Shaders and the SimpleShader Library

What are Shaders?

Shaders are small programs that can be executed by a GPU, generally as part of a rendering pipeline. This is what makes a rendering pipeline "programmable". Prior to programmable pipelines, the work these stages performed was embedded onto graphics cards and could be customized by changing various graphics state. Now that we can place our own programs into the pipeline, the possibilities for exactly what these stages do are endless. However, there are some specific tasks that are generally required for each stage.

When programmable shaders were first introduced, they had to be written in pure assembly code to match the exact instructions available on GPUs. These days, we can write in a C-like higher-level *shading language* and pass that code to a *shader compiler,* which outputs GPU assembly code. Direct3D uses the **High-Level Shading Language**, or **HLSL**, as its shading language. While the syntax of this language differs from OpenGL's Shading Language (GLSL), both languages compile to essentially the same underlying assembly code since both can run on any modern GPU.

Since we're using Direct3D and Visual Studio, any shaders that are part of our Visual Studio project will be compiled at build-time (when we compile our overall program). We can then load these pre-compiled shaders from files and feed them to our Graphics API, though the ability to compile shaders at run-time is certainly possible. Visual Studio's HLSL editor will do basic syntax highlighting and show compiler errors, but for a more robust HLSL writing experience, I highly recommend the HLSL Tools extension for Visual Studio.

HLSL

HLSL, developed by Microsoft, is Direct3D's shading language. It was developed alongside Nvidia's Cg shading language and, as such, the two languages are almost identical. There are many versions (or *Shader Models*) of HLSL, usually corresponding to a particular version of Direct3D. For instance, Shader Model 5.0 corresponds to Direct3D 11 and Shader Model 6.0 corresponds to Direct3D 12.

> *The Unity3D engine originally used Cg as its shading language, which is why Unity shaders traditionally looked like HLSL. Recently, Unity switched to pure HLSL. Unreal also uses HLSL.*

Basic Syntax

HLSL a C-like language, meaning most of the syntax of our HLSL code will be similar to basic C or C++ code: variables, statements, functions, code blocks and so forth all follow the same basic rules. The language also has a set of intrinsic functions available for performing common mathematical, vector and numeric operations, such as abs(), sin(), cos(), dot(), cross(), etc. It also has functions for GPU-specific

work like sampling textures. While we'll dive into some of the specifics of HLSL syntax below, the official HLSL Language Syntax documentation contains the full details.

Entry Point

As with most C-like languages, the entry point to a shader is a function named "main" (though the HLSL compiler can be configured to look for a different name). Parameters to a shader's main() function are how the shader receives data from a previous pipeline stage. The return value of a shader's main() function is how that shader passes data to the next pipeline stage. If multiple pieces of data are expected as input (parameters) or output (the return value), custom structs of data can be defined and used.

Below you'll see two examples of shader entry points. The first is a vertex shader, which uses two custom structs – VertexToPixel and VertexShaderInput – as its return type and parameter, respectively

```
VertexToPixel main(VertexShaderInput input)
```

The second is a pixel shader, which uses a single custom struct (VertexToPixel) to represent the data coming into the shader. Notice that both happen to use the same custom struct: VertexToPixel. Since the output of the vertex shader eventually becomes the input to the pixel shader, it's common to use the same struct among multiple shaders.

```
float4 main(VertexToPixel input) : SV_TARGET
```

We'll discuss that "SV_TARGET" piece soon. This is called a *semantic*. For now, just know that it is a built-in keyword denoting the purpose of the shader's output.

Variables, Data Types & Operators

Variable declarations, assignments and scope are very similar to C/C++. There are several optional keywords available when declaring variables, though we won't need any of them for our immediate purposes. All of the basic mathematical operators you'd expect are available in HLSL.

As for data types, HLSL primarily deals with numeric data, so we have basic scalar types such as *bool*, *int*, *uint*, *float* and *double*, among others, though we'll almost exclusively use **float** and **int**.

Since vectors and matrices are so prominent in our math, HLSL provides vector and matrix versions of each scalar type: **float2**, **float3**, **float4**, **float3x3**, **float4x4**, int2, int3, etc. The **matrix** data type is an alias for float4x4. Here are some examples of variable declarations and assignments:

```
float  size  = 9.2f;
float2 uv    = float2(0.1f, 1.0f);
float3 left  = float3(-1, 0, 0);
float4 color = float4(1.0f, 0.5f, 0.0f, 1.0f);
```

If we want a more complex set of data, we can create custom structs. We'll most often create these to match our vertex data or to express sets of data being returned from a shader. There are also pre-defined types for different GPU resources such as textures and samplers, which we'll cover in the future. Below is an example of a custom struct to describe our vertex data. Note the use of *semantics* again.

```
struct VertexShaderInput
{
	float3 localPosition : POSITION; // XYZ position
	float4 color         : COLOR;    // RGBA color
};
```

Semantics

Semantics are identifiers, like “COLOR” and “SV_TARGET” on the previous page, attached to certain variables. Any data that flows through the pipeline – that is, any data used as input to or output from a shader – is required to have a semantic. Semantics denote the intended usage of our input and output shader variables.

Some semantics are required and expected by the Graphics API. These are called “System Value” semantics and all start with “SV_”. SV_TARGET is a system value semantic that means “this value goes to the primary render target”. SV_POSITION is another system value semantic that means “this value is the screen position used by the rasterizer”. Every pipeline shader has at least one required System Value output semantic.

Let’s look at the pixel shader entry point from the previous page again:

```
float4 main(VertexToPixel input) : SV_TARGET
```

The return type of this function is a float4 (a 4-component vector). However, since this is shader output, it is required to have a semantic. In this case, as there is no variable here, we place the semantic on the whole function, signifying that the function’s return value should go to the primary render target.

The vertex shader entry point is a bit different:

```
VertexToPixel main(VertexShaderInput input)
```

Where are the semantics? They’re defined inside the structs, one per variable:

```
struct VertexShaderInput
{
	float3 localPosition : POSITION; // XYZ position
	float4 color         : COLOR;    // RGBA color
};

struct VertexToPixel
{
	float4 screenPosition : SV_POSITION; // XYZW - System Value Position
	float4 color          : COLOR;       // RGBA color
};
```

Note the usage of SV_POSITION in the VertexToPixel struct. Every vertex shader is required to have one 4D vector tagged as SV_POSITION, as that is how the rasterizer finds its normalized device coordinates. What are POSITION and COLOR? In older versions of HLSL, these specific identifiers had meaning to the pipeline. These days, we can use any identifier (like HORSE or CHEESE) for our custom data, though it is customary to stick with the more traditional semantic names like COLOR, POSITION, TEXCOORD, etc.

Flow Control

Flow control is the ability to control which line of code comes next. If statements, loops and the like are examples of flow control. HLSL supports all the classics: if, switch, while, do/while, for, break and continue. Though these are available, they are not always the most efficient, as per our discussions of GPU SIMD architecture and branching. While not strictly flow control, the ternary operator is also available to execute different statements based on an expression.

Functions

Just like most languages, we can write our own functions in HLSL. They require return types (or void) and zero or more parameters. You can create overloads of functions having different signatures. I highly recommend creating helper functions for common tasks rather than copying and pasting that code over and over.

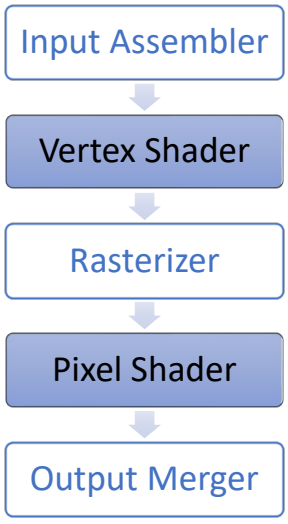
One non-obvious fact about HLSL functions is that they are all inline functions. An inline function is a function whose code is inserted where the function is actually called, rather than invoking a true function call. This has two effects: One, there is no "function call overhead" of maintaining a call stack, so the code is a bit faster. Two, it makes recursion impossible. We don't necessarily have to think about this while writing our code, but remember that recursive calls won't work.

#include

The shader compiler can include one code file at the top of another. A common setup is to put helper functions into one or more external files and all custom structs in another. These files often use the .hlsli extension (HLSL Include), though they can have any extension really. Do note that *#pragma once* is not supported by the shader compiler, so a C-style include guard (created with *#ifndef*) is required.

Shaders & The Pipeline

Even though might we consider shaders to be "programs", they're not like standard Windows applications. Since they're executed as part of a rendering pipeline, our shaders must be written to accept data from a previous stage and pass particular data to the next stage. In other words, there is standard input & output for each type of shader, though we can often customize it to fit our needs.



When it comes to "standard" 3D rendering – rasterizing a set of triangles in a 3D world from the point of view of a virtual camera – there are two required shaders: **vertex shaders** and **pixel shaders**. Other, optional shader stages exist, though we'll be focusing on these two, as they're all we need for now.

> *There is even a type of shader (a compute shader) that can be executed on its own without the rendering pipeline. We may touch on these later in the course.*

Vertex Shaders

The Vertex Shader stage of the Direct3D 11 pipeline sits between the Input Assembler and Rasterizer stages. At a minimum, it is required to output the 4D vector of normalized device coordinates (-1 to 1 on X and Y, and 0 to 1 on Z) used by the rasterizer. As was discussed in the reading on projection matrices, this 4D vector is assumed to be a set of homogenous coordinates: a coordinate system used to express a point in a projection. The rasterizer will perform the perspective divide – dividing (x,y,z) by w – for us and will then clip any triangles that fall partially, or wholly, outside the screen.

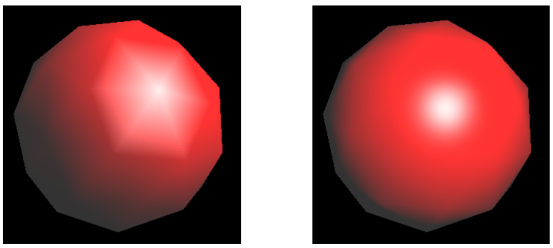
When rendering, one vertex shader thread is launched for each unique vertex being processed. The input to a vertex shader thread (one instance of the shader program) is all of the data associated with a single vertex. Though the output of the shader must include a float4 tagged with the SV_POSITION semantic, it will often contain other per-vertex data such as a color, texture coordinate and/or normal. Some of this data, like the texture coordinate, can simply be passed through the shader unaltered. Other data, like the normal of the vertex, will most likely need to be transformed similarly to the vertex position before being returned.

> *Reminder: To support returning multiple pieces of data, a custom struct is required for the return value. Inside this struct, each variable is required to be tagged with a semantic, since this is data moving from one pipeline stage to another.*

If we need external data, like transformation matrices or a time value for time-based animation, it must be passed in through a constant buffer. While not common, a vertex shader can also sample data from a texture; this is most often used for heightmaps (used for terrain rendering) or displacement maps (moving the points on a surface along their normals).

Though we could perform lighting equations here in the vertex shader, the results of those equations would then be linearly interpolated by the rasterizer, leading to rather bland looking shading. Instead, we pass the surface data down to the pixel shader so we can get non-linear results across the faces of our triangles. Here is a comparison of vertex lighting and pixel lighting on the same triangles:

Per-Vertex Lighting Per-Pixel Lighting



Pixel Shaders

The Pixel Shader stage of the Direct3D 11 pipeline sits between the Rasterizer and Output Merger stages. At a minimum, it is required to output a float4 tagged as SV_TARGET; the color that should be put into the primary render target. The values of each color channel must range from 0.0f to 1.0f. Thus, (0,0,0) is black, (1,1,1) is white and (0.5, 0.5, 0.5) is grey. By default, *blending* (the way in which we

approximate transparency) is turned off in most Graphics APIs, so the value we return for the alpha channel of our color will have no bearing on the result.

> *If you've used shaders in OpenGL, you may have seen or written fragment shaders. They're equivalent to pixel shaders in DirectX. It's just a different name.*

One pixel shader instance is launched for each rasterized pixel within the bounds of the screen. The input to each shader instance is an interpolated version of the data at each vertex of the triangle from which this pixel was rasterized. In this way, each pixel will get a unique set of data, ensuring the final color of each pixel is slightly different from the pixels surrounding it.

This is where the bulk of our shading work will occur. For instance, pixel shaders are where we run lighting equations, sample textures and approximate environmental effects like reflections. Due to this, our pixel shaders are usually our most expensive shaders and can be big performance bottlenecks.

Shaders & Constant Buffers

Though our shaders can have inputs (parameters) and outputs (return values), this data originates as vertex data from the Input Assembler stage. In other words, without any other external data, the only information available to our shaders comes from a vertex buffer. To access other data, such as transformation and camera matrices, we need to use constant buffers.

A constant buffer is a chunk of GPU memory that most often holds a set of data copied from CPU memory. This data can be overwritten whenever we want, allowing us to provide fresh data to our shaders each frame. A *cbuffer* definition in a shader describes the layout of a set of variables and maps them to a particular constant buffer *register* (or "slot"), allowing those cbuffer variables to literally pull their data from the corresponding constant buffer in GPU memory.

As we've discussed though, our cbuffer definition in our shader and the C++ struct that originally held that data must always match. Adding a variable in a shader means we need to add the same variable to the corresponding C++ struct. If we make a new shader with a unique set of cbuffer variables, we actually need *another unique C++ struct*, and *another constant buffer in GPU memory*! Two more shaders? Two more structs and two more constant buffers.

More like Constant Bummers…

As you can imagine, this gets quite unwieldy. If we're trying to make a somewhat reusable engine where we can write arbitrary shaders and easily feed data to them, it would be nice to make this whole constant-buffer-wrangling mess a bit more dynamic.

But that would mean we'd need to write an extensive set of helper functions that interpreted compiled shaders to extract constant buffer details, automatically created corresponding constant buffers on the GPU and allowed us to put data into them using the shader variable name directly in C++, all while aiming to cut down in CPU->GPU memory copies to remain efficient.

Yikes, that sounds like a lot of work! Yeah, it was!

SimpleShader

A set of classes to simplify Direct3D 11 shader & constant buffer handling, written by me, available on GitHub. SimpleShader lets you turn these…

```
struct VertexShaderExternalData
{
    DirectX::XMFLOAT4 colorTint;
    DirectX::XMFLOAT4X4 worldMatrix;
    DirectX::XMFLOAT4X4 viewMatrix;
    DirectX::XMFLOAT4X4 projectionMatrix;
};
```

```
VertexShaderExternalData vsData;
vsData.colorTint = material->GetColorTint();
vsData.worldMatrix = transform.GetWorldMatrix();
vsData.viewMatrix = camera->GetView();
vsData.projectionMatrix = camera->GetProjection();

D3D11_MAPPED_SUBRESOURCE mappedBuffer = {};
context->Map(vsConstantBuffer.Get(), 0, D3D11_MAP_WRITE_DISCARD, 0, &mappedBuffer);
memcpy(mappedBuffer.pData, &vsData, sizeof(vsData));
context->Unmap(vsConstantBuffer.Get(), 0);
```

Into this…

```
// Set variables using their actual names!
// Data is collected locally (in C++) until copy step below
simpleVertShader->SetFloat4("colorTint", material->GetColorTint());
simpleVertShader->SetMatrix4x4("world", transform.GetWorldMatrix());
simpleVertShader->SetMatrix4x4("view", camera->GetView());
simpleVertShader->SetMatrix4x4("proj", camera->GetProjection());

// Actually copy the entire buffer of data to the GPU
simpleVertShader->CopyAllBufferData();
```

No buffer struct necessary! A future assignment will guide you through you implementing SimpleShader.

Basic Usage

SimpleShader is made up of several classes, like SimplePixelShader and SimpleVertexShader. Instead of manually loading shader code yourself, you'll create one of these objects for each shader you want to load. SimpleShader then loads the compiled shader and uses *reflection* to gather information about the shader's constant buffers and the variables therein. Internally, it builds a hash table that maps the variables to their offsets in the constant buffers and creates its own CPU-side buffers to match.

If you had two vertex shaders and three pixel shaders in your project, you'd create two SimpleVertexShader and three SimplePixelShader objects. Each of these objects would create the necessary GPU resources and CPU mappings to handle all of the constant buffers. When you want to "send data to the GPU", you'd do so through the corresponding SimpleShader object.

These objects contain functions like SetFloat(), SetFloat4() and SetMatrix4x4(), which take a string and the actual data to set. This string should be the *actual variable name* in your shader. SimpleShader will look up that variable in its own internal mapping and put a copy of the data into its own CPU-side buffer. This is analogous to you filling out the struct before the map/memcpy/unmap steps. Once you've set all of the data you intend to send, call CopyAllBufferData() to perform the CPU→GPU copy. This allows you to set data as often as you'd like but exactly control when the GPU copy step occurs, as we want to minimize those copies.

Odds and Ends

Since the SimpleShader object is now your stand-in for the actual Direct3D shader, it has a simple function – SetShader() – to activate the shader in the Graphics API. It also has functions to simplify the usage of other resources, like textures and samplers, which will come in handy down the line.

In addition, since SimpleShader is digging through the compiled shader code anyway, it reverse engineers an InputLayout from each vertex shader that it loads, saving you the trouble of making one yourself. The InputLayout for a vertex shader is then set automatically whenever you call SetShader() on a SimpleVertexShader.

Lastly, the library will fail silently as much as possible. That means when an error occurs, like a shader failing to load, or a variable name not being found, the library carries on and doesn't tell you. However, you probably want to know when things go wrong. To that end, you can turn on error and warning reporting in the library using the following lines before loading any shaders. You'll then see messages in the console window when something goes wrong.

```
ISimpleShader::ReportErrors = true;
ISimpleShader::ReportWarnings = true;
```