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 7 - NORMAL MAPS

Adding textures to our shader was nice and easy, but now it's time to tackle normal maps. Normal maps make the surface appear bumpy and adds a whole new level of detail to our shader, but it also adds in a while lot more code.
So prepare yourself, normal maps are difficult to explain, difficult to understand, but really really cool.

HOW DOES IT WORK?

So a normal, as you should know by now is the direction a vertex is facing, these normals are interpolated across the surface between vertices to get a smooth result.
A tangent space normal map rotates these interpolated normals to give the illusion of detail in the surface.
A object space normal map overwrites these normals completely for a nicer result, but these are far more complex and we won't be getting into these in this lesson.

Now how do we rotate these surface normals?
Well we need three things. The normal direction, the tangent direction and the binormal(co-tangent) direction.
The tangent is as we discussed previously a vector at 90 degrees to the normal across the surface, the binormal is 90 degrees to the tangent to go the other way across the surface. Using these we can rotate our surface normals.

The formula is available on wikipedia if you want it, but I found it made things more confusing so I will not include it here.

THE CODE

Let's start off with loading in a normal map.

```
_BumpMap ("Bump", 2D) = "bump" {}
```

Very similar to the _MainTex, but this time the default value is bump instead of white, this makes sure the user loads in the correct type of map. Of course we then sample the map and get the offset and scaling:

```
uniform sampler2D _BumpMap;
uniform float4 _BumpMap_ST;
```

And that's that part done, make sure it is called _BumpMap as some more advanced modules require that name just like for _MainTex, any other textures do not matter though.
Let's get started with the structs.

THE STRUCTS

```
struct vertexInput{
    float4 vertex : POSITION;
    float3 normal : NORMAL;
    float4 texcoord : TEXCOORD0;
    float4 tangent : TANGENT;
};
struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 tex : TEXCOORD0;
    float4 posWorld : TEXCOORD1;
    float3 tangentWorld : TEXCOORD2;
    float3 normalWorld : TEXCOORD3;
    float3 binormalWorld : TEXCOORD4;
};
```

Ok so a few new things here, first we need to grab the tangent vector to pass to the vertex program, and we create three more floats for the tangent, normal and binormal to pass to the fragment program.
Let's move on to the vertex program.

VERTEX PROGRAM

Let's just look at the three new lines here:

```
o.tangentWorld = normalize( float3( mul( _Object2World, float4( float3(v.tangent.xyz), 0.0) ).xyz ) );
o.normalWorld = normalize( mul( float4(v.normal.xyz, 0.0), _World2Object).xyz );
o.binormalWorld = normalize( cross(o.normalWorld, o.tangentWorld).xyz * v.tangent.w);
```

Ok so o.normalWorld is the same as the previous o.normalDir, so we can ignore that line. But let's take a look at the other two.

```
o.tangentWorld = normalize( float3( mul( _Object2World, float4( float3(v.tangent.xyz), 0.0) ).xyz ) );
```

Fairly straightforward, we multiply the _Object2World transpose by our tangent direction to get the tangent direction in world space to use in the fragment shader, as before all the rest of that code is merely making it work, normalize squishes the vector into 0 - 1, float4 makes the fourth vector of the tangent 0 as we are only interested in the xyz direction, float3 truncates off the w component of the equation. All stuff we have covered previously.

```
o.binormalWorld = normalize( cross(o.normalWorld, o.tangentWorld).xyz * v.tangent.w);
```

Now binormal on the other hand, this is new. We now get a look at the cross product. The cross product takes two vectors and makes a new vector pointing perpendicular to the first two. the length of this vector varies based on the rotational difference of the first two vectors, but when we normalize it it will be back within the correct range. Finally multiplying the cross product by v.tangent.w will give us the correct binormal. Multiplying by the w component is specific to unity and let's unity do cool behind the scenes stuff to make the normals work.

Whew, that's the easy part over, let's take a look at the fragment program.

FRAGMENT PROGRAM

Ok so it's not that bad, but it is a little more code.

```
float4 texN =  tex2D (_BumpMap, _BumpMap_ST.xy * i.tex.xy + _BumpMap_ST.zw);

//unpackNormal function
float3 localCoords = float3(2.0 * texN.ag - float2(1.0, 1.0), 0.0);
localCoords.z = 1.0 - 0.5 * dot(localCoords, localCoords);

float3x3 local2WorldTranspose = float3x3(
  i.tangentWorld,
  i.binormalWorld,
  i.normalWorld);
float3 normalDirection = normalize(mul(localCoords, local2WorldTranspose));
```

Heh, so what does all this do? you know the drill, let's work through this line by line.

```
float4 texN =  tex2D (_BumpMap, _BumpMap_ST.xy * i.tex.xy + _BumpMap_ST.zw);
```

First we need to unwrap the texture to the coordinates, just like we did with the diffuse texture.
Next we need to unpack the normals, To do this, we take the normal map which is in 0 - 1 scale, subtract one and multiply by two to place it in -1 to 1 scale, and place 0 in the third coordinate as we only need it so unity considers it a vector.
Now unity stores normal map data in the alpha and green channels of the image, this is to make it lighter on memory.

```
float3 localCoords = float3(2.0 * texN.ag - float2(1.0, 1.0), 0.0);
```

Ok so that's cool, but with the third coordinate at 0 there is no depth, which means a really bumpy normal map (0 = bumpy), so what we need to do is calculate the the third depth coordinate from the other two.

```
localCoords.z = 1.0 - 0.5 * dot(localCoords, localCoords);
```

That'll do it, we take the dot product of the vector against itself (x*x + y*y + z*z), divide it by two and subtract it from 1 to get a bumpy surface that is still relevant to the normal map.
To complicated or need to save on math, just leave localCoords.z = 1.0; it won't look much different, alternatively you can hook up a slider to this localCoords.z value to give control over the bumpyness.

So what's next, well we now need to actually rotate those original vectors by our new texture coordinate vectors we made. How do we do that? with a matrix of course!

```
float3x3 local2WorldTranspose = float3x3(
  i.tangentWorld,
  i.binormalWorld,
  i.normalWorld);
```

Ok so this is a new one, float3x3 creates a three by three matrix, in this case we stack up the tangent, binormal and normal float3's into a big matrix which we can then multiply our new texture coordinates by to get the rotated direction.
Doesn't make sense? Let's elaborate on this.

```
float3 normalDirection = normalize(mul(localCoords, local2WorldTranspose));
```

This equates to:

```
float3 normalDirection;
normalDirection.x = localCoords.x * i.tangentWorld; //The x axis is at tangent to the normal
normalDirection.y = localCoords.y * i.binormalWorld; //The y axis is at tangents to both normal and tangent.
normalDirection.z = localCoords.z * i.normalWorld; //The z axis is depth, which happens to be along the normal
axis.
```

Make more sense now? Good, let's incorporate this into our shader.
But wait one second, We have run out of mathmatical operators, we could again remove that darn rim lighting, but instead, let's upgrade our shader from shader model 2 (default) to shader model 3! There is a total of 5 shader models at the time of writing this, We usually want to avoid upgrading, but shader model 3 is still well supported. We could also make localCoords.z = 1 which would stop the need to upgrade aswell.

```
#pragma target 3.0
```

FINAL CODE

```
Shader “unityCookie/introduction/7 - Normal Map - pixel” {
    Properties {
        _Color (“Color Tint”, Color) = (1.0,1.0,1.0,1.0)
        _MainTex (“Diffuse Texture, Gloss(A)”, 2D) = “white” {}
        _BumpMap (“Bump”, 2D) = “bump” {}
        _SpecColor (“Specular Color”, Color) = (1,1,1,1)
        _Shininess (“Shininess”, Float) = 10
        _RimColor (“Rim Color”, Color) = (1,1,1,1)
        _RimPower (“Rim Power”,Range(0.1,10)) = 3.0
    }
    SubShader {
        Pass {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #pragma exclude_renderers flash
            #pragma target 3.0

            //user defined variables
            uniform sampler2D _MainTex;
            uniform float4 _MainTex_ST;
            uniform sampler2D _BumpMap;
            uniform float4 _BumpMap_ST;
            uniform float4 _Color;
            uniform float4 _SpecColor;
            uniform float _Shininess;
            uniform float4 _RimColor;
            uniform float _RimPower;

            //Unity defined Variables
            uniform float4 _LightColor0;

            //Base Input Structs
            struct vertexInput{
                float4 vertex : POSITION;
                float3 normal : NORMAL;
                float4 texcoord : TEXCOORD0;
                float4 tangent : TANGENT;
            };
            struct vertexOutput {
                float4 pos : SV_POSITION;
                float4 tex : TEXCOORD0;
                float4 posWorld : TEXCOORD1;
                float3 tangentWorld : TEXCOORD2;
                float3 normalWorld : TEXCOORD3;
                float3 binormalWorld : TEXCOORD4;
            };

            //vertex function
            vertexOutput vert(vertexInput v){

                vertexOutput o;

                o.tangentWorld = normalize( float3( mul( _Object2World, float4( float3(v.tangent.xyz), 0.0) ).xyz ) );
                o.normalWorld = normalize( mul( float4(v.normal.xyz, 0.0), _World2Object).xyz );
                o.binormalWorld = normalize( cross(o.normalWorld, o.tangentWorld).xyz * v.tangent.w);


                o.posWorld = mul(_Object2World, v.vertex);
                //o.normalDir = normalize( float3( mul( float4(v.normal, 0.0),_World2Object ).xyz ) );
```

```
            o.pos = mul(UNITY_MATRIX_MVP, v.vertex);
            o.tex = v.texcoord;
            return o;
        }

        //fragment function
        float4 frag(vertexOutput i) : COLOR
        {
            //float3 normalDirection = normalize(i.normalDir);
            float3 viewDirection = normalize(_WorldSpaceCameraPos.xyz - float3(i.posWorld.xyz));
            float3 lightDirection;
            float atten;

            if (0.0 == _WorldSpaceLightPos0.w) //directional Light
            {
                atten = 1.0;
                lightDirection = normalize(float3(_WorldSpaceLightPos0.xyz));
            }
            else{
                float3 vertexToLightSource = float3(_WorldSpaceLightPos0.xyz - float3(i.posWorld.xyz));
                float distance = length(vertexToLightSource);
                atten = 1.0/distance;
                lightDirection = normalize(vertexToLightSource);
            }

            //Texture Maps
            float4 tex = tex2D(_MainTex, _MainTex_ST.xy * i.tex.xy + _MainTex_ST.zw);
            float4 texN =  tex2D (_BumpMap, _BumpMap_ST.xy * i.tex.xy + _BumpMap_ST.zw);

            //unpackNormal function
            float3 localCoords = float3(2.0 * texN.ag - float2(1.0, 1.0), 0.0);
            localCoords.z = 1.0 - 0.5 * dot(localCoords, localCoords);

            float3x3 local2WorldTranspose = float3x3(
              i.tangentWorld,
              i.binormalWorld,
              i.normalWorld);
            float3 normalDirection = normalize(mul(localCoords, local2WorldTranspose));


            //Lighting
            float3 ambientLight = UNITY_LIGHTMODEL_AMBIENT.rgb;
            float3 diffuseReflection = atten * float3(_LightColor0.rgb) * max(0.0, dot(normalDirection,
lightDirection));
            float3 specularReflection = max(0.0, dot(normalDirection, lightDirection)) * atten * float3(_LightColor0.
rgb) * float3(_SpecColor.rgb) * pow(max(0.0,dot(reflect(-lightDirection, normalDirection), viewDirection)),
_Shininess);

            //Rim Lighting
            half rim = 1 - saturate(dot(normalize(viewDirection), normalDirection));
            float3 rimLighting = max(0.0,dot(normalDirection, lightDirection) * _RimColor * pow(rim, _RimPower));

            float3 lightFinal = ambientLight + diffuseReflection + specularReflection + rimLighting;

            return float4(tex.rgb * lightFinal * _Color.rgb, 1.0);
        }
        ENDCG
    }
  }
  //fallback commented out during development
  //Fallback "Diffuse"
}
```

And that is it, in the next lesson we move on to look at different texture maps we can use.