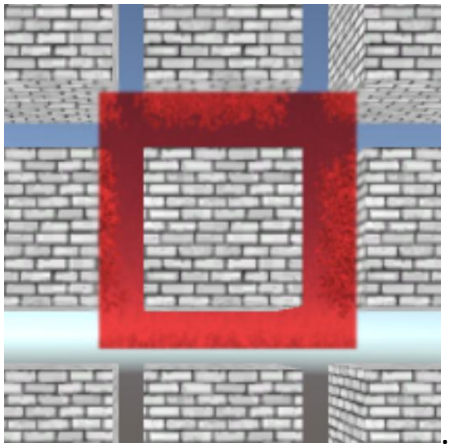
Unity Shader Tutorial (Outline Object)

– Morgan James s1602293

Intro: In this tutorial we will go through how to construct basic Unity shaders leading up to a shader that applies an outline to objects that we hover over with our mouse. That looks like this

 .

1. Apply Texture

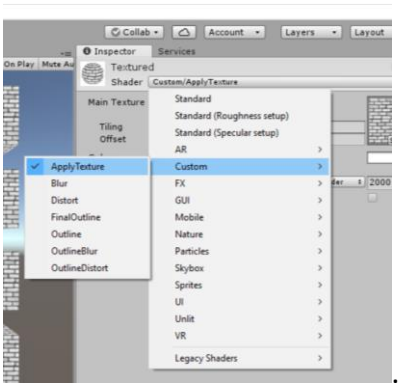
    1.1. Create a new Unity Project.

    1.2. Right click the project file location and press Create > Shader > Unlit Shader name the shader ApplyTexture and open it up in Visual Studio.

```
Shader "Unlit/ApplyTexture"
{
        Properties
        {

        }
}
```

    1.3. Delete everything apart from the parts below

    1.4. You can now change the name of the folder that the shader is contained in by changing Unlit to whatever you like (I named my folder Custom). When you go to apply the shader it will appear in that folder like so



1.5. Next, we want to add some properties to our shader. Properties are the variables of the shader world they are the inputs to the function that is our shader. If we add a texture property allowing us to declare the input texture of the object and a colour which will tint that texture property like so

```
Properties//Variables
{
        _MainTex("Main Texture (RBG)", 2D) = "white" {}
        _Color("Color", Color) = (1,1,1,1)
}
```

We will be able to see these variables inside the inspector when applying this shader to a material as seen below

1.6. Beneath the Properties we can declare our sub shader and our passes. Each pass can be looked at as how many times the object will receive rendering iterations, so if you have two passes it will render the first pass and then the second and so on. You can have as many passes as you like but for this shader we only need

one. Go ahead and add the pass like this

```
SubShader
        {
                Pass
                {

                }
        }
```

1.7. Inside of our pass we need to add the following opening and closing statements

```
Pass
{
        CGPROGRAM

        ENDCG
}
```

We need to add these statements as Unity has its own scripting language (Shader Lab) separate to C for graphics by Nvidia which is the shader language and thus everything inside of the statement is converted from C for graphics to shader lab so that Unity knows what to do.

1.8. Now that we have a shader that can read C for graphics lets start writing some. The first thing we should do when writing a shader is declare our function names. A shader is made up of two main functions the first being a vertex function that builds the object and the second being the fragment function which colours the object in. We can declare them like so

```
#pragma vertex vert//Define for the building function.

#pragma fragment frag//Define for coloring function.
```

This means that when the shader is looking to do its vertex calculations it is going to go to the function named “vert” and when it’s looking to colour it in it will use the “frag” function.

1.9. Before we say what’s in our vert and frag functions we must do a few more things. The first of which is to include a library of useful functions and definitions to use in our shader called UnityCG.cginc. We can easily include this just like any other C# include underneath the vert and frag declarations like so

```
#include "UnityCG.cginc"//Built in shader functions.
```

1.10. Next, we need to define some structures to say how our vert and frag functions will receive data. The first of which is called “appdata” and this structure defines how and what data the vertex function receives. Appdata can

be defined like so

```
struct appdata//How the vertex function receives info.
{
        float4 vertex : POSITION;
        float2 uv : TEXCOORD0;
};
```

1.11. After saying how the vertex function receives info we need to say how the fragment function receives info from the vertex function in a structure called v2f (vertex to fragment). This will look something like this

```
struct v2f//How the fragment function receives info.
{
        float4 pos : SV_POSITION;
        float2 uv : TEXCOORD0;
};
```

1.12. We still need one more thing before we can say what our vertex and fragment functions do and that is to reimport the properties but into the C for graphics language inside of our pass. This can be done underneath our structs like so

```
float4 _Color;
sampler2D _MainTex;
```

For textures we use sampler2D as it takes a sample of that texture. You must make sure that the imported names match the property names at the top of the shader.

1.13. Now on to the fun stuff, defining the contented of the vertex function “vert”. What we want this function to do is take in the info about the object and build it in the camera’s view and then to output that info to the fragment function, so it knows what to colour. The vertex function will look like this

```
v2f vert(appdata IN)
{
    v2f OUT;

    OUT.pos = UnityObjectToClipPos(IN.vertex);
    OUT.uv = IN.uv;

    return OUT;
}
```

The function will output a v2f structure to go to the fragment function and take in the objects data (appdata). We create a v2f inside of the function that will be the output. We set

the pos of the output to be equal to the appdata but constructed inside of the camera's view. We also set the outputs uv's to the inputs uv's as they don't change.

1.14. You probably already know the last part to this shader. You're right it's the fragment function "frag". Now the fragment function of just applying a texture and a tint is simple and looks like this

```
fixed4 frag(v2f IN) : SV_Target
{
float4 texColor = tex2D(_MainTex, IN.uv);//Wraps the texture around the uv's.
return texColor * _Color;//Tints the texture.
}
```

The function outputs a colour(fixed4) and takes in the v2f from the "vert" function we also put : SV_Target to say that the function is giving us a colour. We defined a texture colour called texColor which is equal to the texture wraped around the objects uv's and then we return this multiplied by the colour variable to tint it whatever colour we want.

1.15. You did it! You have created your first shader. Now to test it out, inside of Unity create a new object for example a cube by right clicking the hierarchy and clicking 3D Object > Cube. Then also create a new material named "Textured" by right clicking the project assets location and clicking Create > Material and typing the name. We can now position our camera to look at our cube and drag the material onto the cube. Now drag your shader onto the material and set the colour and texture to whatever you want and you should end up with something like this

2. Outline

2.1. Now to get a shader that outlines we are going to have to add somethings to a regular shader like and extra pass as we must render the border and the shape and when we render the border we must render it larger than the actual shape by

multiplying the vertices by a set amount. To start making this shader duplicate the “ApplyTexture” shader by clicking on it in the project asset viewer panel and press CTRL + D and name this shader “Outline”.

2.2. Open the newly made outline shader to start editing it by double clicking on it.

2.3. Change the name of the shader to Outline so it looks like the following

```
Shader "Custom/Outline"
```

2.4. Next, we need to add a few more properties due to the outline so go ahead and add them to look like this

```
Properties//Variables
{
        _MainTex("Main Texture (RBG)", 2D) = "white" {}
        _Color("Color", Color) = (1,1,1,1)

        _OutlineTex("Outline Texture", 2D) = "white" {}
        _OutlineColor("Outline Color", Color) = (1,1,1,1)
        _OutlineWidth("Outline Width", Range(1.0,10.0)) = 1.1
}
```

We added a texture and colour for our outline for increased customisation and an outline width float in the range of 1 – 10 with a default of 1.1. This outline width will be used to determine how much to scale the outline by.

2.5. A second pass is needed and, so we can duplicate our first pass and paste it directly underneath our first pass.

2.6. We can give our passes names not only, so we know what they do but also, so they can be used in other shaders. So, lets go ahead and give our passes names like so

We will name out first pass “OUTLINE” and our second “OBJECT” as we will first render the border then the object in front of it.

```
Pass
{
        Name "OUTLINE"
```

```
Pass
{
        Name "OBJECT"
```

2.7. The object pass is fine as it is. So, all we need to change is the outline pass and we can do so by first changing the re-imports to match the outline's variables by going from

```
float4 _Color;
sampler2D _MainTex;
```

To

```
float _OutlineWidth;
float4 _OutlineColor;
sampler2D _OutlineTex;
```

2.8. The next thing to change in the outline pass is the vertex function "vert" and all we need to do is to add one line of code above where we declare v2f OUT;

```
IN.vertex.xyz *= _OutlineWidth;
```

What this line of code will do is multiply the vertices of the object so that is larger than the default object. It will increase in size depending on the outline width variable.

2.9. Now to make the fragment function work with the outline variables. Just change the names of the texture and colour it uses to match the outline variables like so

```
fixed4 frag(v2f IN) : SV_Target
{
        float4 texColor = tex2D(_OutlineTex, IN.uv);
        return texColor * _OutlineColor;
}
```

2.10. The shader is almost complete apart from two small details that make a big difference. The first is the transparent queue tag that we will add above the pass but still in the SubShader and we can do that, so it looks like this

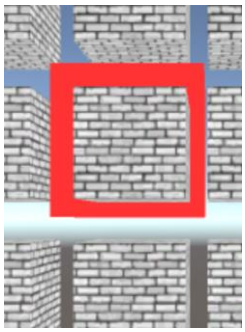
```
SubShader
{
        Tags
        {
                "Queue" = "Transparent"
        }
        Pass
        {
                Name "OUTLINE"
```

What this does is that it makes the objects render queue the same as a transparent object which is very high therefore making it render after other objects so that other objects don't appear in front of our outline.

2.11. The final thing we need to add is an indication to not use ZWriting no the outline pass so that we can render the outline and then render the object on top of the outline. We can do this by adding this line of code beneath the name of the outline pass

```
ZWrite Off
```

2.12. Now our outline shader is done we can go ahead and test it out the same way we tested our ApplyTexture shader. So, make a new material and drag the shader onto it set the variables to whatever you like and drag the material onto the cube. You should get something like this



3. Blur

3.1. Now to get a bit more mathematical! To sum up what we are going to do in this shader we are going to grab everything behind the object, blur it horizontally, grab that and blur it vertically. The blurring algorithm we are going to use is gaussian blur. To start duplicate the border shader and name it "Blur".

3.2. Open the blur shader in Visual Studio by double clicking it.

3.3. Rename the shader to look like this

```
Shader "Custom/Blur"
```

3.4. We're going to change the variables to only include a blur radius and intensity like so

```
Properties//Variables
        {
```

```
                _BlurRadius("Blur Radius", Range(0.0,20.0)) = 1
                _Intensity("Blur Intensity", Range(0.0,1.0)) = 0.01
        }
```

3.5. Now change the name of the first pass to be called “BLURHORIZONTALLY” like so

```
Pass
                {
                        Name "HORIZONTALBLUR"
```

3.6. Just before the pass we need to add a grab pass like this

```
GrabPass{}

Pass
{
```

This grab pass takes everything behind the object and puts it into a texture for the horizontal blur pass to use.

3.7. We can now change the v2f structure to indicate that its using the grab pass uv’s in a float4 like so

```
struct v2f
{
        float4 vertex : SV_POSITION;
        float4 uvgrab : TEXCOORD0;
};
```

3.8. The appdata structure may also be deleted due to having built in structures that we can use in our vertex function instead like appdata_base which in theory is a similar thing. So, our vert function should start like this now

```
v2f vert(appdata_base IN)
{
```

3.9. The variables that are being reimported again need to match the variables at the top of the shader like so

```
float _BlurRadius;
float _Intensity;
sampler2D _GrabTexture;
float4 _GrabTexture_TexelSize;
```

Notice the _GrabTexture and _GrabTexture_Texel size? We can import these without declaring them at the start as they are made from the grab pass and contain information about what’s behind the object.

3.10. Now for the vertex function “vert” I am going to show you what is should look like and then explain it underneath

```
v2f vert(appdata_base IN)
{
        v2f OUT;
```

```
        OUT.vertex = UnityObjectToClipPos(IN.vertex);

        #if UNITY_UV_STARTS_AT_TOP
                float scale = -1.0;
        #else
                float scale = 1.0;
        #endif

        OUT.uvgrab.xy = (float2(OUT.vertex.x, OUT.vertex.y * scale) + OUT.vertex.
w) * 0.5;
        OUT.uvgrab.zw = OUT.vertex.zw;

        return OUT;
}
```

So, we set our return variable at the start, then we set the vertices so that they are in the cameras view. Where the code changes the uvgrab of OUT it is refining what is shown on the faces of the object as if we didn't have those lines the whole camera view would be projected onto each face of the object instead of just what's behind it. The UNITY_UV_STARTS_AT_TOP part is added to change the scale we use depending on if you use DirectX or have other software and hardware variant's due to these alternatives reading UV's differently.

3.11. To end the first pass, we are going to change the fragment function to be the following

```
half4 frag(v2f IN) : COLOR
{
        half4 texcol = tex2Dproj(_GrabTexture, UNITY_PROJ_COORD(IN.uvgrab));
        half4 texsum = half4(0, 0, 0, 0);


        #define GRABPIXEL(weight, kernelx) tex2Dproj(_GrabTexture, UNITY_PROJ_COO
RD(float4(IN.uvgrab.x + _GrabTexture_TexelSize.x * kernelx * _BlurRadius, IN.uvgra
b.y,  IN.uvgrab.z,  IN.uvgrab.w))) * weight

        texsum += GRABPIXEL(0.05, -4.0);
        texsum += GRABPIXEL(0.09, -3.0);
        texsum += GRABPIXEL(0.12, -2.0);
        texsum += GRABPIXEL(0.15, -1.0);
        texsum += GRABPIXEL(0.18, 0.0);
        texsum += GRABPIXEL(0.15, 1.0);
        texsum += GRABPIXEL(0.12, 2.0);
        texsum += GRABPIXEL(0.09, 3.0);
        texsum += GRABPIXEL(0.05, 4.0);

        texcol = lerp(texcol, texsum, _Intensity);
        return texcol;
}
```

The first part of the function we set up two colours texcol and texsum. texcol is equal to everything the grab pass got refined down to just things behind the object (well one pixel of that anyway, as the shader goes through all the pixels like an array). texsum is just a white colour with no alpha. The basic jist of the function is for each pixel grab other pixels on the horizontal axis' colours and add a certain amount(weight) of that colour to texsum. Thus, making texsum equal to an averaged colour of the pixels around it (blurred). The

GRABPIXEL definition is basically an equation of a Gaussian blur function in one dimension which is G(X) = (1/(sqrt(2*PI*deviation*deviation))) * exp(-(x * x / (2 * deviation* deviation))) implemented into Unity and shader code. So, after we add our pixels colours together (based on how far away they are from our original pixel and the weight we set them) we can lerp between the ordinal colour of the pixel and the blurred colour based on intensity with the one before last line of code. We then return this colour.

3.12. Now that our horizontal blur pass is complete let's replace the pass beneath it with the content from the horizontal pass so that we end up with a shader that has two passes that both horizontally blur.

3.13. Change the name of the second pass to be VERTICALBLUR like so

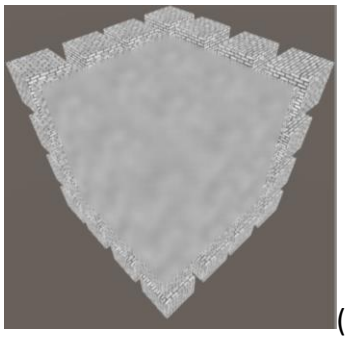
```
Pass
{
        Name "VERTICALBLUR"
```

3.14. Go to the fragment function and replace the GRABPIXEL function with this one

```
#define GRABPIXEL(weight, kernely) tex2Dproj(_GrabTexture, UNITY_PROJ_COORD(f
loat4(IN.uvgrab.x, IN.uvgrab.y + _GrabTexture_TexelSize.y * kernely * _BlurRa
dius,  IN.uvgrab.z,  IN.uvgrab.w))) * weight
```

We do this to change the grabbing of pixels from the horizontal axis to the vertical axis.

3.15. We have made a blur shader! To test the shader go into Unity and make a new material named Blur and drag the shader onto it then put that material on the cube. You may also want to put a cube or two behind it, so you can see the full effect. You can then mess around with the radius and intensity in the inspector. You should end up with something like this.

 (My blur radius is 3.2 and the intensity is 1 to show off the effect)

4. Distort

4.1. Moving swiftly forwards in the delve of the world of shaders we will start to make a distortion shader which gets everything behind the object and offsets them according to a normal map along with applying a tint. To start duplicate blur shader and name this one Distort.

4.2. Open the shader in visual studio by double clicking on it in the project asset view panel.

4.3. Replace the name of Blur with Distort like so

```
Shader "Custom/Distort"
```

4.4. We are going to need change the properties so that we can input a colour to tint the distortion, a bump amount as to change the intensity of the distortion, a distort map for if we ever wanted to add a texture to the distortion and a normal map field of which the offsetting will use. This should end up like

```
Properties//Variables
        {
                _DistortColor("Distort Color", Color) = (1,1,1,1)
                _BumpAmt("Distortion", Range(0,128)) = 10
                _DistortTex("Distort Texture (RGB)", 2D) = "white" {}
                _BumpMap("Normal Map", 2D) = "bump" {}
        }
```

4.5. The second pass currently named VERTICALBLUR and the grab pass before if can both be removed.

4.6. Rename the remaining pass DISTORT like so

```
Name "DISTORT"
```

4.7. We are also going to need to add in appdata again as the base won't cut it for this pass so go ahead and add the code below underneath the include.

```
struct appdata//How the vertex function receives info.
{
        float4 vertex : POSITION;
        float2 texcoord : TEXCOORD0;
};
```

4.8. Change the vert function so it uses this newly created appdata instead of the base like so

```
v2f vert(appdata IN)
```

4.9. The v2f struct also needs a bit of editing to accommodate for the extra textures so make the v2f struct look like this

```
struct v2f
{
        float4 vertex : SV_POSITION;
        float4 uvgrab : TEXCOORD0;
        float2 uvbump : TEXCOORD1;
        float uvmain : TEXCOORD2;
};
```

4.10. The reimports also need to consider the new properties so let's add them so it looks like this

```
float _BumpAmt;
float4 _BumpMap_ST;
float4 _DistortTex_ST;
fixed4 _DistortColor;
sampler2D _GrabTexture;
float4 _GrabTexture_TexelSize;
ampler2D _BumpMap;
sampler2D _DistortTex;
```

4.11. The vertex function is almost correct it just needs an extra two lines between the return and the line above it that will allow the uv's of the textures for the distortion to be passed into the fragment function like so

```
v2f vert(appdata IN)
{
        v2f OUT;
        OUT.vertex = UnityObjectToClipPos(IN.vertex);

        #if UNITY_UV_STARTS_AT_TOP
                float scale = -1.0;
        #else
                float scale = 1.0;
        #endif

        OUT.uvgrab.xy = (float2(OUT.vertex.x, OUT.vertex.y * scale) + OUT.vertex.
w) * 0.5;
        OUT.uvgrab.zw = OUT.vertex.zw;

        OUT.uvbump = TRANSFORM_TEX(IN.texcoord, _BumpMap);
        OUT.uvmain = TRANSFORM_TEX(IN.texcoord, _DistortTex);

        return OUT;
}
```

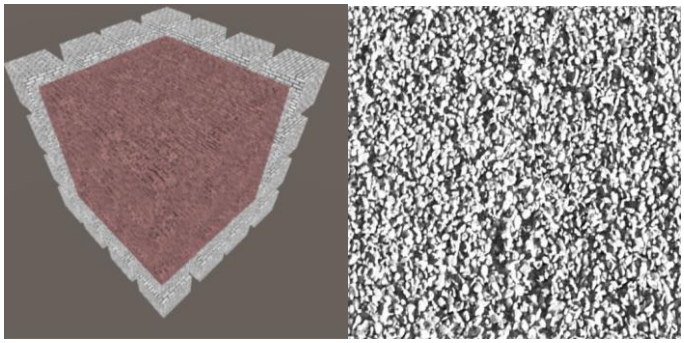
4.12. Now for the meat of this shader the fragment function! It should look like this

```
half4 frag(v2f IN) : COLOR
{
        half2 bump = UnpackNormal(tex2D(_BumpMap, IN.uvbump)).rg;
        float2 offset = bump * _BumpAmt * _GrabTexture_TexelSize.xy;
        IN.uvgrab.xy = offset * IN.uvgrab.z + IN.uvgrab.xy;

        half4 col = tex2Dproj(_GrabTexture, UNITY_PROJ_COORD(IN.uvgrab));
        half4 tint = tex2D(_DistortTex, IN.uvmain) * _DistortColor;

        return col * tint;
}
```

What this function does is that it calculates an offset for the grabbed texture (what is behind the object) and applies it to the uv's of said grabbed texture making the image look

distorted if a bump map is applied. After distorting everything behind the object it multiplies the colour of the distortion by the colour property to give it a tint.

4.13. Now for a test! Again, go into Unity and create a new material called Distort and drag the shader onto it. Put the material on the cube and experiment with different normal maps to see the different distortions. You can get results like the one below by using a light red as the tint, no texture and the normal map next to the result.



5. Final Shader (Outline with blur and distort)

5.1. The final shader will be a mix of all the shaders we just made however the blur and distort shaders currently only blur distort the shape of the box when we want them to distort and blur the size of the border. Thus, we must duplicate those two shaders and rename OutlineDistort and OutlineBlur respectively.

5.2. Rename the shaders at the top so that they look like

```
Shader "Custom/OutlineBlur"
```

And

```
Shader "Custom/OutlineDistort"
```

5.3. In both of there property fields add

```
_OutlineWidth("Outline Width", Range(1.0,10.0)) = 1.1
```

5.4. Rename both passes in the blur and the one pass in the distort to have OUTLINE in front like so

```
Name "OUTLINEDISTORT"
```

```
Name "OUTLINEHORIZONTALBLUR"
```

```
Name "OUTLINEVERTICALBLUR"
```

5.5. In every pass of these two shaders turn off ZWriting by adding the line below just beneath each of the passes names.

```
ZWrite Off
```

5.6. To make the distort cover the whole outline we add the line below to the top of the vertex function of the OutlineDistort shader.

```
IN.vertex.xyz *= _OutlineWidth;
```

5.7. Now we want the blur to not only cover the outline but exceed it a little bit so that the edges of the outline blur with the background, so we add the line below to the top of the vertex function.

```
IN.vertex.xyz *= _OutlineWidth + 0.1;
```

5.8. Finally, time to set up our final shader! Make a new Unlit Shader In Unity or duplicate one of the others and name it Final Outline or whatever name you see fit.

5.9. Change the name inside the shader to match the one you gave it and change the folder name to match the others like so.

```
Shader "Custom/FinalOutline"
```

5.10. In the properties field we need to add all previous properties used apart from outline colour and texture and we are using the variables from the distort instead. Your code should look like this

```
Properties//Variables
        {
                _MainTex("Main Texture (RBG)", 2D) = "white" {}
                _Color("Color", Color) = (1,1,1,1)
                _OutlineWidth("Outline Width", Range(1.0,10.0)) = 1.1

                _BlurRadius("Blur Radius", Range(0.0,20.0)) = 1
                _Intensity("Blur Intensity", Range(0.0,1.0)) = 0.01

                _DistortColor("Distort Color", Color) = (1,1,1,1)
                _BumpAmt("Distortion", Range(0,128)) = 10
                _DistortTex("Distort Texture (RGB)", 2D) = "white" {}
                _BumpMap("Normal Map", 2D) = "bump" {}
        }
```

5.11. In the SubShader we don't have any passes so delete any you may have in there but keep the tags like so

```
Tags
{
        "Queue" = "Transparent"
```

```
}
```

5.12. Now underneath the tags we need to call all our passes we want to use so we start with a grab pass to get everything behind the object

```
GrabPass{}//Grabs everything behind the object.
```

Then we apply the distortion pass by using UsePass

```
UsePass "Custom/OutlineDistort/OUTLINEDISTORT"
```

Then we grab this output with another grab pass

```
GrabPass{}//Grabs the distorted pass.
```

We then apply the horizontal blur

```
UsePass "Custom/OutlineBlur/OUTLINEHORIZONTALBLUR"
```

And grab its output with

```
GrabPass{}//Grabs the horizontally blurred distortion.
```

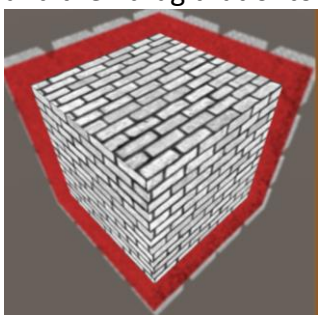
Then apply the vertical blur

```
UsePass "Custom/OutlineBlur/OUTLINEVERTICALBLUR"
```

And finally, we render the shape without a grab pass as we need fresh inputs

```
UsePass "Custom/Outline/OBJECT"//Renders the object.
```

5.13. With that this journey comes to an end and we can reap the rewards of our shader programming skills. Create a material called FinalOutline and drag this shader onto it and then drag that onto the cube to see the effect. You will see something like this



5.14. To really show off our shader we can add a script to the object that makes it so when the mouse is hovering over an object it uses a different material. We can use the apply texture material as the non-hovered material and the final border as the hovered. To do this create a new C# script in Unity and name it OnHoverBorder.

Put this code in the script

```
using UnityEngine;
```

```
public class OnHoverBorder : MonoBehaviour
{
        public Material border;
        public Material nonBorder;

        void OnMouseOver()
        {
                GetComponent<Renderer>().material = border;
        }
        void OnMouseExit()
        {
                GetComponent<Renderer>().material = nonBorder;
        }
}
```

This code changes the renderers material for the object when the mouse is over and exits the object. So, drag this script on all objects you want it to effect and plug in the materials you have made, press play and let the magic happen.

Congratulations you now have a nice looking shader to play around with in your games perhaps in games like point and click adventures or detective games.