

Shader Series Primer: Fundamentals of the Programmable Pipeline in XNA Game Studio Express

Level: Intermediate

Area: Graphics Programming

Summary

This document is an introduction to the series of samples, tutorials, and articles known as the Shader Series. This is a serial set of educational documentation and sample code that should allow an intermediate 3D developer to begin to explore the programmable graphics pipeline.

Audience

This document will be most useful for developers with some previous experience with either the fixed-function pipeline or the **BasicEffect** type in the XNA Framework. This document assumes no previous experience writing shaders or effects.

Background

History

In 1999, Microsoft introduced an important new feature in DirectX 7 that came to be known as *hardware transformation and lighting*, or *hardware T&L*. The new technology moved the expensive vertex transformation and lighting calculations from the CPU to the GPU. The DirectX 7 API exposed an elaborate set of state values that allowed a fixed number of lights, textures, and other states to be applied to a given draw function.

While hardware texturing and lighting had an incredible impact on the quality of 3D graphics on the personal computer, there was a significant drawback. The calculations used to light and display the world were hard-wired on the GPU. As a result, games of that time began to look very similar, since they couldn't differentiate their lighting models except by applying different states. The era of universal 3D acceleration had raised the overall quality bar, but at the cost of the flexibility afforded by software rendering.

In the field of offline software image rendering for movie special effects, small programs or functions called shaders were increasingly being used to define custom interactions between a variety of materials and lighting conditions. Real-time applications were soon to follow; in 2002, the first consumer level programmable GPUs became available. Game developers were eager to make use of the new functionality. Most early consumers of programmable GPUs associated shaders with the classic rippling water effect, which at the time was considered the height of real-time graphical eye candy.

DirectX 8.0 was the first Microsoft graphics API to support programmable shaders, though initially, all shader programs had to be written in assembly code. As shader hardware increased in complexity, so did the programs. High-level shading languages were introduced to make shader development manageable. Today, Microsoft high-level shader language (HLSL) is the standard language used by all Microsoft 3D APIs, including the XNA Framework. The language compiles directly to the byte code used to execute shaders on GPUs.

Shaders have evolved greatly over the years. Generations of shader hardware are usually categorized by the DirectX shader models they support. Early shader models had extreme limits on the kinds and number of instructions that could be run on each vertex or pixel. Later models defined more instructions, added larger numbers of instructions per shader program, and enabled looping and branching functionality. Many XNA Windows materials are written with a minimum bar of Shader Model 2.0, while the Xbox 360 platform supports its own version of Shader Model 3.0. These models have the flexibility to support a huge number of rendering and optimization scenarios.

High-Level Shader Language (HLSL)

HLSL is the programming language created by Microsoft for writing shaders. It is similar to many C-style languages. The DirectX SDK is a great place to get more information about HLSL. A very complete set of documentation on HLSL can be found here:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/HLSL_Shaders.asp.

A full description of HLSL is outside the scope of this document. Familiarity with HLSL is not required for the rest of this document, but the link above is a great starting point for further reading.

Often the best way to learn something is to jump straight in. All of the upcoming example shaders are labeled and commented in such a way to make comprehension more straightforward. This document assumes no previous experience with HLSL and attempts to clarify new HLSL grammars as they come up.

Other Shader Languages

There are plenty of other high-level shader languages available, but HLSL is the only language supported intrinsically by XNA and DirectX. Other common languages are GLSL (Open GL Shading Language) and NVidia's Cg language. When non-XNA materials reference these languages, they essentially fill the role that HLSL does for DirectX.

Effects

Introduction to Vertex Shaders

Vertex shaders expose functionality that was originally hard-coded into fixed-function hardware texture and lighting. Vertex shader programs are functions run once on each vertex passed into a **Draw** call. They are responsible for transforming raw, unlit vertices into processed vertex data usable by the rest of the graphics pipeline. The input of the vertex shader corresponds to the untransformed data in the vertex buffer.

At the bare minimum, a vertex shader only needs to return a transformed position.

Introduction to Pixel Shaders

Pixel shaders add a level of control not available in classic fixed-function pipelines. To understand a pixel shader, you have to know a bit about what happens after the vertex shader runs. The processed vertex data is used to set up triangles, which in turn are used to determine which

pixels on the screen will be drawn. The input to the pixel shader is calculated using the vertex shader outputs from each of the triangle's three vertices.

The inputs of a pixel shader function are therefore bound to the outputs of the vertex shader. So if a vertex shader returns color data, the pixel shader inputs may include this color data. The data is typically interpolated using the three surrounding vertices. For example, imagine an equilateral triangle. Each of its three vertices is a different color. One is red, one is green, and one is blue. The color input to the pixel shader will be calculated by linear interpolation on those three colors. Pixels that are close to the red vertex will be mostly red, pixels that are closer to the blue vertex will be blue, and the pixel in the exact center of the triangle will have equal parts of red, green, and blue.

At a minimum, pixel shaders must output color data. Pixel shader outputs translate directly into the colors seen on the screen. Pixel shaders are primarily used for a number of per-pixel color operations, including texturing, lighting, and image processing.

Introduction to Effects

Effects combine the ideas of vertex shaders, pixel shaders, and graphics device states into one common file format. XNA supports shaders primarily through effects, so they are central to the idea of writing your own shader programs. An effect file is a text file that contains the HLSL code used by any number of vertex and pixel shaders.

An effect contains one or more techniques, which are in turn made up of at least one pass. Each pass usually contains one vertex shader function, one pixel shader function, and any number of render state and sampler state settings. In the next section, we'll look at a simple example effect and go over each line in detail.

Examining a Simple Effect

```
float4x4 mWorldViewProj; // World * View * Projection transformation

float4 Vertex_Shader_Transform(
    in float4 vPosition : POSITION ) : POSITION
{
    float4 TransformedPosition;

    // Transform the vertex into projection space.
    TransformedPosition = mul( vPosition, mWorldViewProj );

    return TransformedPosition;
}

float4 Pixel_Shader() : COLOR0
{
    return float4(1,1,1,1);
}

technique ColorShaded
{
    pass P0
    {
        VertexShader = compile vs_1_1 Vertex_Shader_Transform();
        PixelShader = compile ps_1_4 Pixel_Shader();
    }
}
```

This effect is one of the simplest effects that will produce a useable render. This shader will take a world-view-projection matrix and render white geometry based on its vertex positions. We'll now break this shader down and explain each part in detail.

HLSL Semantics

In the code listing above, a syntax that may be somewhat unfamiliar are the capitalized keywords that follow a variable and a colon. Consider the following line of HLSL code:

```
in float4 vPosition : POSITION
```

The **POSITION** keyword is called a *semantic*, which has an important place in HLSL code. These keywords indicate to the shader compiler how to map the inputs and outputs of the graphics data to the shader variables. In this example, vertex position data is being mapped to an argument called *vPosition*. This informs the shader that the *vPosition* argument will contain position data from the vertex buffer.

This document will explain the usage of semantics as they come up in the effect code.

HLSL Types

```
float4 TransformedPosition;
```

One aspect of HLSL programming that will quickly become intuitive is the different primitive types available when initializing variables. In this case, a **float4x4** primitive is used, indicating a matrix of four floats by four floats. A **Vector3**, which is a structure of three floats, is a **float3** in HLSL. HLSL defines a number of primitive types, and a robust documentation can be found here: <http://msdn2.microsoft.com/en-us/library/bb206325.aspx>.

Also provided here is a table mapping some basic HLSL types to their .NET or XNA Framework equivalents.

HLSL Type	XNA or .NET Framework Type
Float	Float
float2	Vector2
float3	Vector3
float4	Vector4, Quaternion, Color
float4x4	Matrix
Int	Int32

Effect Parameters

Effect parameters are the uniform data that remains constant for every vertex or pixel processed by the **Draw** call. These can be initialized in the effect, though many times it's only appropriate to set these values in the render loop. Effect constants are used to represent a variety of things, but most commonly they represent transformation data, light settings, and material information.

```
float4x4 mWorldViewProj; // World * View * Projection transformation
```

Only one constant has been specified in the example effect. In this case, it's the world-view-projection matrix used to transform the vertices drawn from object space into clip space.

By itself, this uninitialized parameter isn't all that helpful. The application must provide this data. The XNA Framework API facilitates this assignment using the **EffectParameter** type, which is used to get or set the value of the parameter in the effect. The following condensed example shows how one might set the above matrix in C# code.

```
//Initialize the parameter
Effect exampleEffect = content.Load<Effect>("ExampleEffect");
EffectParameter worldViewProjParameter =
    exampleEffect.Parameters["mWorldViewProj"];

Matrix worldViewProj = Matrix.Identity * //world transform
    Matrix.CreateLookAt( //view transform
        new Vector3(0f, 0f, -10f),
        Vector3.Zero,
        Vector3.Up) *
    Matrix.CreatePerspectiveFieldOfView( //projection transform
        MathHelper.PiOver4,
        1.333f,
        0.1f,
        100f);

//Set the world-view-projection matrix
worldViewProjParameter.SetValue(worldViewProj);
```

Uniform and Varying Inputs

The data that makes shaders function comes in two flavors: *varying* and *uniform*. Varying data is unique to each execution of the shader function. In the case of vertex shaders, it's the data that comes from the vertex buffer. For pixel shaders, it is the data specific to the individual pixel being rendered.

The other type of data is uniform, and it includes data that applies across the entire draw call. This is also referred to as constant data, and is treated differently. The developer can set the values of any of the shader constants through the Effect API. In the previous example, one of the constants was a **float4x4** (a 4x4 matrix of floating-point values) called *mWorldViewProj*. In the XNA Framework API, the developer can look up the **wvp** field by name and set it to a matrix available in the application. In this example, the matrix being set is the word-view-projection matrix information required by nearly every basic vertex shader.

Vertex Shader Function

Vertex shaders take a variety of inputs, and the values of these inputs vary for each vertex rendered. Usually, there's a one-to-one correspondence between a vertex shader's inputs and the structure of the vertices in the supplied vertex buffer.

```
float4 Vertex Shader Transform(
    in float4 vPosition : POSITION ) : POSITION
```

In the provided example shader, the vertex shader takes a single input: the untransformed vertex position. The way that the shader informs Direct3D what the purpose of each variable is through semantics. In this case, the **POSITION** semantic is applied to *vPosition*, meaning that *vPosition* will correspond to the x, y, and z-coordinates of a vertex.

There is a second **POSITION** semantic declared after the function declaration. This semantic applies to the **float4** return value of the vertex shader function. This is an output semantic that informs the effect compiler that the return value is a transformed position.

Next, the body of the vertex shader function will be examined, starting with the first line:

```
float4 TransformedPosition;
```

Here, we're initializing a variable that will hold the results of the vertex shader. This is a structure of the type **float4**. The syntax for declaring a local variable is similar to variable initialization in C# or other C-style languages.

Transformation Pipeline

In the fixed-function pipeline, the actual transformation function was hidden from the developer. In the programmable pipeline, shader flexibility is contingent on allowing the developer to apply transforms as needed.

In this example, the vertex shader is responsible for transforming the incoming vertex data. This requires a calculation in the shader that multiplies a position vector by a world-view-projection matrix.

```
// Transform the vertex into projection space.  
TransformedPosition = mul( vPosition, mWorldViewProj );
```

This calculation transforms the vertex position from model space to projection space. These transformed positions are used by the geometry processing portion of the Direct3D pipeline to define the triangles that make up primitives on the screen. This is a matter of a simple multiply (the **mul** function in HLSL). That function in the shader is identical to calling **Vector4.Transform(vPosition, mWorldViewProj)** in the XNA Framework.

For more information about world-view-projection transforms, see the [Coordinate Spaces](#) article.

Returning Data from the Vertex Shader

The last part of the vertex shader function simply returns the output from the shader. Like C++ and C#, the **return** keyword is used to return the vertex shader outputs.

```
return TransformedPosition;
```

The Pixel Shader Function

```
float4 Pixel_Shader( ) : COLOR0
```

The first thing to note is that the pixel shader is returning a **float4** value. This value represents the color of the pixel after the draw call. A pixel shader's primary function is to return the color of the current pixel. Like the vertex shader, a semantic (**COLOR0**) is defined for the return value of the function.

Most simple pixel shader functions will only ever return an RGBA color. In most shaders, color values are represented by floating-point vectors with 0.0 being completely dark and 1.0 as the maximum or fully-on-state. The graphics hardware then translates this value into a color that is meaningful in the context of the current back-buffer format.

```
return float4(1,1,1,1);
```

There's not much to this pixel shader, since the vertex shader has done all the hard. The pixel shader simply returns a white color. This means that all of the triangles drawn with this shader will appear flat-white.

State Setting

The last part of the effect is used to set state on the **GraphicsDevice**. It tells the device how the shader functions should be used.

```
technique ColorShaded
{
    pass P0
    {
        VertexShader = compile vs_1_0 Vertex_Shader_Transform();
        PixelShader   = compile ps_1_4 Pixel_Shader();
    }
}
```

This section informs the effect of which shaders to apply using a given technique or pass. An effect file may contain several techniques. However, for this example, the effect is limited to a single technique. Passes are included to allow multiple-pass renders, which are common in more complex shaders. In this example, `P0` refers to the name of the pass.

There are only two states being set in this technique . the pixel shader and the vertex shader. The compile command also indicates what shader model to which to compile the shader. For now, it's best not to get bogged down by shader model specifics. The samples all use appropriate shader models for the techniques being employed.

There are lots of other states that can be set in the technique. For example, nearly all of the render states and sampler states available on the **GraphicsDevice** can be set in the technique code. This is a useful way to organize specific states with their accompanying shaders. Later samples in the Shader Series will cover more of these states in depth.

The Shader Series Samples

The next step is to try out the Vertex Lighting sample to begin using working examples of shader code. You are encouraged to refer back to this guide if you run into conceptual problems in the samples. As you progress, more and more of the core Programmable Pipe Reference Pages documentation will start making sense, and you'll find yourself using the reference documents quite often to learn the features available in shader programs.