# Direct3D 11 Shader Reflection Interface

**By Jason Zink**

**Last Modified: 6/18/2009**

## Introduction

While building a renderer that will utilize the Direct3D 11 API, one of the many challenges that must be faced is learning how to load a shader from file and have your renderer comprehend what it has loaded. The renderer must be able to tell what resources that particular shader will require to operate; otherwise it would be impossible to properly use a shader unless its resource binding is hard-coded. Thus to keep the renderer as flexible as possible it is necessary to query the shader to find out what resources it uses. This is a non-trivial task due to the increased number of shader types as well as quite a large proliferation of resource types that a shader can use.

In addition to resource usage issues, the renderer must also be able to ensure that the interface signature between shader stages match. Since the shader files can be loaded individually there is no guarantee that two subsequent shaders' input and output signatures will match. The ability to validate that a sequence of shaders are able to be linked together is a strong requirement, especially during development while the shader code base is being built up.

Together these two tasks can be resolved by utilizing one of the features of D3D11: the shader reflection interface. This document is intended to provide a quick reference to the shader reflection system. It is currently being based on the March 2009 DXSDK, which is a CTP release of D3D11. This means that the documentation of the some of the features are still incomplete. However, based on my own work with the API and the existing documentation and header files, I have drawn as much information as I could to share. This document will likely be updated as additional features are added to the documentation in future DXSDK releases.

## Shader Reflection

In order to utilize the shader reflection system, you first have to have the ability to compile a shader. There is some sample code for performing shader compilation in the DXSDK samples, but for simplicity we will briefly discuss the process. My function for performing the compilation is shown below for reference:

```
int RendererDX11::LoadShader( ShaderType type, std::string& filename,
                              std::string& function, std::string& model )
{
	HRESULT hr = S_OK;

	ID3DBlob* pCompiledShader = NULL;
	ID3DBlob* pErrorMessages = NULL;
```

```
        if ( FAILED( hr = D3DX11CompileFromFile(
                filename.c_str(),     // Shader file name
                0,
                0,
                function.c_str(),     // Shader function name
                model.c_str(),        // Shader file name
                0,                    // Shader compile flags
                0,                    // Effect compile flags
                0,
                &pCompiledShader,
                &pErrorMessages,
                &hr
                ) ) )
        {
                if ( pErrorMessages != 0 )
                {
                        LPVOID pCompileErrors = pErrorMessages->GetBufferPointer();
                        const char* pMessage = (const char*)pCompileErrors;
                        CLog::Get().Write( (const char*)pCompileErrors );
                }

                return( -1 );
        }

        // Do something with pCompiledShader…

        return( 1 );
}
```

This function utilizes the D3DX11 library function for compiling a shader. There are several more advanced features that can be utilized with this function, but this is a bare bones usage to simply compile a single file shader. To utilize this function, you need to include the D3DX11.h header file, and link to the D3DX11.lib library.

Assuming that your shader is syntactically correct and that you are compiling it against the correct shader model, then this function should produce a valid compiled shader in the `pCompiledShader` variable. But how do you know what to do with the shader now? What other shaders is it compatible with? Now we need to use the shader reflection interface to answer these questions.

## The Shader Reflection API

To begin using the shader reflection system, we first need to understand what exactly it is and how it functions. The actual class that we will be using is actually a COM interface named `ID3D11ShaderReflection`. This interface is obtained through a call to the `D3DReflect` function. The function itself takes pointers to the compiled shader code, the size of the compiled shader code, the interface identifier for the `ID3D11ShaderReflection` interface, and then the output interface pointer. From our previous code listing, the reflection interface could be obtained as follows:

```
ID3D11ShaderReflection* pReflector = NULL;
hr = D3DReflect( pCompiledShader->GetBufferPointer(),
                    pCompiledShader->GetBufferSize(), IID_ID3D11ShaderReflection,
                        (void**) &pReflector);
```

If the function call is successful (you will know by checking the HRESULT) then you will have a pointer to the shader reflection interface for this particular compiled shader. One point to note that was surprising to me was that the type of the shader does is not an input into this system – the resulting interface is independent of the type of shader that you are compiling. This is actually a good thing, since you don't need to perform any special case code depending on the shader type.

Once you have a valid interface, you are ready to perform some inspection of the shader details. So what exactly can you do with it? Let's look at the various member functions for this interface:

| Method Name | Description |
|---|---|
| GetBitwiseInstructionCount | Gets the number of bitwise instructions. |
| GetConstantBufferByIndex | Gets a constant buffer by index. |
| GetConstantBufferByName | Gets a constant buffer by name. |
| GetConversionInstructionCount | Gets the number of conversion instructions. |
| GetDesc | Gets a shader description. |
| GetGSInputPrimitive | Gets the geometry-shader input-primitive description. |
| GetInputParameterDesc | Gets an input-parameter description for a shader. |
| GetMinFeatureLevel | TBD |
| GetMovInstructionCount | Gets the number of Mov instructions. |
| GetMovcInstructionCount | Gets the number of Movc instructions. |
| GetNumInterfaceSlots | TBD |
| GetOutputParameterDesc | Gets an output-parameter description for a shader. |
| GetPatchConstantParameterDesc | Gets a patch-constant parameter description for a shader. |
| GetResourceBindingDesc | Gets a description of the resources bound to a shader. |
| GetResourceBindingDescByName | Gets a description of the resources bound to a shader. |
| GetVariableByName | Gets a variable by name. |
| IsSampleFrequencyShader | TBD |

As you can see, there are several functions that don't even have a description yet, but for the majority of them you can figure out what they do based on their name. As the documentation is updated I will add any relevant descriptions.

The functions are generally separated into roughly two groups: shader code information, and shader resource information. With respect to the code information, you can see several functions to determine how many instructions of a certain type there are. While these functions are interesting and could potentially be used to gauge a shader's performance potential, for this article we will be focusing on the resource information functions.

To get started, we will utilize the `ID3D11ShaderReflection->GetDesc` method to obtain a `D3D11_SHADER_DESC` structure. This structure provides an overall description of the shader that we will use throughout the remainder of the article. The code to perform this lookup is shown here:

```
D3D11_SHADER_DESC desc;
pReflector->GetDesc( &desc );
```

The members of the description structure are as follows:

```
typedef 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;
    D3D10_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;
} D3D11_SHADER_DESC;
```

That is a whole bunch of information about a shader that you may or may not be interested in for any given situation. We'll show two possible uses in each of the next two sections.

## Shader Input and Output Parameter Descriptions

The first two functions that we will look at are used to determine the input and output signatures for the shader. Specifically, they are `GetInputParameterDesc` and `GetOutputParameterDesc`. Both of these functions are used to obtain a D3D11_SIGNATURE_PARAMETER_DESC structure – one function for obtaining the input signatures and one for obtaining the output signatures of the shader. We use two members of the `D3D11_SHADER_DESC` structure to see how many input and output parameters there are – the `InputParameters` and `OutputParameters` members.

With the number of parameters for both input and output, we use the `GetInputParameterDesc` and `GetOutputParameterDesc` functions to build the overall signature. The code to obtain these signatures is shown here:

```
for ( UINT i = 0; i < desc.InputParameters; i++ )
{
	D3D11_SIGNATURE_PARAMETER_DESC input_desc;
	pReflector->GetInputParameterDesc( i, &input_desc );
	// Do something with the description…
}
for ( UINT i = 0; i < desc.OutputParameters; i++ )
{
	D3D11_SIGNATURE_PARAMETER_DESC output_desc;
	pReflector->GetInputParameterDesc( i, &output_desc );
	// Do something with the description…
}
```

The individual components of the structure are shown here for reference:

```
typedef struct D3D11_SIGNATURE_PARAMETER_DESC {
    LPCSTR SemanticName;
    UINT SemanticIndex;
    UINT Register;
    D3D11_NAME SystemValueType;
    D3D11_REGISTER_COMPONENT_TYPE ComponentType;
    BYTE Mask;
    BYTE ReadWriteMask;
} D3D11_SIGNATURE_PARAMETER_DESC;
```

Once again, these members are fairly self explanatory. The mask member is an indication of which components of the input register are used, while the ReadWriteMask indicate if a particular input is always read or a particular output is never written. This information could presumably be used in trying to combine variables into a single vector parameter or eliminating unused components. In any case, since the input and output signatures use the same structure, it is relatively simple to perform a comparison of one shader's output with the next shader's input signatures to see if they are compatible.

## Shader Resource Descriptions

Once you know if a shader is compatible for use in your pipeline configuration, the next thing that you need to do is to find out what resources the shader will need. Unlike the input/output signatures, obtaining the information about shader resource utilization is not quite as clear cut. However, we'll take a look at it step by step and see what needs to be done.

To begin, we will look at the `ConstantBuffers` member of the D3D11_SHADER_DESC. This is the number of 'Constant Buffers' that the shader utilizes. This name was somewhat surprising to me as well, due to the fact that this constant buffer count is actually the total number of resources that are used in the shader – including CBUFFERs (which hold constant parameters like vectors and matrices), TBUFFERs (which typically contain texture variables), as well as bound resources (which typically contain a resource view). That's alright though, now we know what to expect from the ensuing descriptions.

For each of the parameters indicated in that count, we will obtain a D3D11_SHADER_BUFFER_DESC description for it. In addition to this, we will obtain two other descriptions, D3D11_SHADER_VARIABLE_DESC and D3D11_SHADER_TYPE_DESC for each sub-parameter of the given resource. These variable and type descriptions are ways to determine exactly what resources and parameters need to be loaded into a buffer to be used by the shader.

These additional sub-parameters really only make sense for a structure type of resource, which is typically a CBUFFER. The idea here is to identify what type of overall resource you are dealing with, then to drill down to the details about the resource so that you know what to put into it. Even if the resource is not a structured resource, such as a shader resource view, then the additional sub-parameters will describe only that resource view. Here is some sample code to show how to query each of these parameters:

```
struct ConstantBufferLayout
{
	D3D11_SHADER_BUFFER_DESC Description;
	TArray<D3D11_SHADER_VARIABLE_DESC> Variables;
	TArray<D3D11_SHADER_TYPE_DESC> Types;
}

for ( UINT i = 0; i < desc.ConstantBuffers; i++ )
{
	ConstantBufferLayout BufferLayout;
	ID3D11ShaderReflectionConstantBuffer* pConstBuffer = pReflector-
>GetConstantBufferByIndex( i );
	pConstBuffer->GetDesc( &BufferLayout.Description );

	// Load the description of each variable for use later on when binding a
buffer
	for ( UINT j = 0; j < BufferLayout.Description.Variables; j++ )
	{
		// Get the variable description and store it
		ID3D11ShaderReflectionVariable* pVariable = pConstBuffer-
>GetVariableByIndex( j );
		D3D11_SHADER_VARIABLE_DESC var_desc;
		pVariable->GetDesc( &var_desc );

		BufferLayout.Variables.add( var_desc );

		// Get the variable type description and store it
		ID3D11ShaderReflectionType* pType = pVariable->GetType();
		D3D11_SHADER_TYPE_DESC type_desc;
		pType->GetDesc( &type_desc );

		BufferLayout.Types.add( type_desc );
	}

	pShaderWrapper->ConstantBuffers.add( BufferLayout );
}
```

Let's take a quick look at the three types of structures obtained in the code above. First will be the D3D11_SHADER_BUFFER_DESC structure:

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

In here, you will find the name of the resource followed by its buffer type. Next you will find the number of variables that are included in this buffer, which is later used to obtain the variable and type structures for each of those variables. After that, is the size of the buffer overall, followed by a flags parameter. There isn't much documentation on the flags parameter, so that will be updated at a later time. Next we will look at the D3D11_SHADER_VARIABLE_DESC structure:

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

Here you find the name of the parameter first, then an offset into the structure where this parameter starts (again this makes sense for something like a constant buffer, but not really for a texture). Next is the size of that parameter. There isn't much documentation on the flags parameter, so that will be updated at a later time.

And finally the D3D11_SHADER_TYPE_DESC structure:

```
typedef struct D3D11_SHADER_TYPE_DESC {
    D3D11_SHADER_VARIABLE_CLASS Class;
    D3D11_SHADER_VARIABLE_TYPE Type;
    UINT Rows;
    UINT Columns;
    UINT Elements;
    UINT Members;
    UINT Offset;
} D3D11_SHADER_TYPE_DESC;
```

So for each variable in the buffer, we also get information on the type of that parameter. The Class and Type members basically identify exactly what type of parameter this structure represents. The remaining parameters provide some ancillary information that may or may not be important to your understanding of the parameter (for example, the rows and columns parameters only really apply to the matrix type of parameters).

Overall, with these three types of structures describing the resources used by our shader we can determine at compile time what we need to supply to it for proper operation.

## Conclusion

In this article, we have looked at how to utilize the D3D11 shader reflection API to query a compiled shader for its input and output signatures, as well as how to build a detailed description of each of the resources that are used by the shader.  As additional information becomes available on the D3D11 API I will be adding additional detail to this article to provide a more comprehensive reference to this reflection system.