LEARN THE FUNDAMENTALS OF SHADER WRITING IN UNITY IN THIS COMPLETE TUTORIAL SERIES.

In this Unity tutorial series we will be giving you a basic introduction to shader scripting, showing you how to write your own vertex/fragment shaders from scratch in unity.
I have learnt the hard way how to write shaders through books and experimentation, trial and error. The goal of this series is to take everything I have learned and structure it into easy to follow, ordered lessons that work in unity 4, with clear instructions on not only how to write clean and efficient code, but also how to debug and optimize your shaders. Effectively this is the series I wish I had when I started learning CG.

In this beginner section we will be covering how to deal with colors and textures, how to write complete lighting models from scratch and even delve into normal mapping.

This text tutorial is accompanied by a complete video. I recommend watching the video and referring to the text as reference.

Be sure to read the tutorial carefully, and post any questions you may have if you get stuck, as there is a lot to cover. I have referenced some material from the following books throughout this series:
http://en.wikibooks.org/wiki/Cg_Programming/Unity
http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter01.html
http://http.developer.nvidia.com/GPUGems/gpugems_pref02.html
http://http.developer.nvidia.com/GPUGems2/gpugems2_frontmatter.html
http://http.developer.nvidia.com/GPUGems3/gpugems3_pref02.html

I certainly recommend reading those books as they are fantastic and cover a lot more than what you will find here, what I have done is broken everything down into easy to follow, ordered lessons that will work in unity 4.

NOTE: This tutorial requires very basic knowledge of scripting such as variables and functions. If you have no scripting experience I recommend you first watch our Introduction to Scripting tutorial in the getting started section.

PART 1 - INTRODUCTION AND FLAT SHADING

When creating your materials for games there is no such thing as a single material that will work in every situation. There are a lot of different surface types and often you just can't represent that surface with an existing material in unity. This is where shader writing comes in. In this first part we will take a look at what a shader is and how they are structured.
If you find this first lesson a bit too much you might want to check out our introduction to scripting video to get accustomed to scripting in unity.
I recommend you follow the introduction to surface shaders before this tutorial, though it is not required. Also this first lesson is the most difficult of all the shader tutorials as you need to learn how shaders work, if you get stuck at all, do not worry you can leave a comment on the tutorial and we will do our best to explain things further.

SO WHAT IS A SHADER?

Behind every material in Unity is a small block of text called a 'shader', and it uses the .shader extension.
A shader contains a series of commands for everything that is needed to create the surface properties of the material such as:

Albedo – *also known as the ouput color of the material*
Alpha – *also known as transparency*
Normal – *the way the light reacts to the surface to give more bump and detail*
Specularity and Gloss – *how shiny the surface is*
Emission – *the light properties such as rim lighting*

We will also look at other properties such as parallax and displacement later in this series but these are the 5 base properties in unity.

WHAT MAKES UP A SHADER?

Very little actually, we have some user definable properties such as color, images, sliders. We also have what's called a 'subshader', This contains all the info required to make the shader work. And finally, there is a 'fallback' which is what the shader will revert to upon failure such as if the device cannot support that shader.
However unlike the surface shaders we covered previously, we now have two more elements to consider. The vertex and fragment programs of a shader which act as the functions that control our shader. More on this later.

WHY WOULD WE WANT TO KNOW HOW TO WRITE SHADERS?

The primary reason is so you can have your surface react to lighting exactly how you want it to without spending hours trying to find a shader that fits your needs. Another reason is often you will find a shader someone has written and either want to find out how it works, or perhaps you want to change it to work how you want it to. A good example is perhaps you found a nice toon shader that is exactly what you want, but you want to have your vertex colors affect the shader. You could modify it yourself to do this.

WHAT LANGUAGE DOES UNITY USE FOR IT'S SHADERS?

There are a few different shader languages for Unity: shaderlab, CG/HLSL, GLSL

But you don't have to learn them all, there is a few things to note though. No matter what shader you are writing, you will use shaderlab as a wrapper around the code, this is how unity processes your shader, and the the CG code goes in the middle. To extend this example, if you want to convert a unity shader to work in maya/max viewport, you replace the shaderlab with the maya/max equivelent. This makes it very easy to transfer your shaders between programs. I use this technique so that I can have the same shader in my modeling app as in Unity so I know what I see in the viewport will be the same as what is in unity.

WHAT TYPES OF SHADERS ARE THERE?

There are three types of shaders in Unity:

Surface shaders – *These are the most common and was covered in a previous series. These shaders will interact with lights and shadows and are the easiest shader to use. However they do a lot of calculations and can sometimes cause a drop in frame rate. These are written in CG/HLSL*
Vertex and Fragment shaders – *These shaders are the best when you need something custom from scratch or want to work with post processing effects. However, they require more code. These are written in CG/HLSL*
Fixed Function shaders – *These are written for old hardware and completely in shaderlab. We use these as a last resort shader when the device is too old to support the newer shaders.*

WHAT IS VERTEX/FRAGMENT SHADERS?

As we are covering vertex/fragment shaders in this series I'd like to cover a little about what they are first.
To start off with, every shader is built out of three parts, the interface, the vertex program, and the fragment program. Shaderlab shaders and surface shaders generate all this behind the scenes so we never see it, but here we will learn how to write it ourselves.
In the unity documentation you will find that these are called vertex and fragment programs, shaders or functions.
it really does not matter what you call them as long as you understand what each part does, in essence they are just a function.

Interface
The 'interface' is user defined properties that are passed to the shader, unity built in functions, and any definitions(tags/pragma).

Vertex Program
The vertex program takes everything the interface hands the shader and works with it on a per-vertex level.
The vertex program also requires two structs, the input struct defines any semantics such as vertex position and texture coordinates it is allowed to access, the output struct is an empty struct for the vertex program to write to.

Fragment Program
The fragment program takes the result from the vertex program and works with it on a per pixel level, the fragment program is often called the pixel shader, and only requires the vertex program as an input.

Where to place code
Does your code belong in the vertex program, fragment program, or outside them both? This is an often overlooked question that we will be discussing in the last part of this beginner section, but in short:
Does this code need to be evaluated once per frame?
- for example calculating the position of the object relative to another object.
- It belongs outside the programs in the pass.
Does this code need to be evaluated for each vertex?
- For example calculating the distance to a point light.
- It goes in the vertex program.
Does this code need to be evaluated for every pixel?
- For example handling a texture.
- It goes in the fragment program.

Don't worry if this does not make sense, everything will become more apparent as we progress through the series.

WRITING A FLAT SHADER

Ok so here is the format of a basic flat color shader:

```
Shader "unityCookie/introduction/1 - Flat Color" {
  Properties {
        _Color ("Color", Color) = (1.0,1.0,1.0,1.0)
  }
  SubShader {
        Pass {
              CGPROGRAM
              #pragma vertex vert
              #pragma fragment frag

              //user defined variables
              float4 _Color;

              //Base Input Structs
              struct vertexInput{
                    float4 vertex : POSITION;
              };
              struct vertexOutput {
                    float4 pos : SV_POSITION;
              };

              //vertex function
              vertexOutput vert(vertexInput v){
                    vertexOutput o;
                    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);
                    return o;
              }
```

```
                        //fragment function
                        float4 frag(vertexOutput i) : COLOR
                        {
                                return _Color;
                        }
                        ENDCG
                }
        }
        //fallback commented out during development
        //Fallback “Diffuse”
}
```

Let’s break this down into it’s simplest elements.

```
Shader “unityCookie/introduction/1 - Flat Color” { //This is the name of your shader
   Properties {
      //This is where any properties such as images, sliders and color pickers can go.
   }
   Pass{
      SubShader {
         //Your shader goes in here
      }
   }
}
```

That looks much simpler right?

That is shaderlab, without any of the extra bits. We have user definable properties, and the subshader. These are nicely grouped into little blocks. The subshader is where all of our shader code goes. So let’s break this down even further.

Let’s start by talking about properties, as we will be using these a lot.

Properties
A properties block allows us to have adjustable parameters within Unity such as images, slider values and color pickers.
This saves us having to go into the shader every time we want to adjust something.
We can place properties in the properties block, and they come in a range of different types.
They are written in the format:
- name(“display name”, Type) = Starting value
The name usually starts with an underscore, but is not required
Here is some examples of properties

```
Properties {
   _MainTex (“Texture”, 2D) = “white” {} //This is a standard 2D image input, default white
   _BumpTex (“Normal Map”, 2D) = “bump” {} // This is a normal map, the “bump” start value tells Unity to make
sure it is a normal map
   _ColorTint (“Color Tint”, Color) = (1,1,1,1) //This creates a color picker, default white
   _TintValue (“Tint Intensity”, Range(0.0,1.0)) = 0.5 //creates a slider
   _Darkness (“Darkness”, Float) = 0.25 //Creates a number input
   _Coord (“Vector coord”, Vector) = (1,1,1,1) //Creates a vector input
   _Cube (“Cube Map”,Cube) = “” {} //This is a cubemap
}
```

Our properties go inside the subshader when we define them.

To define a property so we can use it we need to convert it into something Unity understands.

sampler2D [property];
This tells Unity the property is a 2D image

float4 [property];

This tells Unity the property uses 4 floating point digits, most common for RGBA values such as a color picker, this is the most computationally expensive.

half3 [property];
This is a half vector, a half is smaller than a float so takes less data, and is less accurate at +-60000.000 range with 3.3 decimal precision. The 3 is telling Unity the property has 3 half values, such as a vector.

fixed4 [property];
A fixed value is smaller than a half, and the most optimized at +-2.0 with 1/256th precision. For mobile shaders you will see a lot of fixed variables, we also tend to use fixed if we are not doing any calculations with it.

So after that your probably like "Woah! so why are will still using floats?!"
Well floats are easier to work with, and it's not a major performance hit to use them.
We will tend to use fixed as the last variable as it is faster, such as if we want to take the alpha channel of an image and use it as a gloss map. We simply aren't using the high accuracy of the float, do note that there is more to half, fixed and float than just accuracy and performace, but we won't go into that here, just understand that they are interchangeable terms, we will discuss them in part 9 when we cover optimization and also when we go into mobile shaders they play a big role.

Pass

Most shaders are made out of multiple passes, similar to how render passes work, in this shader we have one pass but we will be learning more about passes later on.

SubShader

The entirety of the shader is contained with the shader function, while the actual shader code is contained in what is called a subshader
- This is like the actual shader function within our shader function.
These subshaders run all the code for our shaders and you can have multiple of these.
- For example you may have a subshader for pc, another for xbox and another for wii, and Unity will use the subshader we tell it to for the target platform allowing us to easily save out shaders for each platform within the single shader.
For now we will be using a single subshader.

CGPROGRAM

to

ENGCG

This block tells unity that you are about to start writing in CGPROGRAM, also known as CGFX.
CGPROGRAM is the most common subshader you will come across in Unity

#pragma vertex vert
#pragma fragment frag

Pragma's are little definitions that tell unity which function does what. In this case we are telling unity we want to use the function 'vert' as the vertex function of our shader, and 'frag' as the fragment function of our shader. We can name these whatever we want but vert and frag are the most common so we will stick with these in this series.

float4 _Color;

Here we take the _Color variable we created in the properties block and assign it to a float4 variable, a float4 is like an array of 4 floating point numbers, other variations is:
float, float2, float3, float4 - array's of floats
half, half2, half3, half4 - like a smaller float
fixed, fixed2, fixed3, fixed4 - small floats designed for direct output (i.e: the last calculation for this variable)

There are many more that we will discuss later, but just understand for now that we must assign the property to a variable so we can access it in our shader.

The structs

```
//Base Input Structs
struct vertexInput{
  float4 vertex : POSITION;
};
struct vertexOutput {
  float4 pos : SV_POSITION;
};
```

The structs are very important in shader witing, these control what comes and goes from the vertex shader. A struct is short for structure, a type of record that contains variables and their respective values. We use an input struct to define what information we want to pass to the vertex program, and an output struct to define what we want to recieve from the vertex program. I use vertexInput and vertexOutput as my struct names but you can use what you like here.

Now let's talk about semantics, a semantic is a kind of variable unity passes to shaders such as vertex position, normal direction and texture coordinates.
COLOR - a color, contains the vertex colors.
POSITION - the vertex position.
SV_POSITION - output only, this is like position but cross platform, required for dx11 shaders.
NORMAL - the vertex normal.
TANGENT - the tangent direction.
TEXCOORD0 - The first UV map
TEXCOORD1 - The second uv map
TEXCOORD2 . . . - Empty semantics for values you want to transfer between the vertex and fragment programs, this will make more sense later.
You will also notice we use a colon instead of an equals sign here when defining semantics.

Vertex Program

```
vertexOutput vert(vertexInput v){
  vertexOutput o;
  o.pos = mul(UNITY_MATRIX_MVP, v.vertex);
  return o;
}
```

The vertex function is where we process eall commands every vertex, every frame. We create the vertex program with the vert function (you can name this whatever as long as it is the same as in the pragma), We then pass it vertexInput as the input struct and assign it to the variable 'v', similarily as you would pass variables to a function we pass the entire struct. Placing vertexOutput before the vert function means we are going to be editing the vertexOutput struct we set up earlier.
Next we start the vertex program with:

```
vertexOutput o;
```

This means we want to use vertexOutput as a base for the variable o, which is shorter and easier to use.

```
o.pos = mul(UNITY_MATRIX_MVP, v.vertex);
```

UNITY_MATRIX_MVP is a float4x4 matrix (like 4 float4's stacked in an array), this is the view projection matrix. Multiplying the view projection matrix with the vertex position gets a screen space vertex position to pass to the gpu.
Ok so something I want to talk about is that mul() function as it confused me for a while, it is a multiplier that is specific to when working with matrices.
You might think you can use a multiplication symbol, but this won't work as we are not multiplying the whole matrix, but rather each part of the matrix. Here is a more visual example:

```
float4x4 myMatrix;
myMatrix[0] = UNITY_MATRIX_MVP[0] * v.vertex;
myMatrix[1] = UNITY_MATRIX_MVP[1] * v.vertex;
myMatrix[2] = UNITY_MATRIX_MVP[2] * v.vertex;
```

```
    myMatrix[3] = UNITY_MATRIX_MVP[3] * v.vertex;
```

This is the same as

```
    myMatrix = mul(UNITY_MATRIX_MVP, v.vertex);
```

Fragment Program

```
    float4 frag(vertexOutput i) : COLOR
    {
        return _Color;
    }
```

The fragment program is also known as the pixel shader as it calulates everything every pixel every frame.
You will notice this has a slightly different layout than the vertex program, we have no base struct to assign it to, so we create one, float4 frag(vertexOutput i) : COLOR looks very similar to float4 frag : COLOR doesn't it?
We are merely skipping the output struct and directly assigning it to a semantic as we don't need to do anything with it after the fragment program.
We once again pass a struct to the function and assign it to 'i' as we did with the vertex function, but this time we use vertexOutput so we can get at all those variables we worked with in the vertex program.
Finally we return _Color to directly return that float4 variable we have in our properties.

Fallback "Diffuse"

Should the shader fail to run then we will use a diffuse shader instead so it doesn't look all funky. An example of this is if your PS3 shader does not run on your iPhone.
We comment this out during development so we can see when we have errors in unity.

And that is a quick breakdown of a basic shader in unity. In the next part we will take a look at writing a basic lambert lighting model for our shader.