High Level Shading Language

6.1 introduction

The core of the Direct3D 11 pipeline is its various programmable shader stages. These stages are where the vast majority of rendering tasks occur, and are also the means by which a CPU's power and flexibility are made available to the user. While various API functions can manipulate the state in which a shader program will be executed, the actual authoring of shader functionality and behavior is not done with these APIs. Instead, shader programs (also referred to as simply *shaders)* are authored, using a specially designed programming language known as *High Level Shading Language.*

High Level Shading Language (commonly abbreviated as *HLSL),* is primarily a C/C++-derived language with a simplified feature set. Things like semicolons for line termination, braces for statement blocks, and C-style function and struct declarations make the language immediately familiar to a C or C++ programmer. The major exceptions are that pointer types and operations are not supported, as well as C++-style templates. In addition, the language does not support a dynamic memory allocation model. As in C or C++, programs are written by authoring a function that serves as the entry point. This code is executed at runtime and runs until the end of the function is reached. It should be noted that in D3D11, the language is always compiled in advance, before shaders are bound to the pipeline, as opposed to being dynamically interpreted, like a scripting language. To support static compilation, the compiler features a C-style preprocessor. The preprocessor supports common uses such as macro definitions, conditional compilation, and include statements.

One of the key features of HLSL in D3D11 is that the same language is used to author shader programs for all shader stages. This provides a consistent set of standard features for all stages, allowing common code to be shared among shader programs written for each stage of the pipeline. Authoring in a high level language also allows many of the low-level details of the underlying graphics hardware to be abstracted, allowing shader programs to extract performance from a wide range of graphics hardware.

6.2 Usage Process

Figure 6.1 shows the typical sequence for authoring and running an HLSL program.

6.2.1 Authoring and Compilation

The usage process begins by authoring the shader program in HLSL. Typically, this is done by creating a standard ASCII text file containing the code for one or more shader programs. This can be done with any standard text editor or development environment. Once the code for the program is completed, it is compiled by the D3D shader compiler. The compiler is accessed through the D3DCompile function exported by the D3DCompiler DLL, or through the command line by using the standalone `fxc.exe` tool. There are also D3DX helper functions (D3DXCompileFromFile\D3DXCompileFromResource) that simplify the process of compiling directly from a file or from an embedded resource. In all cases, the compiler accepts configuration parameters, in addition to code itself. These parameters indicate the entry point function, the shader profile to use for compilation, a list of preprocessor macro definitions, a list of additional files to be included by the preprocessor, and a list of compilation flags (including optimization and debug options). If compilation fails or warnings are produced, the D3DCompile and D3DX functions store the warning or error messages in a string embedded in an ID3D10Blob object. As with C++ compilation, these messages include an informative message and a line number, so that the errors or warnings can be easily found and corrected. If `fxc.exe` is used, the messages will simply be output to the command line.

For debug builds of applications, it is typically desirable to compile shader programs with the D3D10_SHADER_DEBUG (/Zi switch in fxc) flag enabled. This causes the compiler to embed debug information into the bytecode stream, which can be used to map instructions and constants back to their respective lines in the HLSL source code. This enables source-level debugging of shader programs, which can be much more convenient than directly debugging the assembly. Note that it may also be desirable to disable optimizations by using D3D10_SHADER_SKIP_OPTIMIZATIONS (/Od switch in fxc), as this will prevent instruction reordering and folding, making it easier to debug the programs.

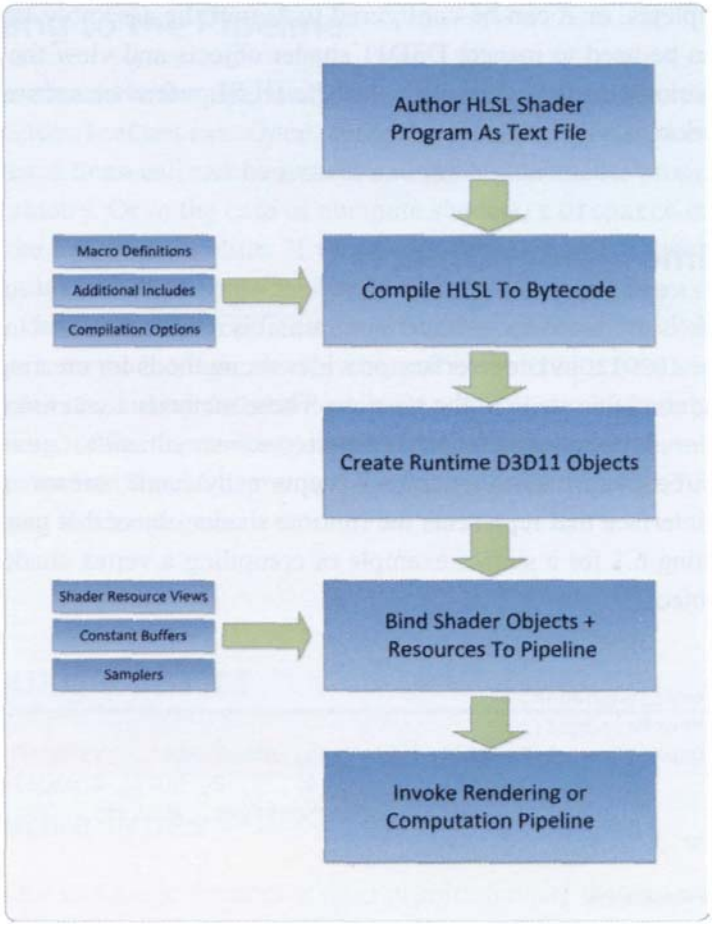

Figure 6.1. Shader program usage pipeline.

Shader programs can be debugged using PIX, a debugging and profiling utility included with the DirectX SDK. Consult the SDK documentation for additional information and usage instructions.

6.2.2 Bytecode

Once compilation successfully completes, the compiler outputs to an intermediate format known as *bytecode*. This is an opaque binary stream containing all of the assembly instructions for the shader, as well as embedded reflection and debugging metadata. If a programmer wants to inspect the resulting assembly instructions for verification, it can be obtained by using the D3DDisassemble function exported by the D3DCompiler DLL. The command-line tool fxc.exe will also output the assembly to the command line when

compilation completes, or it can be configured to output the assembly listing to a file. In addition, PIX can be used to inspect D3D11 shader objects and view the assembly code. For more information on shader assembly, see the HLSL reference section in the DirectX SDK documentation.

6.2.3 Runtime Shader Objects

Once the bytecode is produced for a shader program, it is ready to be used to create a D3D11 shader object. The ID3D11Device interface provides six methods for creating shader objects, one for each programmable stage of the pipeline. These methods are CreateVertexShader, CreateHullShader,CreateDomainShader,CreateGeometryShader,CreatePixelShader, and CreateComputeShader. Each of these accepts a bytecode stream and produces an ID3Dll*Shader interface that represents the runtime shader object that can be bound to the pipeline. See Listing 6.1 for a simple example of compiling a vertex shader and creating a runtime shader object.

```
ID3D10Blob* compiledShader;
ID3D10Blob* errorMessages;
HRESULT hr = D3DXllCompileFromFile( filePath, NULL, NULL, "VSMain",
                                    "vs_5_0", 0, 0, NULL, &compiledShader,
                                    &errorMessages, NULL );
if ( FAILED( hr ) )
{
    if ( errorMessages )
    {

        const char* msg = (char*)( errorMessages->GetBufferPointer() );
        Log::6et().Write( msg );
    }
  else
    {
        Log::Get().Write( "D3DXllCompileFromFile failed" );
    }
}
else
{
    ID3DllVertexShader* vertexShader = NULL;
    device->CreateVertexShader( compiledShader->GetBufferPointer(),
                                compiledShader->GetBufferSize(),
                                NULL,
                                &vertexShader );
}
```

Listing 6.1. Compiling a vertex shader from a file.

6.2.4 Binding to the Pipeline

Shader objects are bound to the pipeline by calling the various *SetShader methods available on an ID3D11DeviceContext. Once all required shader objects are bound to their corresponding stages, a Draw call can be issued, and the bound shader programs will be used to render the geometry. Or in the case of compute shaders, a Dispatch call can be issued, instead, to use the compute pipeline. If the shader program uses shader resources, samplers, or constant buffers, these must be bound before issuing the Draw or Dispatch call. This is done by calling the appropriate *SetShaderResources, *SetConstantBuffers, and *SetSamplers methods on the device context. Unordered access views are bound to the compute shader stage using CSSetUnorderedAccessViews, and are bound to the pixel shader stage using OMSetRenderTargetsAndUnorderedAccessViews. See Chapter 3, "The Rendering Pipeline," and Chapter 5, "The Computation Pipeline," for more information on configuring shader stages.

6.3 Language Basics

6.3.1 Primitive Types

High Level Shader Language features several primitive types that are common across all shader profiles. Like the primitive types in C++, HLSL primitives consist of various integral and floating-point numerical types. These types are listed in Table 6.1.

Type	Description
`bool`	32-bit integer that can contain the logical values true and false
`int`	32-bit signed integer
`uint`	32-bit unsigned integer
`half`	16-bit floating point value (provided only for backward compatibility; D3D11 shader profiles will map all half values to float)
`float`	32-bit floating point value
`double`	64-bit floating point value

Table 6.1. HLSL primitive types.

As in C or C++, these types can be used to declare both scalar and array values. Scalar variables support a standard set of mathematical operators, including addition, subtraction, negation, multiplication, division, and modulus. Scalars also support a standard set of logical and comparison operators, using the same syntax as C/C++. Array variables must have a fixed, statically declared size determined at the time of compilation, because HLSL does not support dynamic memory allocation.

6.3.2 Vectors

HLSL also supports declaring vector and matrix variables. Vector types support 1-4 components, with each component using the storage format specified by the primitive type. Vector variables are declared with the syntax shown in Listing 6.2.

```
vector<float, 4>    floatVector;    // 4-component vector with float components
vector<int, 2> intVector;           // 2-component vector with int components
```

Listing 6.2. Vector declaration syntax.

For convenience, HLSL predefines shorthand-type definitions for all combinations of primitive types and components. Listing 6.3 demonstrates the declaration of vector variables using the shorthand notation.

```
float4    floatVector;    // 4-component vector with float components
int2      intVector;      // 2-component vector with int components
```

Listing 6.3. Vector shorthand declaration syntax.

Vector types can be initialized when declared using array initializer syntax, with the number of values matching the number of components. Vector types also support constructor syntax, which allows passing scalar values or even other vectors. In addition, vectors can be initialized using a single scalar value, in which case the scalar value is replicated to all components. Listing 6.4 demonstrates these initialization patterns.

```
float2 vec0 = { 0.0f, 1.0f };
float3 vec1 = float3( 0.0f, 0.1f, 0.2f );
float4 vec2 = float4( vec1, 1.0f );
float4 vec3 = 1.0f;
```

Listing 6.4. Vector initialization syntax.

Individual components of a vector can be accessed using notation similar to accessing structures in C++. The members *x*, *y*, z, and *w* each correspond to the first, second, third, and fourth components of the vector, respectively. Alternatively the members *r*, *g*, *b*, and a can be used in the same manner. Vector components also support being accessed using array index syntax, which can be useful for looping through components. Listing 6.5 demonstrates usage of these accessors.

```
float4 floatVector = 1.0f;
float firstComponent = floatVector.x;
float secondComponent = floatVector.g;
float thirdComponent = floatVector[2];
```

Listing 6.5. Vector component access syntax.

In addition to supporting single-component access, vectors also support accessing multiple components simultaneously through the use of *swizzles*. A swizzle is an operation that returns one or more components of a vector in an arbitrary ordering. Swizzles can also replicate a component more than once, to create a new vector value with a greater number of components than in the source vector. While there are no restrictions on the ordering of components in swizzles, the notation used must be consistent. In other words, the *xyzw* notation cannot be mixed with the *rgba* notation within a single swizzle. Listing 6.6 contains sample code that uses swizzles in various ways.

```
float4 vec0 = float4(0.0f, 1.0f, 2.0f, 3.0f);
float3 vec1 = vec0.xyz;
float2 vec2 = vec1.rg;
float3 vec3 = vec0.zxy;
float4 vec4 = vec2.xyxy;
float4 vec5;
vec5.zyxw = vec5.xyzw;
```

Listing 6.6. Vector swizzle syntax.

If a vector value is assigned to a vector variable with a fewer number of components, the value is implicitly truncated when the assignment is performed. Programmers should always be careful not to inadvertently cause such a truncation, since values will be discarded. To make the truncation explicit, a C++-style cast or a swizzle can be used. As a convenience, the HLSL compiler will emit a warning when an implicit truncation occurs.

Vectors support the same set of mathematical, logical, and comparison operators supported by scalar variables. When these operators are used on vector types, the operation is performed on a per-component basis and produces a vector result value with a number of components equal to that of the operands.

6.3.3 Matrices

Matrix variables are declared and used in a manner similar to vector variables. Matrix declarations specify a base primitive type, in addition to the number of rows and columns. The number of rows and columns are listed to 4 each, making for a maximum of 16 individual components. Component access can be done with two-dimensional array syntax, where the first index specifies the row and the second specifies the column. Additionally, when only a single array index is specified, it returns the corresponding row of the matrix as a vector type. Matrices also support their own syntax for member access, which differs from the format used for vectors. These formats are shown in Listing 6.7.

```
float4x4 worldMatrix = float4x4( float4( 1.0f, 0.0f, 0.0f, 0.0f ),
                                 float4( 0.0f, 1.0f, 0.0f, 0.0f ),
                                 float4( 0.0f, 0.0f, 1.0f, 0.0f ),
                                 float4( 0.0f, 0.0f, 0.0f, 1.0f ) );
float matVal0 =   worldMatrix._m00;   // Value from first row, first column
float matVal1 = worldMatrix._12;      // Value from first row, second column
float matVal2 = worldMatrix[0][1];    // Value from first row, second column
float2 matVal3 = worldMatrix._11_12; // Swizzles
```

Listing 6.7. Matrix component access syntax.

As with vectors, operators used with matrix types are performed per-component. Consequently the multiplication operator should not be used to perform matrix multiplications and transformations. Instead the mul intrinsic function is provided for matrix/matrix and matrix/vector multiplications. See the "Intrinsic Functions" section in this chapter or the SDK documentation for more details.

Typically, shader programs initialize matrices in a row-major format and handle all vector/matrix transformations accordingly. However, in many cases the compiler will optimize the assembly such that the matrices contain column-major data, because that format allows for a more efficient expression in assembly using four dot products. In addition, the compiler defaults to treating all matrices declared in constant buffers as if they contained column-major data, even when row-major transformation is performed with the mul intrinsic. Consequently, matrices may need to be transposed by the host application before being set into the constant buffer, or the matrix would have to be transposed in the shader program. This behavior can be changed by passing D3D10_SHADER_PACK_MATRIX_ROW_MAJOR to shader compilation functions, or by declaring matrices with the row_major modifier.

6.3.4 Structures

HLSL allows for custom structure declarations, with rules very similar to C++. Structures can contain an arbitrary number of members with scalar, vector, or matrix primitive types.

They can also contain members with array types, or with other custom structure types. Listing 6.8 demonstrates a simple structure declaration.

```
struct MyStructure
{
    float4 Vec;
    int Scalar;
    float4x4 Mat;
    float Array[8];
```

Listing 6.8. Structure declaration syntax.

6.3.5 Functions

HLSL allows for declaration of free functions in a manner similar to C/C++. Functions can have any return type (including void), and can accept an arbitrary number of parameters. Since HLSL does not support reference or pointer types, it has its own syntax for providing input/output semantics for parameters. This syntax includes four modifiers for parameters, which are listed in Table 6.2.

Modifier	Description
in	Parameter is an input to the function. Any changes to the parameter value are temporary, and are not reflected in the calling code. Similar to pass-by-value in C++. This is the default modifier.
out	Parameter is an output of the function. The value of the parameter must be set by the function, and this change is reflected in the calling code.
inout	Parameter is both an input and an output to the function. The function can access the value of the parameter set by the calling code, and changes that occur in the function are reflected in the calling code. Similar to pass-by-reference in C++.
uniform	Parameter is constant for the duration of execution. For functions that aren't entry points, this modifier is equivalent to in. For entry point functions, the parameter is declared as part of the SParam default constant buffer.

Table 6.2. Function parameter modifiers.

When using functions, it is important to keep in mind that shader programs do not use a traditional stack as in C++. Consequently, it is not possible to recursively call functions. Instead dynamic loop constructs must be used to implement recursive algorithms.

6.3.6 Interfaces

Interfaces in HLSL are similar to abstract virtual classes in C++, and are primarily used to enable dynamic shader linkage. Interfaces can only contain member functions, and not member variables. Listing 6.9 contains a declaration of a simple interface type.

```
interface MyInterface
{
    float3 Method1();
    float2 Method2( float2 param );
};
```

Listing 6.9. Interface declaration syntax.

The methods declared in an interface are always considered to be pure virtual functions, and thus no virtual keyword is necessary. See the "Dynamic Shader Linkage" section for more information on using interfaces in HLSL.

6.3.7 Classes

HLSL classes, like classes in C++, can contain member variables, as well as member functions. They can also inherit from a single base class, and can inherit from multiple interfaces. However if a class inherits from an interface, it must fully implement all methods declared in that interface. Listing 6.10 contains a simple class declaration that implements an interface.

```
interface MyInterface
{
    float3 Method1( );
    float2 Method2( float2 param );
};

class MyClass : MyInterface
{
    float3 Member1;
```

```
    float3 Method1(  )
    {
        return Member1;
    }
};
float2 MyClass: :Method2( float2 param )
{
    return Member1.xy + param.xy;
}
```

Listing 6.10. Class declaration syntax.

While instances of classes can be declared and have methods called on them in normal shader programs, they are primarily used to enable dynamic shader linkage. See the "Dynamic Shader Linkage" section for more information on using classes in HLSL.

6.3.8 Conditionals

Conditional code execution is supported through the if and case statements, which work in the same manner as in C/C++. All if statements operate on a single Boolean value, which can be created through the use of logical and comparison operators. It should be noted that the Boolean result of vector operations cannot be directly used, because such operations produce a vector result, rather than a single Boolean value. By extension this same principle applies to switch statements.

It should be noted that conditional branching based on values that are only known during the execution of a program can be expressed by the compiled shader assembly in one of two ways: *predication* or *dynamic branching.* When predication is used, the compiler will emit code that evaluates the expressions on both sides of the branch. A compare instruction is then used to "select" the correct value, based on the result of a comparison. Dynamic branching, on the other hand, emits branching instructions that actually control the flow of execution in the shader program. Thus, it can be used to *potentially* skip unneeded calculations and memory accesses.

Whether or not instructions in a branch are skipped is based on the coherency of the branching. In other words, simultaneous executions of a shader program must all choose the same branch, to prevent the hardware from executing both sides of the branch. Typically, the hardware will simultaneously execute a number of consecutive vertices in the vertex shader, consecutive primitives in the geometry shader, and consecutive grids of pixels in the pixel shader. For compute shaders, the threads are explicitly mapped to thread groups, requiring coherency within each group. In addition, dynamic branching can impose

a constant performance penalty, due to the overhead incurred from executing the actual branch instructions. The basic texture Sample method also cannot be used within a dynamic branch in a pixel shader, since the partial derivatives may not be available, because pixels in a quad may potentially take different branches. Consequently, it is important to weigh the additional overhead of dynamic branching against the number of potentially skipped instructions, while also considering the coherency of the branch.

By default, the compiler will automatically choose between predication and dynamic branching, using heuristics. However, it is possible to override the compiler's decision-using attributes. See the "Attributes" section in this chapter for more information.

6.3.9 Loops

HLSL supports for, while, and do while looping constructs. Like the conditional statements, their syntax and usage is identical to their C/C++ equivalents. They are also similar to conditionals, in that they can potentially produce dynamic flow control instructions when looping based on runtime values, and they thus have the same performance characteristics with regard to coherency.

6.3.10 Semantics

In HLSL, semantics are string metadata attached to variables. They serve three primary purposes when used in shader programs:

1. To specify the binding of variables used for passing values between shader stages
2. To allow shader programs to accept special system values generated by the pipeline
3. To allow shader programs to pass special system values interpreted by the pipeline

The code in listing 6.11 demonstrates these three use cases.

```
cbuffer PSConstants
{
    float4x4 WVPMatrix;
}
```

```
void VSMain(   in float4 VPos  :  POSITION        // Binds variable to Input
                                                  // Assembler
               in uint VID  :  SV_VertexID        // Binds variable to a
                                                  // system generated value
               out float4 OPos  :  SV_Position    // Tells the pipeline to
                                                  // interpret the value as the
                                                  // output vertex position
           )
{
    OutPos = mul( VPos, WVPMatrix );
}
```

Listing 6.11. Using semantics in a vertex shader.

In the example given in Listing 6.11, applying the POSITION semantic to the input allows the shader to specify which element from a vertex it requires. Thus, the input assembler will use the input layout to determine the proper byte offset within a vertex that is required to read in the appropriate element. In the same way, values passed from one shader stage to another can be identified with semantics. This allows the runtime to ensure that the proper output value is matched to a given input parameter.

System-value semantics, denoted with an SV_ prefix, are instead used to accept a value from the runtime, or to pass a value to the runtime. In the example shader, SV_VertexID specifies that the parameter should be set to a value that indicates the index of the vertex within the vertex buffer. The SVPosition semantic indicates that the value being output is the final transformed vertex position that should be used for rasterization. For a full listing of supported system-value semantics, consult the HLSL reference section of the SDK documentation. You can also refer to Chapter 3 for more information on the meaning and usage of each of these system value semantics.

6.3.11 Attributes

Attributes are special tags that can be inserted into HLSL code to modify the assembly code emitted by the compiler, or to control how the shader is executed by the pipeline. Some are purely optional, while others are required for certain shader types. In all cases, they are used by being declared immediately before the function, branch, loop, or statement they are affecting. Table 6.3 contains a list of all attributes, with a short description.

a matching alignment. Some compilers also support pragmas for specifying the packing alignment. The packing for an HLSL constant buffer can also be manually specified though the packoffset keyword. Listing 6.13 demonstrates usage of packoffset to declare a tightly packed constant buffer with four-byte alignment.

```
cbuffer VSConstants
{
    float4x4 WorldMatrix : packoffset(c0);
    float4x4 ViewProjMatrix : packoffset(c4);
    fioat3 Color : packoffset(c8);
    uint EnableFog : packoffset(c8.w);
    float2 ViewportXY : packoffset(c9);
    float2 ViewportWH : packoffset(c9.z);
}
```

Listing 6.13. Specifying constant buffer packing.

When a constant buffer is declared in HLSL, the compiler automatically maps it to one of 15 constant buffer registers for the corresponding stage of the pipeline. These registers are named cb0 through cb14, and the index of the register directly corresponds to the slot passed to *SSetConstantBuffers when binding a constant buffer to a shader stage. The register index can be queried for a shader program using the reflection APIs, so that the host application knows which slot to specify. Alternatively, the register index can be manually specified in the HLSL declaration, using the register keyword. Listing 6.14 demonstrates the usage of this keyword.

```
cbuffer VSConstants : register(cb0)
{
    float4x4 WorldMatrix : packoffset(c0);
    float4x4 ViewProjMatrix : packoffset(c4);
    float3 Color : packoffset(c8);
    uint EnableFog : packoffset(c8.w);
    float2 ViewportXY : packoffset(c9);
    float2 ViewportWH : packoffset(c9.z);
}
```

Listing 6.14. Specifying the constant buffer register index.

6.4.1 Default Constant Buffers

Any global variables declared without the static const modifiers will be treated by the compiler as constants inside a default constant buffer named $Globals. Similarly,

parameters to a shader entry point function marked as uniform will be placed inside another default constant buffer, named $Params.

6.4.2 Texture Buffers

Since constant buffers are optimized for small sizes and uniform access patterns, they can have undesirable performance characteristics in certain situations. One common situation is an array of bone matrices used for skinning. In this scenario, each vertex contains one or more indices indicating which bone in the array should be used to transform the position and normal. On many hardware types, this access pattern will cause the threads executing the shader program to serialize their access to the bone data, stalling their execution.

To rectify this issue, D3D11 allows a special type of constant buffer known as a *texture buffer*. A texture buffer uses the same syntax for declaring constants and accessing them in a shader program. However, under the hood, memory access will be done through the texture fetching pipeline. Thus it will have the same cached, asynchronous access pattern as a texture sample. Listing 6.15 demonstrates declaration and usage of a texture buffer in a vertex shader program.

```
cbuffer VSConstants : register(cb0)
{
    float4x4 WVPMatrix;
}

tbuffer Bones : register(t0)
{
    float4x4 BoneMatrices[256];
}
float4 VSMain( in float4 VtxPosition : POSITION,
               in uint BoneIndex : BONEINDEX ) : SV_Position
{
    float4x4 boneMatrix = BoneMatrices[BoneIndex];
    float4 skinnedPos = mul( VtxPosition, boneMatrix );
    return mul( skinnedPos, WVPMatrix );
}
```

Listing 6.15. Using a texture buffer in a vertex shader program.

One important point to keep in mind when declaring a texture buffer is that it must be mapped to a texture register, rather than a constant buffer register. Also, the host application must create a shader resource view for the texture buffer and bind it through *SSetShaderResources, as opposed to calling *SSetConstantBuffers. The texture buffer resource itself must also be created as a texture resource, rather than a buffer resource.

6.5 Resource Objects

HLSL provides several types of resource objects that allow HLSL shader programs to interact with the various resource types available in D3D11. Each object type exposes the functionality of the resource through a set of methods that can be called on a declared object of that type. Most of these methods provide a means of reading data from the resource when given an address or index, while some also provide information about the resource itself. In the case of read/write resources, methods are also exposed for writing to the resource data.

All of the read-only resource objects correspond to a type of shader resource view, while the read/write resource objects correspond to a type of unordered access view. As with constant buffers, each declaration of a resource object is mapped by the compiler to a register index that corresponds to the slot passed to *SSetShaderResources, OMSetRenderTargetsAndUnorderedAccessViews, or CSSetUnorderedAccessViews. Like constant buffers, the register can be manually specified when declaring the resource object, using the `register` keyword. For shader resource views the registers are t0 through t127, and for unordered access views the registers are u0 through u7. Refer to Chapter 2 for more information regarding resources, and how they are bound to the pipeline.

6.5.1 Buffers

The buffer resource objects correspond to a shader resource view created for an ID3D11Buffer resource. For the most part, they have a very simple interface that consists of methods for obtaining the size of the resource, a Load method for reading a value when given an address, and an array operator for reading a value when given an index. In cases where a buffer can contain different types, the data type returned by the Load method or array operator is specified using a template-like syntax. Listing 6.16 demonstrates declaration of a standard Buffer object with float4 return type, corresponding to DXGI_FORMAT_R32G32B32A32_FLOAT.

```
Buffer<float4> Float4Buffer : register(t0);
```

Listing 6.16. Declaring a float4 buffer.

Buffer

The Buffer object is the simplest resource object type in HLSL. A Buffer provides read-only access to its data through the Load method and an array operator, both of which return the data at the specified index. It also provides the GetDimensions method for querying the size of the buffer in bytes.

ByteAddressBuffer

The ByteAddressBuffer type allows access to a buffer using a byte offset, rather than an index. This functionality is exposed through the Load, Load2, Load3, and Load4 methods, which return 1, 2, 3, or 4 `uint` values, respectively. Since those methods only return type `uint`, buffers containing other types of data must have the return value manually converted using one of the conversion/casting intrinsics.

StructuredBuffer

A StructuredBuffer provides read-only access to a buffer containing elements defined by a structure, as opposed to the regular Buffer type, which can only access types corresponding to DXGI_FORMAT values. The structure must be declared in HLSL as a `struct`, and that type must be specified as the template argument when declaring the StructuredBuffer object.

Read/Write Buffers

Read/Write buffers allow random-access reads and writes to buffer resources. Since the reads and writes are not automatically synchronized by the device, synchronization intrinsics or atomics must be used to control memory access across threads. By default, a synchronization intrinsic will only cause a write to flush across that thread group. In order for the write to be flushed across the entire GPU, the buffer declaration must use the `globallycoherent` prefix.

The three types of read/write buffers are RWBuffer, RWByteAddressBuffer, and RWStructuredBuf f er. For reads, they behave exactly like their read-only equivalents, and use the same methods. For writing to memory, the array operator can be used for RWBuffer and RWStructuredBuffer, while RWByteAddressBuffer provides the Store, Store2, Store3, and Store4 methods. RWByteAddressBuffer also provides a set of interlocked methods that allow atomic operations to be performed on the contents of the resource. For a RWStructuredBuffer that contains a hidden counter (which is the case for structured buffer resources created with the D3D11_BUFFER_UAV_FLAG_COUNTER flag), it can be incremented or decremented with IncrementCounter and DecrementCounter. See Chapter 2 for more details on structured buffer resources.

6.5.2 Append/Consume Buffers

Append and consume buffers allow adding and removing of values from a resource in a pixel or compute shader. The addition and-removal operations are done at the end of a buffer, causing it to behave somewhat like a stack. However unlike traditional stacks, the

ordering of additions and removals is not guaranteed, because the resource is accessed by many threads simultaneously. Addition of values is done with the Append method of the AppendStructuredBuffer type, while removal is done with the Consume method of the ConsumeStructunedBuffer type. As their names suggest, these types are both structured buffer resources.

6.5.3 Stream-Output Buffer

The stream-output buffer is a simple resource object used by the geometry shader to emit primitive vertex data. Unlike other resource types, it is not declared globally and mapped to a register. Instead, it is taken as an inout parameter to the geometry shader entry point function. Three types of stream-output buffers can be declared, each corresponding to a different primitive type. They are PointStream, LineStream, and TriangleStream, which correspond to a point strip, a line strip, and a triangle strip, respectively. The parameter declaration should also include a type as the template argument, which indicates the data format of the vertices. Typically, this type is a structure containing all vertex data that needs to be passed to the pixel shader, or that will be output by the stream output stage. Adding a vertex to the strip is done with the Append method, while RestartStrip cuts the current strip and begins a new strip. If a strip is cut before the minimum number of vertices is appended, the incomplete primitive is discarded.[1]

6.5.4 Input and Output Patches

HLSL includes two patch types, which are used for accessing an array of control point data in the hull and domain shaders, and are also available as input to the geometry shader.[2] The InputPatch contains an array of points that can be declared as an input attribute for the hull shader, the patch constant function, and the geometry shader. The OutputPatch contains an array of points that is declared as an input to the domain shader. Both of types are accessed using the array operator, and expose a Length property for determining the number of elements they contain.

[1] Further details on the declaration and usage of stream output buffers can be found in the "Geometry Shader" and "Stream Output" sections of Chapter 3.

[2] The geometry shader is typically not thought of as being able to receive an *InputPatch* as its input, but it is valid to declare and use this in the same manner as seen with the hull shader. This can be used to perform higher-order surface operations in the geometry shader if the control patches are not consumed by the tessellation stages.

6.5.5 Textures

Textures are among the most commonly-used resources in the rendering pipeline, and make use of specialized texture sampling units in graphics hardware. Consequently HLSL exposes a large interface for sampling texture data, to allow shader programs to make the most use of the hardware's capabilities. HLSL includes read-only texture resource object types, corresponding to the various texture resource types in D3D11. A full listing of these types is given in Table 6.4.

Intrinsic	Description
Texture1D	One-dimensional texture
Texture1DArray	Array of one-dimensional textures
Texture2D	Two-dimensional texture
Texture2DArray	Array of two-dimensional textures
Texture2DMS	Two-dimensional texture with multisampling
Texture2DMSAr>ray	Array of two-dimensional textures with multisampling
Texture3D	Three-dimensional texture
Texture3DArray	Array of three-dimensional textures
TextuneCube	Array of six 2D textures, representing faces of a cube
TextureCubeArray	Array of cube textures

Table 6.4. Texture resource objects.

The texture resource object types each support a subset of all of the texture operations, and consequently, each object type has its own interface. Consult the HLSL documentation to see if a method is supported for a given texture object type. The following sections describe how each of the methods can be used for accessing a texture object.

Sample Methods

Traditional texture sampling operations are performed with the Sample family of methods available on the texture objects. These methods each take a set of floating-point texture coordinates representing the memory location to sample, where each component is normalized to the range [0,1]. The number of components depends on the texture type. For instance, a Texture1D only accepts a single float, while a Texture2D accepts two floats, and a Texture3D accepts three. If a texture array is used, an additional floating-point component is passed, indicating the index of the array to use.

The Sample method allows for hardware texture filtering (minification, magnification, and mip-map interpolation) to be performed, according to a given sampler state. The

sampler state must be declared as a SamplerState object in HLSL, which is mapped to a sampler register (s0 through s15) in the same way constant buffers and resource objects are mapped to their register types. The index of the register corresponds to the slot passed to the *SSetSamplers methods available on the device context. A declared SamplerState object can then be passed to the Sample method, which will in turn use the specified sampler states for filtering. Listing 6.17 contains a simple pixel shader that demonstrates calling Sample on a Texture2D using a sampler state.

```
SamplerState LinearSampler : register(s0);
Texture2D ColorTexture : register(t0);

float4 PSMain( in float2 TexCoord : TEXCOORD ) : SV_Target
{
    return ColorTexture.Sample( LinearSampler, TexCoord );
}
```

Listing 6.17. Sampling a texture in a pixel shader.

When Sample is called on a texture that contains multiple mip-map levels, the mip-map level is automatically selected based on the screen-space derivatives (also known as gradients) of the texture coordinates. This is why Sample is only available in pixel shaders, and only outside of dynamic loop or branch constructs. To use texture sampling in other shader stages, or inside a dynamic branch/loop in a pixel shader, other variants of the Sample method are available, which allow the gradients or the mip level to be explicitly specified. These are SampleGrad and SampleLevel, respectively. A third method, SampleBias, works like Samplelevel with the addition of a bias value.

The return type of a sample method depends on the DXGI_FORMAT specified when creating the shader resource view bound for the texture object. Thus, DXGI_FORMAT_R32G32B32A32_FL0AT will return a float4, DXGI_FORMAT_R32G32_UINT will return a uint2, and DXGI_F0RMAT_R8G8B8A8_UN0RM will return a float4 .

SampleCmp Methods

Rather than directly returning a texture value, SampleCmp returns the result of comparing the sampled value against a value passed to the method as the CompareValue parameter. The returned value is equal to 1.0 if the comparison passes, and 0.0 if the comparison fails. This makes the method quite natural for implementing shadow mapping techniques.

The inequality used in the comparison is specified in the ComparisonFunc member of the D3D11_SAMPLER_DESC structure used to create the sampler state passed to the method. Note that the filter specified for the sampler state must be one of the COMPARISON values of the D3D11_FILTER enumeration. Also, the sampler state declared in HLSL must have the SamplerComparisonState type if it is to be passed to SampleCmp. If a linear filtering mode

is specified for the sampler state, the texture will be sampled multiple times and compared against the comparison value. The final return value is then the filtered result of all comparisons. This can be used to efficiently implement percentage closer filtering for shadow maps.

Gather Methods

The Gather family of methods returns four values when given a single texture coordinate. The values come from a 2x2 quad of texels, using the same location that would be used for bilinear filtering. GatherRed returns the four red components of the texels, GatherGreen returns the four green components, GatherBlue returns the four blue components, and GatherAlpha returns the four alpha components. Gather returns the four red components, making it functionally equivalent to GatherRed. Since the method returns a 2x2 quad, it can only be used on Texture2D and Texture2DArray resource objects. GatherCmp methods are also available, which work similarly to SampleCmp.

One of the more useful cases for the Gather methods is in pixel shaders that perform image processing. Normally gathering rgb values for a 2x2 quad of texels requires four sample operations, with one for each texel. This can be made more efficient by calling GatherRed, GatherGreen, and GatherBlue retrieve the same data in only three instructions. Another useful case for Gather instructions is implementing custom shadow map filtering kernels, which can be done efficiently with GatherCmp.

Load Methods

The Load method provides direct, unaltered, read-only access to texture data in a manner similar to what we have seen with the buffer types. Like the Buffer Load method, it takes an `int` index parameter, rather than a [0,1] texture address. Since it is the most basic means of accessing texture data, it is available on all texture resource types and in all stages. The array operator is also available, which provides similar functionality. For multisampled textures, Load provides access to the individual subsamples of a pixel. This functionality can be used to implement custom MSAA resolves, or to integrate MSAA into a deferred rendering pipeline. Like the Load method for a Buffer object, the return type is determined by the type that was used as a template parameter when declaring the texture object.

Get Methods

Texture resource objects implement several methods for querying information about the underlying resource. All texture resource object types implement GetDimenslons, which returns the size, number of mip levels, number of samples (for multisampled textures), and number of elements (for texture arrays). Texture2DMS objects also support the GetSamplePosition method, which returns the position of a MSAA sample point within the pixel given a sample index.

Cube Textures

Cube textures are a special case of texture arrays meant for representing the six sides of a cube, and are often used for reflection or environment maps. While they can be accessed like a normal texture array through the Texture2DArray type, they can also be declared as a TextureCube type. When calling a Sample method on TextureCube, the texture coordinate passed in is a normalized three-component direction vector. The texture is then sampled by choosing the texel pointed to by the direction vector. This usage is further described in the cube texture section of Chapter 2.

Read/Write Textures

Read/write textures function in a similar manner to read/write buffer resources. Random-access reads and writes are supported through the array operator, and GetDimensions can be called to query the size of the resource. The read/write resource object types supported in HLSL are RWTexturelD, RWTexturelDArray, RWTexture2D, RWTexture2DArray, and RWTexture3D.

6.6 Dynamic Shader Linkage

Since programmable graphics hardware has become mainstream, applications employing three-dimensional graphics have developed increasingly large and complex sets of shader code for implementing their graphical features. However prior to Direct3D 11, HLSL provided no built-in means of enabling dynamic dispatch in shader programs. Shader Model 3.0 introduced the ability for shaders to dynamically branch on values. However, the heavy performance penalties of doing so made it prohibitively expensive as a means for implementing dynamic dispatch. Static branching has also been supported since Shader Model 2.0, but its limitations on the number of branches, as well as performance implications, have made it similarly inadequate. Consequently many applications resorted to statically compiling all required permutations of a shader program, either by using macros and conditional compilation or by piecing together code from smaller fragments. While this approach has the advantages of not requiring dynamic branching, and provides the optimizer with full access to a complete program, the number of required permutations grows exponentially as new options are added. This is often referred to as the *shader combinatorial explosion.* Thus, programmers have often been faced with a difficult choice between decreasing shader performance, increasing shader compilation times, or making their shader build pipeline more complex.

To rectify this situation, Direct3D 11 has introduced a feature known as *dynamic shader linkage*. It essentially allows applications to dynamically choose from multiple implementations of an HLSL code path when binding a shader program to the pipeline, effectively allowing dynamic dispatch at the level of a Draw or Dispatch call. We will explore this capability further in the following sections to understand how it can be used in a real-time rendering application.

6.6.1 Authoring Shaders for Dynamic Linkage

Shader programs that will make use of dynamic linkage must be authored using HLSL interfaces and classes. Essentially, the procedure is similar to virtual dispatch in C++: interfaces are declared with a set of methods, and the shader code calls methods on those interfaces. Then at runtime, the host application assigns an instance of a class that implements the interface to be used for the duration of a Draw or Dispatch call. Shader programs using interfaces can declare a global instance of an interface, just like any other variable type. That instance then functions like a polymorphic pointer in C++, and forwards any methods called on it to the class instance specified by the host application. Listing 6.18 demonstrates the syntax for declaring an interface instance and calling one of its methods.

```
interface Light
{
    float3 GetLighting( float3 Position, float3 Normal );
};

Light LightInstance;

float4 PSMain( in float3 Position : POSITION,
               in float3 Normal : NORMAL ) : SV_Target
{
    float3 lightColor = LightInstance.GetLighting( Position, Normal );
    return float4( lightColor, 1.0f );
}
```

Listing 6.18. Calling a method on an interface.

Any classes that can possibly be used to implement an interface must also be declared in the HLSL shader program, or included through the use of a #include pragma. If the class has member variables, an instance of the class must be declared in a constant buffer. This allows the host application to specify values for those members at runtime. Listing 6.19 demonstrates the syntax for declaring a class that implements an interface, and declaring an instance in a constant buffer.

```
class DirectionalLight
{
    float3 Color;
    float3 Direction;
    float3 GetLighting( float3 Position, float3 Normal )
    {
        return saturate( dot( Normal, Direction) ) * Color;
    }
};

cbuffer ClasslnstanceS : register( cb0 )
{
    DirectionalLight DLightlnstance;
}
```

Listing 6.19. Declaring a class instance.

6.6.2 Linking Classes to Interfaces

At runtime, applications employing dynamic shader linkage must specify which class will be used to implement an interface used by a shader program. The first step is to create an ID3D11ClassLinkage interface, which is done by calling ID3DllDevice: :CreateClass Linkage. Once the class linkage interface is created, it can be bound to an instance of a shader program. This is done by passing the ID3D11Class Linkage interface as the pClassLinkage parameter of CreateVertexShader, CreateHullShader, CreateDomainShader, CreateGeometryShader, CreatePixelShader, or CreateComputeShader. Listing 6.20 demonstrates this process for a pixel shader.

```
ID3D10Blob* compiledShader;
ID3D10Blob* errorMessages;
HRESULT hr = D3DXllCompileFromFile( filePath, NULL, NULL, "PSMain",
                                    "ps_5_0", 0, 0, NULL, &compiledShader,
                                    &errorMessages, NULL );

ID3DllClassLinkage* classLinkage = NULL;
if ( SUCCEEDED( hr ) )
{
    device->CreateClassLinkage( SclassLinkage );
    device->CreatePixelShader(  compiledShader->GetBufferPointer(),
                                compiledShader->GetBufferSize(),
                                classLinkage,
                                &pixelShader );
}
```

Listing 6.20. Creating and binding a class linkage.

Once the class linkage is bound to a shader, an ID3DllClassInstance interface can be retrieved. A class instance represents one of the classes declared in the shader that can be used to implement an interface. The simplest means to acquire one is to call GetClassInstance on the ID3DllClassLinkage interface, and pass the name of a class instance declared in the shader program. CreateClassInstance can also be used for shader programs where a class instance wasn't declared in a constant buffer, which is useful for classes that don't contain member variables. Listing 6.21 demonstrates the process of acquiring a class instance interface.

```
ID3DllClassInstance* dLightInstance = NULL;
classLinkage->GetClassLinkage( L"DLightInstance", 0, &dLightInstance );
```

Listing 6.21. Acquiring a class linkage.

The final step for using dynamic linkage is to specify an array of class instances when binding a shader program to the pipeline. The ID3DllDeviceContext::*SSet Shader methods all have a ppClassInstances parameter that accepts an array of ID3DllClassInstance interfaces. This array must contain one class instance for each the interfaces used in the shader program. Listing 6.22 demonstrates initializing and passing such an array for a pixel shader.

```
ID3DllClassInstance* classInstances[1];
classInstances[0] = dLightInstance;
deviceContext->PSSetShader( pixelShader, classInstances, 1 );
```

Listing 6.22. Specifying class instances when binding a shader.

Each interface used in a shader program has a unique index, which corresponds to an index of the array passed to *SSetShader containing the class instances. This index can be retrieved using the ID3D11ShaderReflectionVariable interface, which is part of the shader reflection APIs. See the "Shader Reflection" section for more details.

6.7 Intrinsic Functions

HLSL provides a built-in set of global functions known as *intrinsics*. Like intrinsics in C or C++, HLSL intrinsics often directly map to specific shader assembly instructions. This

provides HLSL shader programs with full access to the functionality provided by the assembly instruction set. In a few cases, intrinsic functions also provide an optimized set of common mathematical operations, which spares the programmer from implementing them manually. The following sections contain tables listing intrinsic functions grouped by category, with a brief description for each. For a full description (including return types and parameters), consult the SDK documentation

6.7.1 Mathematical Functions

Vector/Matrix Operations

Table 6.5 contains a list of all vector and matrix math intrinsics supported by HLSL. These intrinsics implement mathematical operations that can be performed on vector and matrix data types.

Intrinsic	Description
mul	Matrix/Matrix, Matrix/Vector, or Vector/Vector multiplication
dot	Vector dot product
cross	Vector cross product
transpose	Matrix transpose
determinant	Matrix determinant
length	Vector length/magnitude
normalize	Geometric vector normalize
distance	Computes the distance between two points
faceforward	Flips a surface normal direction such that it points in a direction opposite to an incident vector
reflect	Computes a reflection vector, given normal and incident vectors
refract	Computes a refraction vector, given an entering direction, normal vector, and refraction index

Table 6.5. Vector/matrix math intrinsics.

General Math Functions

Table 6.6 contains a listing of HLSL's general math intrinsics. These instrinsics are mostly scalar math operations, similar to those found in the C Standard Library.

Intrinsic	Description
cos	Cosine function
sin	Sine function
tan	Tangent function
acos	Arcosine function
asin	Arcsine function
atan	Arctangent function
atan2	Signed arctangent function
sincos	Performs sine and cosine simultaneously
cosh	Hyberbolic cosine
sinh	Hyperbolic sine
tanh	Hyperbolic tangent
log	Natural logarithm (base e)
log2	Logarithm (base2)
logie	Logarithm (base 10)
exp	Exponential (base e)
exp2	Exponential (base 2)
pow	Raises a number to a power
sqrt	Square root
abs	Absolute value
trunk	Floating point truncation
floor	Return largest integer less than a value
ceil	Returns smallest integer greater than a value
round	Rounds to nearest integer value
frac	Returns the fractional part of a value
fmod	Floating point remainder
modf	Returns integer and fractional parts of a value
countbits	Number of storage bits for an integer
sign	Returns the sign of a value
all	Returns true if all components of a value are non-zero
any	Returns true if any components of a value are non-zero
clamp	Restricts a value to a specified minimum and maximum
degrees	Converts a value from radians to degrees
firstbithigh	Gets the first set bit of an integer, starting at the highest-order bit and moving downward
firstbitlow	Gets the first set bit of an integer, starting at the lowest-order bit and moving upward

frexp	Returns the mantissa and exponent of a floating-point value
isfinite	Returns true if a floating point value is not infinite
isinf	Returns true if a floating point value is equal to -INF or +INF
isnan	Returns true if a floating point value is equal to NAN or QNAN
ldexp	Multiplies a value to another value equal to 2 raised to a specified exponent
lerp	Linearly interpolates between two values, using a specified scale value
lit	Computes ambient, diffuse, and specular values for a Blinn-Phong BRDF
mad	Multiplies two values and sums the product with a third value
max	Returns the maximum of two values
min	Returns the minimum of two values
modf	Splits a floating-point value into integral and fractional parts
radians	Converts a value from degrees to radians
rep	Calculates a fast, approximate reciprocal of a value
reversebits	Reverses the order of the bits in an integer value
rsqrt	Calculates the reciprocal of the square root of a value
saturate	Clamps a value to the [0, 1] range
smoothstep	Uses a Hermite interpolation to calculate a value between 0 and 1, using a specified value, minimum, and maximum
step	Returns 1 if one specified value is greater than another, and 0 otherwise

Table 6.6. General math intrinsics.

6.7.2 Casting/Conversion Functions

In some cases, a resource will contain values in a particular data format that cannot be directly read in HLSL. Consequently, HLSL provides casting and conversion intrinsics so that shader programs can cast or convert a resource value to an appropriate data type. These intrinsics are listed in Table 6.7.

Intrinsic	Description
asfloat	Reinterprets a value as a float
asdouble	Reinterprets two 32-bit values as a double
asint	Reinterprets a value as an int
asuint	Reinterprets a value as a uint
f16to32	Converts a 16-bit floating-point value to a 32-bit floating-point value
f32to16	Converts a 32-bit floating-point value to a 16-bit floating-point value

Table 6.7. Casting/Conversion intrinsics.

6.7.3 Tessellation Functions

These functions can be used in a hull shader to generate corrected tessellation factors for a patch. A listing of all tessellation intrinsics can be found in Table 6.8

Intrinsic	Description
Process2DQuadTessFactorsAvg	Computes tessellation factors for a 2D quad patch using the average of the edge tessellation factors
Process2DQuadTessFactorsMax	Computes tessellation factors for a 2D quad patch using the maximum of the edge tessellation factors
Process2DQuadTessFactorsMin	Computes tessellation factors for a 2D quad patch using the minimum of the edge tessellation factors
ProcessIsolineTessFactors	Computes rounded tessellation factors for an isoline
ProcessQuadTessFactorsAvg	Computes tessellation factors for a quad patch using the average of the edge tessellation factors
ProcessQuadTessFactorsMax	Computes tessellation factors for a quad patch using the maximum of the edge tessellation factors
ProcessQuadTessFactorsMin	Computes tessellation factors for a quad patch using the minimum of the edge tessellation factors
ProcessTriTessFactorsAvg	Computes tessellation factors for a triangle patch using the average of the edge tessellation factors
ProcessTriTessFactorsMax	Computes tessellation factors for a triangle patch using the maximum of the edge tessellation factors
ProcessTriTessFactorsMin	Computes tessellation factors for a triangle patch using the minimum of the edge tessellation factors

Table 6.8. Tessellation intrinsics.

6.7.4 Pixel Shader Functions

Several intrinsics are specific to pixel shader programs. These intrinsics are listed in Table 6.9.

Intrinsic	Description
clip	Discards the pixel shader result if the specified value is less than 0. Can be used to implement clipping planes.
ddx	Returns the partial derivative of a value with respect to the screen space X-coordinate of the pixel being shaded.

ddx_coarse	Returns a low precision partial derivative of a value with respect to the screen space X-coordinate of the pixel being shaded.
ddx_fine	Returns a high precision partial derivative of a value with respect to the screen space X-coordinate of the pixel being shaded.
ddy	Returns the partial derivative of a value with respect to the screen space Y-coordinate of the pixel being shaded.
ddy_coarse	Returns a low precision partial derivative of a value with respect to the screen space Y-coordinate of the pixel being shaded.
ddy_fine	Returns a high-precision partial derivative of a value with respect to the screen space Y-coordinate of the pixel being shaded.
fwidth	Returns the absolute value of the sum of ddx and ddy for a value.
EvaluateAttributeAtCentroid	Interpolates a pixel shader input using the centroid of all covered sample points, as if the attribute were marked with the centroid modifier.
EvaluateAttributeAtSample	Interpolates a pixel shader input using the sample point indicated by the specified index, as if the attribute were marked with the sample modifier.
EvaluateAttributeSnapped	Interpolates a pixel shader input using the centroid position, with an offset.
GetRenderTargetSampleCount	Retrieves the number of samples in the current render targets.
GetRenderTargetSamplePosition	Retrieves the XY position of a sample point for the given sample index.

Table 6.9. Pixel shader intrinsics.

6.7.5 Synchronization Functions

The synchronization intrinsics listed in Table 6.10 are only available for compute shaders, and are generally used to synchronize access to shared memory or resource objects. These functions and their uses are described in detail in Chapter 5.

Intrinsic	Description
AllMemoryBarrier	Ensures that all pending memory accesses have completed for all threads in the thread group

AllMemoryBarrierWithGroupSync	Ensures that all pending memory accesses have completed for all threads in the thread group, and blocks until all threads reach the sync instruction
DeviceMemoryBarrier	Ensures that all pending device memory accesses (reads and writes to texture and buffer resources) have completed for all threads in the thread group
DeviceMemoryBarrierWithGroupSync	Ensures that all pending device memory accesses (reads and writes to texture and buffer resources) have completed for all threads in the thread group, and blocks until all threads reach the sync instruction
GroupMemoryBarrier	Ensures that all pending shared memory accesses (reads and writes to groupshared variables) have completed for all threads in the thread group
GroupMemoryBarrierWithGroupSync	Ensures that all pending shared memory accesses (reads and writes to groupshared variables) have completed for all threads in the thread group, and blocks until all threads reach the sync instruction

Table 6.10. Synchronization intrinsics.

6.7.6 Atomic Functions

These intrinsics, listed in Table 6.11, perform a guaranteed atomic operation on a local variable, a shared memory variable, or a resource variable. They can only be used on int or uint variables, and only in a pixel or compute shader. Most atomic intrinsics can optionally return the original value of the variable, at the expense of additional runtime cost.

Intrinsic	Description
InterlockedAdd	Atomic add operation, optionally returning the previous value
InterlockedAnd	Atomic AND operation
InterlockedCompareExchange	Compares the variable to a comparison value, and exchanges it with another value
InterlockedCompareStore	Compares the variable to a comparison value
InterlockedExchange	Exchanges a variable with another value
InterlockedMax	Atomic max operation
InterlockedMin	Atomic min operation
InterlockedOr	Atomic OR operation
InterlockedXor	Atomic XOR operation

Table 6.11. Atomic intrinsics.

6.7.7 Debugging Functions

HLSL debugging instrinsics, listed in Table 6.12, allow shader programs to output debugging messages to the information queue.

Intrinsic	Description
`abort`	Outputs a debug message, and aborts the current draw or dispatch
`errorf`	Outputs an error message to the information queue
`printf`	Outputs a debug message to the information queue

Table 6.12. Debugging intrinsics.

6.7.8 Format Conversion Functions

The DirectX SDK includes a file named D3DX_DXGIFormatConvert.inl, which contains a variety of inline format conversion functions. These functions are designed to be used in a pixel shader or compute shader to convert from the raw integer representation of a DXGI format to a vector floating point or integer format that can be used in standard shader arithmetic. For normal read-only textures, this conversion is normally done in the hardware's texture units as part of the Sample or Load operation. However, for byte-address or structured buffer resources accessed through unordered access views, the hardware texture units are not used, and thus the conversion needs to be performed manually in the shader. These inline functions spare the programmer from needing to write these conversion functions manually. For a full listing of all available conversion functions, consult the SDK documentation.

6.8 Shader Reflection

Direct3D 11 provides a rich, full-featured set of APIs for programmatically querying detailed information about a compiled shader program. Effective use of these APIs can enable applications to develop sophisticated content pipelines that determine the requirements of shader programs and prepare runtime data in a custom format. Or alternatively, it can allow applications to dynamically set up an appropriate runtime environment for a generic shader program.

6.8.1 Shader Program Information

General information and statistics regarding a single shader program can be queried through the ID3D11ShaderReflection interface. This interface also provides the means for accessing the interfaces used for querying constant buffer, variable, or type information. To obtain the ID3D11ShaderReflection interface for a compiled shader program, an application must call the D3D11Reflect function exported by the D3DCompiler DLL. Note that this function is actually just a wrapper for D3DReflect, which it simply calls with the interface ID of ID3D11ShaderReflection. Listing 6.23 demonstrates a simple case of compiling a vertex shader program, and obtaining its reflection interface.

```
ID3D10Blob* compiledShader;
ID3D10Blob* errorMessages;
HRESULT hr = D3DXllCompileFromFile( filePath, NULL, NULL, "VSMain",
                                    "vs_5_0", 0, 0, NULL, &compiledShader,
                                    &errorMessages, NULL );

ID3D11ShaderReflection* reflection = NULL;
if ( SUCCEEDED( hr ) )
    D3DllReflect( compiledShader->GetBufferPointer(),
                  compiledShader->GetBufferSize(),
                  &reflection );
```

Listing 6.23. Obtaining the reflection interface for a shader program.

Once the reflection interface is obtained, it can be queried for information by calling its various methods. These include methods for retrieving the thread group size, the minimum device feature level required to run the shader, and the frequency of execution for a pixel shader to name just a few. See the SDK documentation for a full list of available methods.

Most of the information provided by the ID3D11ShaderReflection interface is available by calling the GetDesc method. This method returns a D3D11_SHADER_DESC structure containing a wealth of information. Listing 6.24 contains the declaration of this structure.

```
struct D3D11_SHADER_DESC {
    UINT                                Version;
    LPCSTR                              Creator;
    UINT                                Flags;
    UINT                                ConstantBuffers;
    UINT                                BoundResources;
    UINT                                InputParameters;
    UINT                                OutputParameters;
    UINT                                InstructionCount;
```

```
    UINT                                TempRegisterCount;
    UINT                                TempArrayCount;
    UINT                                DefCount;
    UINT                                DclCount;
    UINT                                TextureNormalInstructions;
    UINT                                TextureLoadInstructions;
    UINT                                TextureCompInstructions;
    UINT                                TextureBiasInstructions;
    UINT                                TextureGradientInstructions;
    UINT                                FloatInstructionCount;
    UINT                                IntInstructionCount;
    UINT                                UintInstructionCount;
    UINT                                StaticFlowControlCount;
    UINT                                DynamicFlowControlCount;
    UINT                                MacroInstructionCount;
    UINT                                ArrayInstructionCount;
    UINT                                CutInstructionCount;
    UINT                                EmitInstructionCount;
    D3D10_PRIMITIVE_TOPOLOGY            GSOutputTopology;
    UINT                                GSMaxOutputVertexCount;
    D3D11_PRIMITIVE                     InputPrimitive;
    UINT                                PatchConstantParameters;
    UINT                                cGSInstanceCount;
    UINT                                cControlPoints;
    D3D11_TESSELLATOR_OUTPUT_PRIMITIVE  HSOutputPrimitive;
    D3D11_TESSELLATOR_PARTITIONING      HSPartitioning;
    D3D11_TESSELLATOR_DOMAIN            TessellatorDomain;
    UINT                                cBarrierInstructions;
    UINT                                cInterlockedInstructions;
    UINT                                cTextureStoreInstructions;
}
```

Listing 6.24. The D3D11_SHADER_DESC structure.

6.8.2 Constant Buffer Information

As previously mentioned, the ID3D11ShaderReflection interface can be used to obtain an ID3D11ShaderReflectionConstantBuffer interface containing information for a constant buffer used in a shader program. This is done by calling either GetConstantBufferByName, or GetConstantBufferByIndex. The former accepts a string parameter, and is useful if an application has prior knowledge of a shader program and simply needs to query a few specific details. The latter is more useful when dealing with an unknown shader program, as it allows enumeration of all constant buffers. The number of constant buffers in a program is provided by the D3D11_SHADER_DESC structure returned by GetDesc. Listing 6.25 demonstrates the common pattern of retrieving the number of constant buffers, and iterating through the ID3D11ShaderReflectionConstantBuffer interfaces.

```
D3D11_SHADER_DESC shaderDesc;
shaderReflection->GetDesc( &shaderDesc );
const UINT numCBuffers = shaderDesc.ConstantBuffers;
for ( UINT i = 0; i < numCBuffers; i++ )
{
    ID3DllShaderReflectionConstantBuffer* cbReflection = NULL;
    cbReflection = shaderReflection->GetConstantBufferByIndex( i );

    // Query constant buffer interface for information
}
```

Listing 6.25. Retrieving constant buffer reflection interfaces.

Like the ID3D11ShaderReflection interface, the ID3D11ShaderReflectionConstant Buffer interface provides a GetDesc method that returns a structure containing general information. In this case, GetDesc returns a D3D11_SHADER_BUFFER_DESC structure, whose declaration is shown in Listing 6.26.

```
struct D3D11_SHADER_BUFFER_DESC {
    LPCSTR             Name;
    D3D11_CBUFFER_TYPE Type;
    UINT               Variables;
    UINT               Size;
    UINT               uFlags;
}
```

Listing 6.26. The D3D11_SHADER_BUFFER_DESC structure.

The main members of interest in the D3D11_SHADER_BUFFER_DESC structure are Size and Variables. The Size member indicates the total size of all constants in the constant buffer, and can be used to initialize an ID3D11Buffer containing the actual constant data bound to the pipeline at runtime. The Variables member indicates the number of constants within the constant buffer, and can be used in conjunction with GetVariableBylndex to enumerate all individual constants within the constant buffer.

6.8.3 Variable Information

Information for individual constants in a constant buffer is available through the ID3D11 ShaderReflectionVariable interface, which can be obtained by calling the Get VariableBylndex or GetVariableByName methods on an ID3DllShaderReflection ConstantBuffer interface. It can also be obtained for any global variable in a shader

program by calling GetVariableByName on an ID3D11ShaderReflectionInterface. Global variables include not just constants, but also resource objects, samplers, and interface instances. The GetDesc method of the ID3D11ShaderReflectionVariable interface returns a D3D11_SHADER_VARIABLE_DESC structure, which contains general information about the variable. The declaration of this structure is listed in Listing 6.27.

```
Struct D3D11_SHADER_VARIABLE_DESC {
    LPCSTR Name;
    UINT   StartOffset;
    UINT   Size;
    UINT   uFlags;
    LPVOID DefaultValue;
}
```

Listing 6.27. The D3D11_SHADER_VARIABLE_DESC structure.

The Name member contains a string indicating the name of the variable in the HLSL source code. This value is useful for creating a system of allowing constant values to be set at runtime via by a string containing the name of the value. StartOffset and Size contain the offset in bytes from the start of the constant buffer, and the size of the value in bytes. These can be used to memcpy a value intended for specific constant into the correct portion of a mapped buffer. The DefaultValue member contains the value that a constant was initialized to in the shader source code, if one was provided. It is typically used to initialize the contents of a constant buffer when it is first created.

If the ID3D11ShaderReflectionVariable represents an interface instance, the GetInterfaceSlot method can be called to query the interface slot for that variable. This value indicates the array index that should be used for this particular interface when passing the array of class instances to a *SSetShader method.

6.8.4 Type Information

Type information for a variable can be queried by obtaining an ID3D11ShaderReflection Type interface, which is obtained by calling GetType on the ID3D11Shader ReflectionVariable interface for a variable. Calling GetDesc on this interface returns a D3D11_SHADER_TYPE_DESC structure, which contains general information regarding the type of the variable. The declaration of this structure is available in Listing 6.28.

```
Struct D3D11_SHADER_TYPE_DESC {
    D3D1B_SHADER_VARIABLE_CLASS  Class;
    D3D10_SHADER_VARIABLE_TYPE   Type;
    UINT                         Rows;
    UINT                         Columns;
    UINT                         Elements;
    UINT                         Members;
    UINT                         Offset;
}
```

Listing 6.28. The D3D11_SHADER_TYPE_DESC structure.

The Class member of the D3D11_SHADER_TYPE_DESC structure indicates whether the variable is a scalar, a vector, a matrix, a resource object, a structure, a class, or an interface pointer. The Type member indicates the primitive type of a scalar, vector, or matrix variable (float, uint, double, etc.). For resource objects it indicates the resource type, such as Texture2D, Buffer, AppendStructuredBuffer, or others. The Rows and Columns members indicates the number of rows and columns for a matrix variable, or are set to 1 for other numeric types. Elements contains the number of elements in array types, while Members contains the number of members for structure or class types. Offset contains the number of bytes from a parent structure.

The ID3DllShaderReflectionType interface also contains a variety of methods that can be used to determine the full inheritance tree for interface or class types. These include a method for querying the base class type, a method for querying all supported interfaces, and a method for querying whether a class implements an interface. The interface also contains GetMemberTypeByName and GetMemberTypeBylndex methods for obtaining the ID3DllShaderReflectionType interfaces for all members in a structure or class.

6.8.5 Input/Output Signature

The ID3DllShaderReflection interface provides a means of querying the full input and output signatures for a shader program. Such information can be important when matching a shader program from one pipeline stage to a shader program for another stage, or when matching a vertex buffer to a vertex shader. Obtaining the input signature is done by calling GetInputParameterDesc for each input parameter (the number of input and output parameters is available as part of the D3D11_SHADER_DESC structure), and calling GetOutputParameterDesc for each output parameter. Both methods return a D3D11_SIGNATURE_PARAMETER_DESC structure, which contains information about the parameter. The declaration for this structure is available in Listing 6.29.

```
Struct D3D11_SIGNATURE_PARAMETER_DESC {
    LPCSTR                          SemanticName;
    UINT                            SemanticIndex;
    UINT                            Register;
    D3D10_NAME                      SystemValueType;
    D3D10_REGISTER_COMPONENT_TYPE   ComponentType;
    BYTE                            Mask;
    BYTE                            ReadWriteMask;
    UINT                            Stream;
}
```

Listing 6.29. The D3D11_SIGNATURE_PARAMETER_DESC structure.

The SemanticName and SemanticIndex members indicate which semantic is bound to the parameter, whether a user-defined semantic or a system-value semantic. The Register indicates which register (*v0* through *v31)* the parameter was mapped to by the compiler. For parameters mapped to a system-value semantic, the SystemValueType member will contain a value indicating which semantic was used. ComponentType indicates whether the parameter is a float, int, or uint, while mask indicates which components of the register are used to store the parameter value. For input parameters the ReadWriteMask member indicates which components are read by the shader program, while for output parameters it indicates which components were written. Finally, the Stream member indicates which stream a geometry shader is using for the parameter.

6.8.6 Resource Bindings

While the ID3D11ShaderReflectionVariable and ID3D11ShaderReflectionType interfaces can be used to obtain information about the resources required by a shader program, they are not very convenient for determining how resources need to be bound to the pipeline to provide a suitable execution environment for a shader. In particular, they don't provide one key piece of information, which is the slot index that a resource is bound to. Instead, this information can be obtained by calling GetResourceBindingDesc or GetResourceBindingDescByName on an ID3DllShaderReflection interface. These methods both return a D3D11_SHADER_INPUT_BIND_DESC structure, whose declaration is in Listing 6.30.

```
struct D3D11_SHADER_INPUT_BIND_DESC {
    LPCSTR                      Name;
    D3D10_SHADER_INPUT_TYPE     Type;
    UINT                        BindPoint;
    UINT                        BindCount;
```

```
    UINT                        uFlags;
    D3D11_RESOURCE_RETURN_TYPE  ReturnType;
    D3D10_SRV_DIMENSION         Dimension;
    UINT                        NumSamples;
}
```

Listing 6.30. The D3D11_SHADER_INPUT_BIND_DESC structure.

Basic pieces of information about the resource binding, such as the name and type of the resource, are available through the Name and Type members Respectively. BindPoint indicates the slot index for singular resources, while for arrays of resources, it indicates the start slot. BindCount indicates the number of resources in resource arrays. The uFlags member can be used to determine whether a resource was manually mapped to its slot through the `register` statement, and whether a sampler is a comparison sampler. ReturnType indicates the return type that was specified as the template parameter of a resource object, while Dimension indicates the required type of shader resource view that can be bound for the resource. Finally, NumSamples indicates the number of samples specified as the template parameter of a Texture2DMS or Texture2DMSArray resource object.

6.9 Usingfxc.exe

The DirectX SDK includes a command-line utility named `fxc.exe`, which can compile shaders and programs and effects using the same compiler provided by the D3DCompiler DLL. While it can be used to precompile a shader program to bytecode and save it to a file for quick loading, it can also produce some helpful diagnostic information. The simplest way to obtain this information is to run f xc and only specify the shader profile, the entry point, and the file name. This will cause the resulting information to be output to the command line. Listing 6.31 shows the output from fxc when compiling a pixel shader from the LightPrepass sample project.

```
Microsoft (R) Direct3D Shader Compiler 9.29.952.3111
Copyright (C) Microsoft Corporation 2002-2009. All rights reserved.
//
// Generated by Microsoft (R) HLSL Shader Compiler 9.29.952.3111
//
//
//   fxc /T ps_5_0 /E PSMainPerSample LightsLP.hlsl
//
//
// Buffer Definitions:
//
```

```
// cbuffer CameraParams
// {
//
//   float4x4 ViewMatrix;              // Offset:    0 Size:    64 [unused]
//   float4x4 ProjMatrix;              // Offset:   64 Size:    64
//   float4x4 InvProjMatrix;           // Offset:  128 Size:    64 [unused]
//   float2   ClipPlanes;              // Offset:  192 Size:     8 [unused]
//
// }
//
//
// Resource Bindings:
//
// Name                                 Type  Format         Dim Slot Elements
// ------------------------------ ---------- ------- ----------- ---- --------
// GBufferTexture                    texture  fiout4        2dMS    0        1
// DepthTexture                      texture   float        2dMS    1        1
// CameraParams                      cbuffer      NA          NA    0        1
//
//
//
// Input signature:
//
// Name                 Index   Mask Register SysValue  Format   Used
// -------------------- -----  ------ -------- -------  ------ ------
// SV_Position              0   xyzw        0      POS   float   xy
// VIEWRAY                  0   xyz         1     NONE   float   xyz
// RANGE                    0      w        1     NONE   float       W
// POSITION                 0   xyz         2     NONE   float   xyz
// COLOR                    0   xyz         3     NONE   float   xyz
// SV_SampleIndex           0   x           4   SAMPLE    uint   x
//
//
// Output signature:
//
// Name                 Index   Mask Register SysValue  Format   Used
// -------------------- -----  ------ -------- -------  ------ ------
//          SV_Target          0   xyzw        0    TARGET   float   xyzw
//
// Pixel Shader runs at sample frequency
//
ps_5_0
dcl_globalFlags refactoringAllowed
dcl_constantbuffer cb0[7], immediateIndexed
dcl_resource_texture2dms(0) (float,float,float,float) t0
dcl_resource_texture2dms(0) (float,float,float,float) t1
dcl_input_ps_siv linear noperspective v0.xy, position
dcl_input_ps linear v1.xyz
dcl_input_ps linear v1.w
dcl_input_ps linear v2.xyz
dcl_input_ps linear v3.xyz
dcl_input_ps_sgv v4.x, sampleIndex
dcl_output o0.xyzw
dcl_temps 4
ftoi r0.xy, v0.xyxx
mov r0.zw, l(0,0,0,0)
```

```
ldms_indexable(texture2dms)(float,float,float,float)  r0.z,  r0.xyzw,  t1.yzxw,  v4.x
ldms_indexable(texture2dms)(float,float,float,float)  r0.xyw,  r0.xyww,  t0.xywz,
v4.x
add r0.z,  r0.z,  -cb0[6].z
div r0.z,  cb0[6].w,  r0.z
mad r1.xyz,  -v1.xyzx,  r0.zzzz,  v2.xyzx
dp3 r1.w,  r1.xyzx,  r1.xyzx
sqrt r1.w,  r1.w
div r1.xyz,  r1.xyzx,  r1.wwww
div r1.w,  r1.w,  v1.w
add r1.w,  -r1.w,  1(1.000000)
max r1.w,  r1.w,  1(0.000000)
mad r2.xyz,  -v1.xyzx,  r0.zzzz,  r1.xyzx
dp3 r0.z,  r2.xyzx,  r2.xyzx
rsq r0.z,  r0.z
mul r2.xyz,  r0.zzzz,  r2.xyzx
mov r3.zw,  1(0,0,-1.000000,1.000000)
mov r3.xy,  r0.xyxx
dp3 r0.z,  r3.xywx,  -r3.xyzx
sqrt r2.w,  r0.z
mul r0.xy,  r2.wwww,  r3.xyxx
mad r0.xyz,  r0.xyzx,  1(2.000000,  2.000000,  2.000000,  0.000000),  1(0.000000,
0.000000,  -1.000000,  0.000000)
dp3_sat r2.x,  r0.xyzx,  r2.xyzx
dp3_sat r0.x,  r0.xyzx,  rl.xyzx
log r0.y,  r2.x
mul r0.y,  r0.y,  r0.w
add r0.z,  r0.w,  1(8.000000)
mul r0.z,  r0.z,  1(0.039789)
exp r0.y,  r0.y
mul r0.y,  r0.z,  r0.y
mul r0.y,  r0.x,  r0.y
mul r0.xzw,  r0.xxxx,  v3.xxyz
mul o0.xyzw,  rl.wwww,  r0.xzwy
ret
// Approximately 35 instruction slots used
```

Listing 6.31. Diagnostic output from fxc. exe.

The first section ("Buffer Definitions:") is a listing of all constant buffers used in the shader program. This includes the offset and size of each individual constant, as well as whether or not the constant is actually used in the shader program. The next section ("Resource Bindings:") contains the name, type, and slot of all resource objects and constant buffers used in the shader. The third section ("Input signature:") contains a listing of all inputs required by the shader program. The input signature can be very important when matching shaders for one stage with shaders from another stage, since the earlier shader must produce enough outputs to satisfy the input signature. This is also true of vertex shaders, except now the bound vertex buffer and. corresponding input layout must provide the elements required by the input signature. The final section ("Output signature:") simply

lists all values returned by the shader. It also indicates if the pixel shader runs at per-sample frequency, which is due to taking SV_SampleIndex as an input.

After the diagnostic information, the. output also contains the fully compiled shader program in assembly. While it's not typically useful to examine the generated assembly, it can be helpful for verification purposes during performance analysis. In particular, it is common to look for dynamic branching or looping constructs, since these can have a drastic effect on performance. Dynamic branches can be spotted by looking for an if_"comp" instruction, where comp is a two-letter abbreviation of a comparison. For instance, lt will be used for a less-than comparison, and ge for a greater-than-or-equal comparison. Dynamic loops will begin with a rep instruction. The output also contains an instruction count at the end, which is simply the number of assembly instructions in the program.

While it is possible to get a *very* rough estimate of the relative performance cost of a shader through this number, in general, it is not a reliable figure. This is because shader assembly is merely an intermediate format that is further compiled by the driver into a microcode format that can be executed by the hardware. Thus, the final program could have a very different number of instructions, and the number of cycles required to execute those instructions could also vary, depending on the hardware. More importantly, even the actual microcode instructions will not properly reflect the larger-scale performance characteristics caused by memory accesses and multithreaded execution. To obtain more accurate performance statistics regarding a shader program, specialized analysis and profiling tools are available from the major graphics hardware vendors.