Depth value

n is an optional, integer. As an example, TEXCOORD3, COLOR0

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). This effect ships with the RenderMonkey shader development environment discussed in the chapter “Shader Programming with RenderMonkey” and is available on the ATI Developer Relations website (www.ati.com/developer).

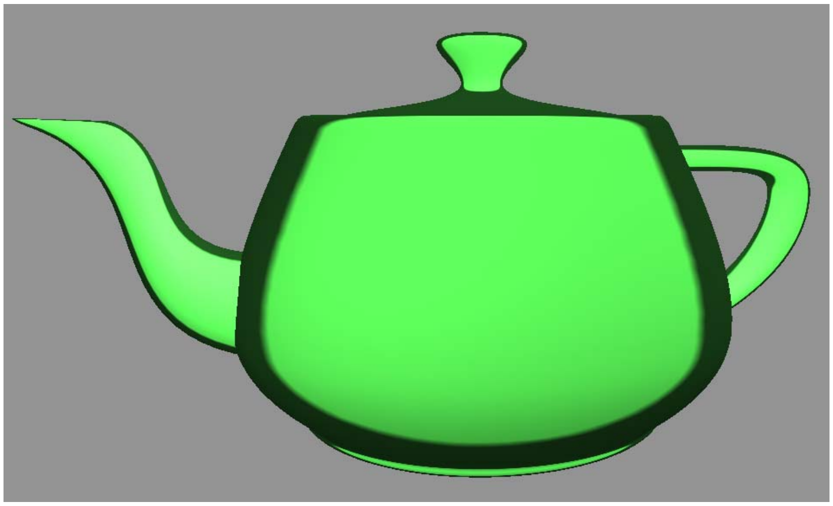

Figure 2 – NPR Metallic

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×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            // Index declared as int
frc r0.w, r1.w                        mul r0.w, c60.x, r1.w
add r2.w, -r0.w, r1.w                 mova a0.x, r0.w
mul r9.w, r2.w, c61.x                 m4x4 oPos, v0, c0[a0.x]
mova a0.x, r9.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.

Precision issues (`logp`, `expp`, `lit`)

A good understanding of precision is necessary for writing shaders that give correct results and reasonable performance. With most shader compile targets, the internal precision is fixed and needs to be taken into account for correct results. For example, the ps_1_*x* models have relatively low precision fixed-point registers. Raising a number to even a low power for specular highlights can quickly generate banding.

In some other models, such as vs_1_1 and vs_2_0, there are low-precision versions of some instructions. Specifically `logp`, `expp`, and `lit` can be used as low-precision versions of `log`, `exp`, and `pow`. On some hardware, the performance difference between the low and high precision variants is not significant. Since `log` and `exp` count for 10 instruction slots each and `logp` and `expp` only count as one instruction each, it is possible to balloon the size of the generated asm code and potentially run out of instructions, particularly on the vs_1_1 compile target. Accessing these low-precision instructions is accomplished by declaring the output to be either cast to or stored into the low precision datatype called 'half'. A low precision output from an operation informs the compiler that the operation should be performed with the lowest precision possible. Some pixel shader hardware can take advantage of performing other operations at a lower precision as well.

```
float LogValue = log(Value);         float LogValue = (half)log(Value);

// counts as 10 instructions         // counts as 1 instruction on
//                 on vs_1_1         //                     vs_1_1
log r0, c0                           logp r0, c0
```

Example of log vs logp

Using the ps_1_*x* Compile Targets

The original pixel shader models (ps_1_1, ps_1_2, ps_1_3 and ps_1_4) offer a large degree of programmability, but have some restrictions in what can be done. It is possible to efficiently target the ps_1_*x* compile targets using HLSL, but it is imperative for the shader writer to understand the underlying set of limitations. This is important in order to write high performance shaders, but more importantly to even get your shader to compile. Instruction count is probably the first limitation that most people hit, but this is usually due to ignoring other limitations of the ps_1_*x* compile targets.

The first thing to remember about the ps_1_*x* compile targets is that the target hardware does not have arbitrary swizzles. This limitation means that the compiler must use extra instructions anytime a swizzle is required. The extra instructions generated can quickly cause programs to overrun the total instructions possible in these compile targets. The ps_1_1 through ps_1_3 targets do not support arbitrary write masks or replicate swizzles (i.e. `.r`, `.g`, `.b` or `.a`) and can cause the same situation. The ps_1_4 compile target does have support for replicate swizzles and arbitrary write masks. Even with these limitations it is quite easy to write interesting and complex shaders. This is just something to keep in mind when writing HLSL code targeted at the ps_1_*x* compile targets.

While the ps_1_*x* targets naturally have disadvantages relative to the newer pixel shader models, one advantage that they do have is the existence of "free" source and dest modifiers (i.e. the ability to clamp values to the 0 to 1 range, take the complement of a source, negate a source, bias a source, etc). These modifiers are extremely handy when generating shaders that accomplish a lot in a small number of instructions. The compiler automatically matches all modifiers that it can, but it is helpful if the HLSL shader writer thinks in terms of using these modifiers to accomplish certain operations. In fact, some intrinsics were added to HLSL to make this type of shader writing easier. For example, it is recommended that you use the `saturate()` intrinsic when trying to generate a free `_sat` modifier in a pixel shader. We will now present a series of HLSL code sequences which will generate free source modifiers when compiling to ps_1_*x* targets.

The `_bx2` Modifier

To cause the HLSL compiler to generate `_bx2` modifiers, there are a number of different HLSL code sequences that can be used. Any of the following main functions will cause the compiler to generate a `_bx2` modifier:

```
float4 main( float3 Col : COLOR0, float3 Tex : TEXCOORD0 ) : COLOR0
{
    return dot(Col, Tex*2 - 1);
}

float4 main( float3 Col : COLOR0, float3 Tex : TEXCOORD0) : COLOR0
{
    float3 val = Tex*2;
    val = val -1;
    return dot(Col,val);
}

float4 main( float3 Col : COLOR0, float3 Tex : TEXCOORD0 ) : COLOR0
{
    return dot(Col, (Tex -.5f)*2 );
}
```

All of these main functions will generate the same asm shader:

```
ps_1_1
texcoord t0
dp3 r0, v0, t0_bx2
```

It is important to note that the `Tex*2 -1` version is recommend because it generates more optimal code in ps_2_0 targets and beyond.

The `_bias` Modifier

The following code causes the HLSL compiler to generate a `_bias` modifier

```
float4 main( float3 Col : COLOR0, float3 Tex : TEXCOORD0 ) : COLOR0
{
    return dot(Col, (Tex - .5f));
}
```

This main function generates the following assembly shader:

```
ps_1_1
texcoord t0
dp3 r0, v0, t0_bias
```

Note that `_bias` cannot be done in ps_1_1, ps_1_2 or ps_1_3 unless the source is known to be in the range of 0 to 1. That is, it must have been previously saturated.

The `_x2` Modifier (ps_1_4 only)

The following code causes the HLSL compiler to generate a `_x2` source modifier:

```
float4 main( float3 Col : COLOR0, float3 Tex : TEXCOORD0 ) : COLOR0
{
    return dot(Col, Tex*2);
}
```

This HLSL code results in the following asm shader code:

```
ps_1_4
texcrd r0.xyz, t0
dp3 r0, v0, r0_x2
```

The `_x2`, `_x4`, `_x8`, `_d2`, `_d4` and `_d8` Destination Write Modifiers

A set of destination write modifiers existed in the ps_1_*x* models and it is possible to write HLSL code to cause the compiler to generate them in the resulting asm. The modifiers to double (`_x2`), quadruple (`_x4`), and halve (`_d2`) the result of the instruction are supported on ps_1_1 through ps_1_3 models while the ps_1_4 model supports all six of the `_x2`, `_x4`, `_x8`, `_d2`, `_d4` and `_d8` modifiers. The following code will generate the corresponding modifiers for N = 2, 4, 8, 0.5, 0.25 or 0.125:

```
static const float N = 2;
```

```
float4 main( float4 Col[2] : COLOR0) : COLOR0
{
    return (Col[0] + Col[1] )*N;
}
```

The above HLSL code will result in the following asm output:

```
ps_1_1
add_x2 r0, v0, v1
```

The Complement Modifier

It is also possible to write HLSL code which will allow the compiler to generate a complement modifier when compiling to a ps_1_*x* target. Note that this will only work if the quantity being complemented is known to be in the 0 to 1 range (i.e. the quantity has previously been saturated). The following HLSL code will cause the compiler to generate a free complement modifier:

```
float4 main( float4 Col[2] : COLOR0) : COLOR0
{
    return (1-Col[0]) * (Col[1]);
}
```

This HLSL code results in the following asm shader:

```
ps_1_1
mul r0, 1-v0, v1
```

The Saturate Modifier

The following two shaders will generate a `_sat` modifier. This modifier is available on all pixel shader compile targets:

```
float4 main( float4 Col[2] : COLOR0) : COLOR0
{
    return saturate(Col[0]);
}

float4 main( float4 Col[2] : COLOR0) : COLOR0
{
    return clamp(Col[0],0,1);
}
```

Both of these HLSL shaders will result in the following asm shader:

```
ps_1_1
mov_sat r0, v0
```

The Negate Modifier

The following shader will generate a negate modifier, which is also available on all shader targets. (Note that on ps_1_*x*, constant registers cannot be directly negated and hence will not result in a single free negation since the constant will have to be moved to a temp before it can be negated.)

```
float4 main( float4 Col[2] : COLOR0) : COLOR0
{
    return -Col[0];
}
```

This HLSL code will result in the following asm shader:

```
ps_1_1
mov r0, -v0
```

Strategy for Targeting ps_1_*x*

The best strategy that we have found to optimize for ps_1_*x* compile targets is to first write your shader on ps_2_0 since this allows for quick and easy prototyping on ps_2_0 capable hardware. Once the shader is working as desired, cross-compile it for the desired ps_1_*x* model. Using the disable validation option, -Vd for fxc.exe, you can see how many instructions the shader would be if there were no instruction limits on the chosen ps_1_*x* model. If the shader did not fit, you can at least see what you are up against and begin paring away the least necessary pieces of your shader to get an efficient ps_1_*x* fallback up and running.

Now that we have presented HLSL shaders in detail, we will discuss the issues involved in integrating HLSL shader support into an application. HLSL can be integrated into your engine with or without the use of D3DX Effects and we will now discuss both approaches. It is also worth mentioning that it is possible to start experimenting with HLSL without writing any application code by using a shader development environment such as RenderMonkey. For more on RenderMonkey, please consult the chapter “Shader Programming with RenderMonkey.”

Integration into an engine using D3DX Effects

The D3DX Effects framework is a very useful component of the D3DX library that is gaining more adoption among professional developers. Naturally, in DirectX 9, D3DX Effects were updated to include support for HLSL. If you aren't familiar with D3DX Effects, they are an abstraction designed to conveniently encapsulate rendering special effects in 3D applications. Effects can encapsulate rendering state as well as shaders written in asm or HLSL, including fall-back versions targeted at legacy hardware. A given *Effect* is generally stored in a single .fx or .fxl file and the file itself can contain multiple versions of the Effect called *Techniques*. For example, you may want to create a more basic version of a given Effect that you can use on older hardware with legacy shader support or no shaders at all. An excellent example of this kind of use of Techniques is the Water sample in the DirectX SDK. This sample uses several different Techniques which are targeted at different generations of hardware. Of course, the more basic Techniques which require less textures and generally less sophistication don't look as impressive, but that's the point: D3DX Effects let you manage this quality/speed trade-off very naturally.

Effect Files

We won't go into all of the facets of Effects here, but you should understand the basic structure of an Effect file in order to see how it can be used with HLSL. A typical effect file might look something like this:

```
// Lighting constants
VECTOR g_Leye;
float4 GlobalAmbient = 0.5;
float Ka = 1;
float Kd = 0.8;
float Ks = 0.9;
float roughness = 0.1;
float noiseFrequency;

MATRIX matWorldViewProj;
MATRIX matWorldView;
MATRIX matITWorldView;
MATRIX matWorld;
MATRIX matTex0;

TEXTURE tVolumeNoise;
TEXTURE tMarbleSpline;

sampler NoiseSampler = sampler_state
{
   Texture = (tVolumeNoise);

   MinFilter = Linear;
   MagFilter = Linear;
   MipFilter = Linear;
   AddressU  = Wrap;
   AddressV  = Wrap;
   AddressW  = Wrap;
   MaxAnisotropy = 16;
};
```

```
sampler MarbleSplineSampler = sampler_state
{
   Texture = (tMarbleSpline);

   MinFilter = Linear;
   MagFilter = Linear;
   MipFilter = Linear;
   AddressU  = Clamp;
   AddressV  = Clamp;
   MaxAnisotropy = 16;
};

float3 snoise (float3 x)
{
    return 2.0f * tex3D (NoiseSampler, x) - 1.0f;
}

float4 ambient(void)
{
   return GlobalAmbient;
}

float4 soft_diffuse(float3 Neye, float3 Peye)
{
   // Compute normalized vector from vertex to light in eye space  (Leye)
   float3 Leye = (g_Leye - Peye) / length(g_Leye - Peye);

   float NdotL = dot(Neye, Leye) * 0.5f + 0.5f;

   // N.L
   return float4(NdotL, NdotL, NdotL, NdotL);
}

float4 specular(float3 NNeye, float3 Peye, float k)
{
    // Compute normalized vector from vertex to light in eye space  (Leye)
    float3 Leye = (g_Leye - Peye) / length(g_Leye - Peye);

    // Compute Veye
    float3 Veye = -(Peye / length(Peye));

    // Compute half-angle
    float3 Heye = (Leye + Veye) / length(Leye + Veye);

    // Compute N.H
    float NdotH = clamp(dot(NNeye, Heye), 0.0f, 1.0f);

    float NdotH_2  = NdotH    * NdotH;
    float NdotH_4  = NdotH_2  * NdotH_2;
    float NdotH_8  = NdotH_4  * NdotH_4;
    float NdotH_16 = NdotH_8  * NdotH_8;
    float NdotH_32 = NdotH_16 * NdotH_16;

    return NdotH_32 * NdotH_32;
}

float4 hlsl_bluemarble (float3 P    : TEXCOORD0,
                        float3 Peye : TEXCOORD1,
                        float3 Neye : TEXCOORD2) : COLOR
{
   float4 Ct;
   float4 Ci;
   float3 NNeye;
   float marble;
   float f;

   // Divide down to nice frequency
   P = P/16;
   marble = -2.0f * snoise(P * noiseFrequency) + 0.75f;

   NNeye = normalize(Neye);

   // Cubic interpolation of f along color spline (gloss in alpha)
   Ct = tex1D (MarbleSplineSampler, marble);

   // Color from illumination
   Ci = Ct * (Ka * ambient() + Kd * soft_diffuse(NNeye, Peye)) +
        Ct.w * Ks * specular(NNeye, Peye, roughness);

   return Ci;
}

VERTEXSHADER asm_marble_vs =
decl {}
asm
{
   vs.1.1

   dcl_position v0
   dcl_normal   v3

   m4x4 oPos, v0, c[0]           // Transform position to clip space

   m4x4 r0, v0, c[17]            // Transformed Pshade (using texture matrix 0)
   mov oT0, r0

   m4x4 oT1, v0, c[4]            // Transform position to eye space
   m3x3 oT2.xyz, v3, c[8]        // Transform normal to eye space
};


technique technique_hlsl_bluemarble
{
   pass P0
   {
      // Only need to map variable names to hardware
      // registers like this for asm shaders:
      VertexShaderConstant[0]  = <matWorldViewProj>;
      VertexShaderConstant[4]  = <matWorldView>;
      VertexShaderConstant[8]  = <matITWorldView>;
      VertexShaderconstant[12] = <matWorld>;
      VertexShaderConstant[17] = <matTex0>;

      VertexShader = <asm_marble_vs>;
      PixelShader  = compile ps_2_0 hlsl_bluemarble();

      CullMode = CCW;
   }
}
```

We will now explain this example Effect file from the bottom up. The very last block of code in this Effect file defines a technique called `technique_hlsl_bluemarble` which has only one rendering pass. This single pass will use a vertex shader written in assembly language and a pixel shader written in HLSL. The first several lines in this pass declare five different matrices which will be loaded into specific hardware constant registers from high level Effect variables when this pass is invoked. This explicit mapping is only done in the Effect file for asm shaders. There are no explicit mappings done like this for the pixel shader, since it is written in HLSL. The next line declares the vertex shader to be used in this pass, an assembly shader called `asm_marble_vs`:

```
VertexShader = <asm_marble_vs>;
```

The following line defines the pixel shader, which will be compiled for the ps_2_0 target using the `hlsl_bluemarble()` function as its main entrypoint:

```
PixelShader  = compile ps_2_0 hlsl_bluemarble();
```

The block of code preceding the technique definition is the vertex shader written by hand in assembly language. Above this is `hlsl_bluemarble`, the main entrypoint for our HLSL pixel shader. If you have a look at this code, you will see that, in addition to the `tex1D()` intrinsic, this function calls several other utility functions such as `ambient()` and `soft_diffuse()`. These utility functions are defined earlier in this Effect and, since we're compiling for the ps_2_0 target, they are inlined into the resulting assembly.

If you look above the utility functions, you'll see the declaration of a pair of samplers called `NoiseSampler` and `MarbleSplineSampler`. These are declared just as before except that when used in an Effect file they can also be followed by the bracketed code defining the addressing and filtering sampler state to be used. Textures may also be defined in Effect files as shown above the sampler declarations. At the very top of the Effect, we see the declaration of a series of global variables which will be settable from the application level.

The Effect API

Now that we have written an effect and stored it in a file, we wish to use it from our application code. Naturally, the first thing that we will do is create the effect using the `D3DXCreateEffectFromFile()` API. Assuming this succeeds, we can use the Effect API to set the appropriate variables needed by our Effect. For example, we can set the matrices with the `SetMatrix()` entrypoint:

```
m_pEffect->SetMatrix ("matWorldViewProj", &m_matWorldViewProj);
m_pEffect->SetMatrix ("matWorldView", &m_matWorldView);
m_pEffect->SetMatrix ("matITWorldView", &m_matITWorldView);
m_pEffect->SetMatrix ("matWorld", &m_matWorld);
m_pEffect->SetMatrix ("matTex0", &m_ObjectParameters.m_matTex0);
```

We could also set any floats and vectors similarly:

```
m_pEffect->SetFloat ("noiseFrequency ", &m_fNoiseFreq);
m_pEffect->SetVector("g_Leye", &g_Leye);
```

Likewise, with textures:

```
m_pEffect->SetTexture("tVolumeNoise",  m_pVolumeNoiseTexture);
m_pEffect->SetTexture("tMarbleSpline", m_pMarbleColorSplineTexture);
```

With all of the proper constants set up, we can set the desired technique and render all of its passes (in this case, just one):

```
m_pEffect->SetTechnique(m_pEffect->GetTechniqueByName("technique_hlsl_bluemarble"));

m_pEffect->Begin(&cPasses, 0);
for (iPass = 0; iPass < cPasses; iPass++)
{
   m_pEffect->Pass(iPass);

   // Render geometry
}
m_pEffect->End();
```

As you can see, this is a straightforward process which hides several unnecessary burdens from the application. For example, the application never needs to know what hardware constant register to load `g_Leye` into or which sampler the noise texture should be bound to. These details are all managed by the D3DX Effects framework.

Integration into an engine without using D3DX Effects

We have found that some ISVs prefer not to wed their code too closely to D3DX because of cross-platform development or overhead concerns. As a result, while the use of D3DX Effects for HLSL shader management is very convenient, it is not required. Of course, giving up the convenience of D3DX Effects means that the application will have to take responsibility for tracking and setting up the appropriate constants and samplers prior to rendering with a given shader. We will now discuss how this is done.

Since you won't be creating D3DX Effects which trigger compilation of HLSL code, you must invoke the HLSL compiler explicitly in your application. In fact, this is very similar to the application code you would write for use of assembly shaders except that you call one of the `D3DXCompileShader*()` routines instead of one of the `D3DXAssembleShader*()` routines. You then pass the resulting asm code to the appropriate `CreatePixelShader()` or `CreateVertexShader()` entrypoint just as you would for an assembly shader that was assembled rather than compiled. An example of this usage is shown in the following code snippet:

```
if (FAILED (hr = D3DXCompileShaderFromFile (g_strVHLFile, NULL, NULL, "main", "vs_1_1",
NULL, &pCode, NULL, &m_VS_ConstantTable)))
{
   return hr;
}

if (FAILED (hr = m_pd3dDevice->CreateVertexShader ((DWORD*)pCode->GetBufferPointer(),
&m_HLSLVertexShader)))
{
   return hr;
}
```

You'll notice in the above code that the D3DXCompileShader*() routines have some additional parameters not found in the D3DXAssembleShader*() routines. Specifically, it is necessary to specify the name of the main entrypoint for the shader as well as the compile target ("main" and "vs_1_1" above). You can also optionally specify values of #defines, include files and flags to control generation of debug information, optimization, validation and matrix data ordering. All of these inputs are passed to the D3DXCompileShader*() routines via the first six parameters. The last three parameters are pointers to buffers which get filled in by the compiler: the binary assembly code, human-readable error messages (optional) and the constant table. The binary assembly code is what gets passed to CreatePixelShader() or CreateVertexShader(), while the constant table must be used by the application to know how to load the proper constant data prior to executing a given HLSL shader. We will devote the remainder of this discussion to the final parameter returned from the D3DXCompileShader*() routine, as this is the most critical piece to understand when integrating HLSL shaders into an application without the use of Effects. You can refer to the documentation for discussion of the other parameters.

The Constant Table

The constant table returned from the D3DXCompileShader*() routine is used to map high level constants and samplers to specific hardware constants and samplers. Non-static variables declared at global scope are considered input parameters to the compiled shader and must be properly initialized in order for the shader to execute correctly. The constant table provides this mapping. Typically, it is most convenient for an application to use the ID3DXConstantTable interface directly, as this does not require the application to parse the actual data structures of the constant table. The ID3DXConstantTable interface provides a number of convenient methods for looking up *handles* of known HLSL variables based upon their ASCII names. The appropriate values for these HLSL variables may then be set as shown in the following code snippet:

```
D3DXHANDLE handle;

if (handle = m_PS_ConstantTable->GetConstantByName(NULL, "ringFreq"))
{
   m_PS_ConstantTable->SetFloat(m_pd3dDevice, handle, m_fRingFrequency);
}

if (handle = m_PS_ConstantTable->GetConstantByName(NULL, "lightWood"))
{
   m_PS_ConstantTable->SetVector(m_pd3dDevice, handle, &lightWood);
}
```

Likewise, textures and sampler state must be set up correctly as shown in the following code snippet:

```
if (handle = m_PS_ConstantTable->GetConstantByName(NULL, "NoiseSampler"))
{
   m_PS_ConstantTable->GetConstantDesc(handle, &constDesc, &count);

   if (constDesc.RegisterSet == D3DXRS_SAMPLER)
   {
      m_pd3dDevice->SetTexture (constDesc.RegisterIndex, m_pVolumeNoiseTexture);

      // Set sampler states appropriate for the Noise Sampler
      m_pd3dDevice->SetSamplerState (constDesc.RegisterIndex, …, …);

   }
}
```

The implication of this, of course, is that render states, texture stage states and sampler states must be maintained by the application and are in no way encapsulated in the HLSL shader code as they would be using D3DX Effects.

Of course, particularly in any kind of shader-authoring tool, there may be no *a priori* application knowledge of the names of variables or samplers expected. In this case, it will be necessary to use the `ID3DXConstantTable::GetDesc()` method to determine the number of constants in the constant table. Subsequently, the application can use the `ID3DXConstantTable::GetConstantElement()` method rather than the `ID3DXConstantTable::GetConstantByName()` method used in the code snippets above. In general, it is a good idea to familiarize yourself with the `ID3DXConstantTable` interface if you intend to integrate support for HLSL shaders into your application without the use of D3DX Effects.

SDK Updates

Since the release of DirectX 9.0 and the subsequent DirectX 9.0a patch, Microsoft has committed to releasing periodic SDK updates for developers. These SDK updates do *not* contain Direct3D runtime changes, but do include upgrades to important D3DX tools including the HLSL compiler. It is highly recommended that you keep up to date with the latest released DirectX SDK updates so that you are using the latest compiler revision and generating the best possible asm from your HLSL source.

Conclusion

We have presented a detailed description of the Direct3D High Level Shading Language (HLSL) which is one of the most significant new features of DirectX 9. We have presented an introduction to the mechanics of the language itself and reinforced key concepts with sample shaders. We have also given some insights into the compilation process and how you can best write shaders for optimal performance. We hope this introduction has provided you with a solid foundation so that you can understand the

HLSL shaders presented in later chapters and begin integrating HLSL shaders into your own projects.

Acknowledgements

Thanks to ATI's 3D Application Research Group for providing the sample HLSL shaders. Thanks to Dan Baker and Loren McQuade of Microsoft for their feedback and specifically their contributions to the section on optimizations. Thanks also to Mark Wang and Wolfgang Engel for valuable comments which resulted in greater clarity.