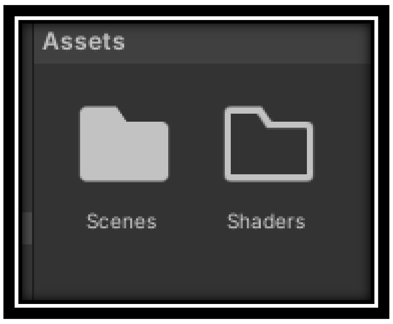
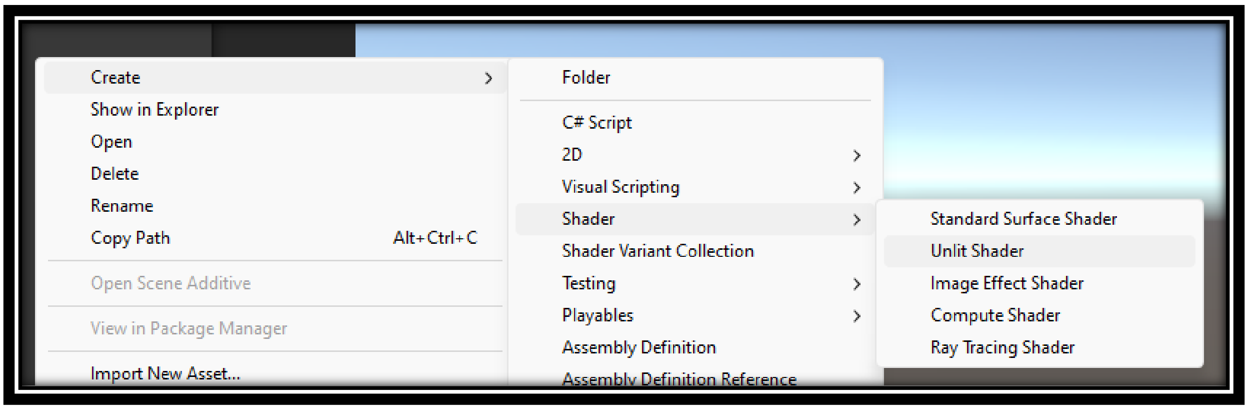
# DX1219: PRACTICAL 1

**The aim of this practical is to create a basic vertex and fragment shader from scratch in Unity.**

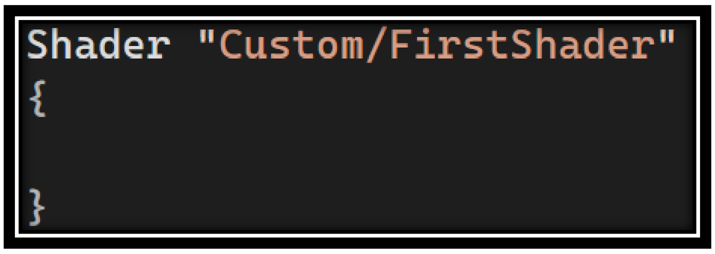
1. Create a 3D Project.



2. Create a folder name **Shaders**
3. Create an **Unlit Shader** in the Shaders folder

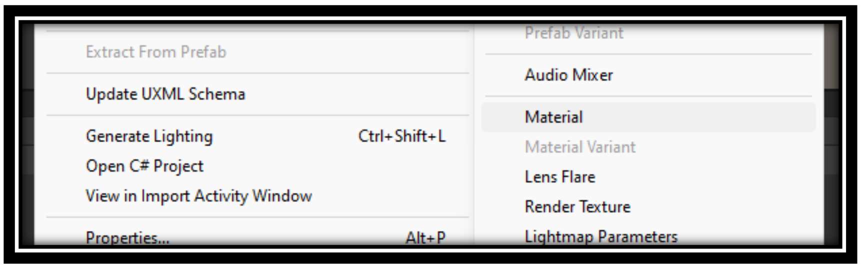


4. Open the file up and empty it like that

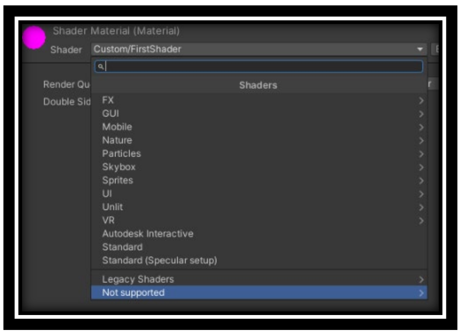


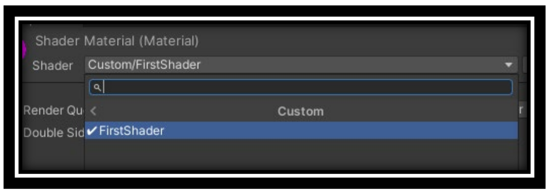
```
Shader "Custom/FirstShader"
{

}
```

5. Save the file. It will not work as the shader is not supported. Regardless test it out first by assign it to a **material**. Create a new **material** into a **Materials** folder.

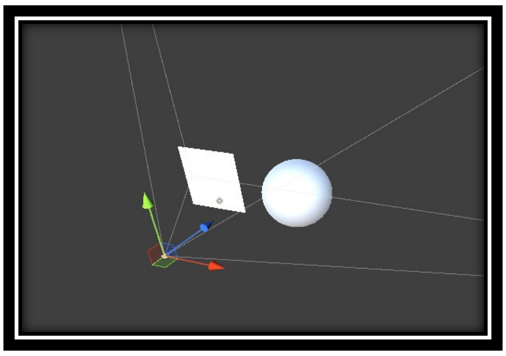


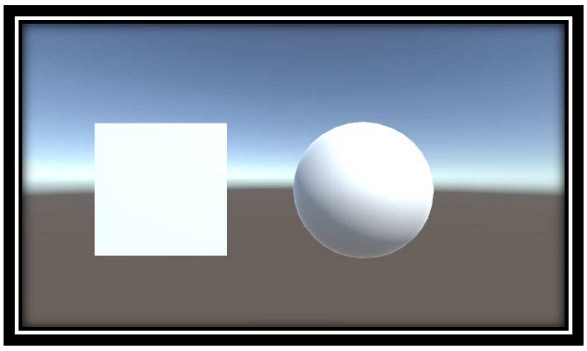
6. Change the shader in the material.



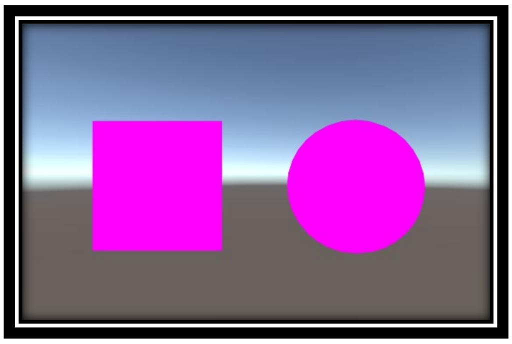


7. Create a plane and rotate it to face the camera. Create a sphere as well.

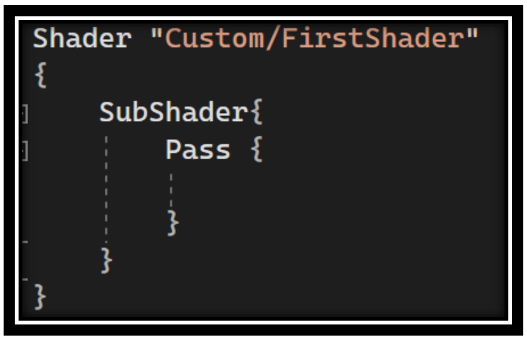




8. Attach that material to the sphere and plane by dragging the material to the objects so it will look something like this as the shader contains errors.

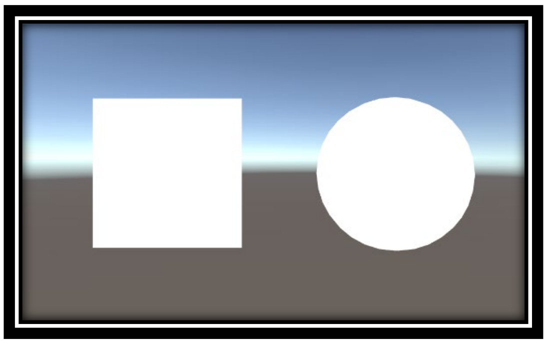


9. To allow our shader to work, we have to add sub-shaders. Unity allows us to provide different sub-shaders for different build platforms or levels of detail. For example, you could have one sub-shader for desktops and another for mobiles. We need just one sub-shader block by default. Each sub shader needs to contain at least one pass. A shader pass is where an object gets rendered. By default, we need one pass but we have more than one in the future which also means the object get rendered more than once. Change the shader file to the following and save the file.

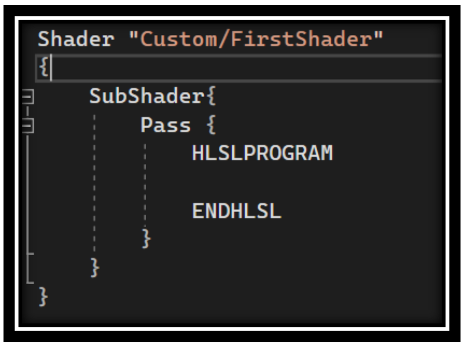


```
Shader "Custom/FirstShader"
{
    SubShader{
        Pass {

        }
    }
}
```

10. Check the scene and you should see something like this.

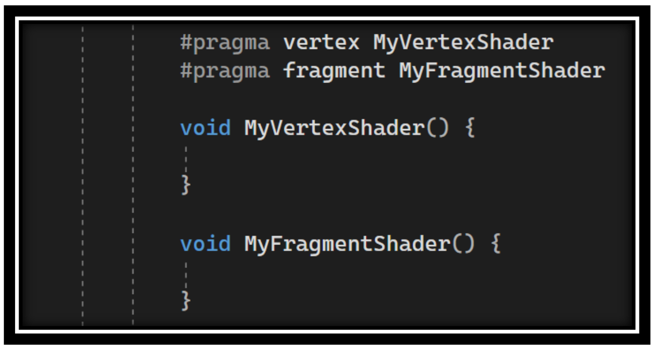


11. Let start writing the real shader. We have using Unity's Shading Language which is a variant of HLSL and CG shading language. We have to start and end the code with something like that.

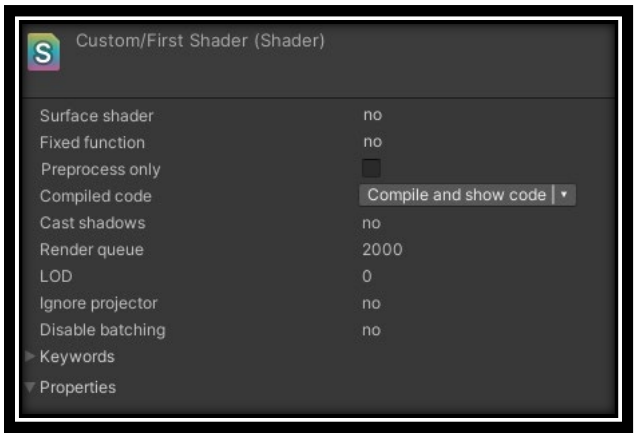


```
Shader "Custom/FirstShader"
{
    SubShader{
        Pass {
            HLSLPROGRAM

            ENDHLSL
        }
    }
}
```

12. Let add the vertex and fragment shader. Save the file and click Compile and show code.



```
            #pragma vertex MyVertexShader
            #pragma fragment MyFragmentShader

            void MyVertexShader() {

            }

            void MyFragmentShader() {

            }
```

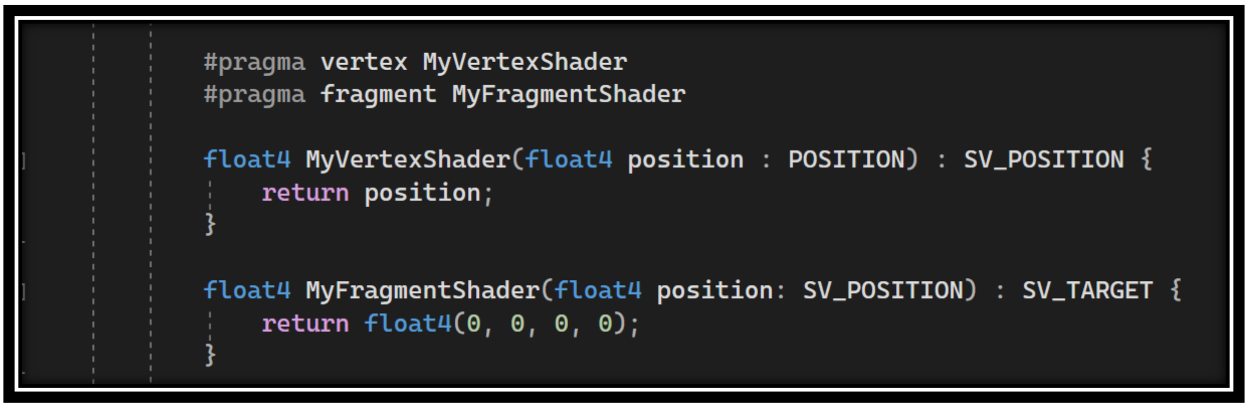


Custom/First Shader (Shader)

| | |
|---|---|
| Surface shader | no |
| Fixed function | no |
| Preprocess only | |
| Compiled code | Compile and show code |
| Cast shadows | no |
| Render queue | 2000 |
| LOD | 0 |
| Ignore projector | no |
| Disable batching | no |

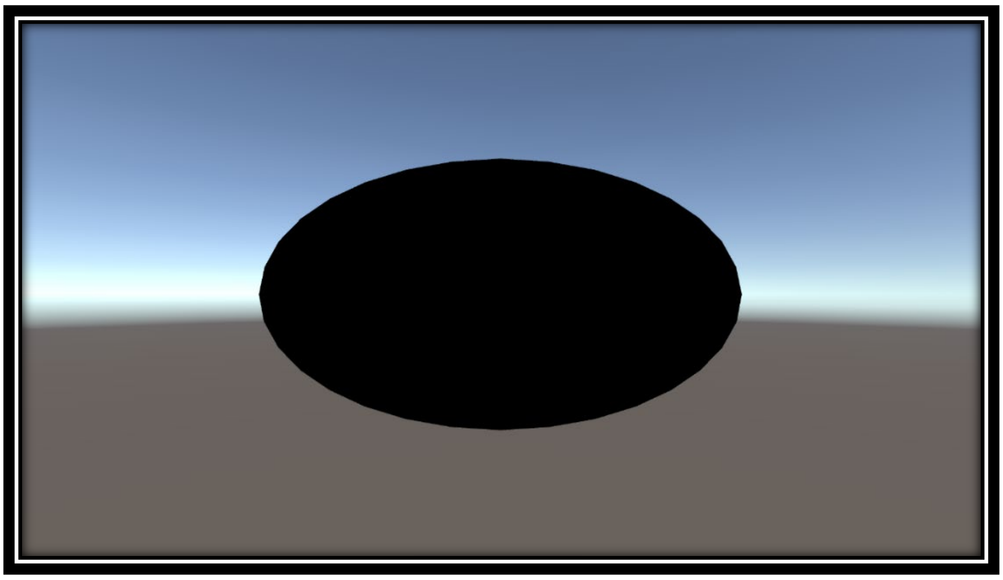
13. After the shader compile, you will notice the objects disappear. If you seed an error, it might be what was the platform that you have set for. Change the type of build platform or shader type output to fit the current build platform if needed.

14. Update your shader to the following. We are now adding return type to the vertex and fragment program. We are also adding a parameter to the shader programs. We have to define what the output of the programs so that the GPU know how to handle it.  Since our output of our vertex shader is the position of the vertex so we indicate this by writing SV_POSITION which refers to the System Value Final Vertex Position. SV_Target refers to the default shader target which is the frame buffer.

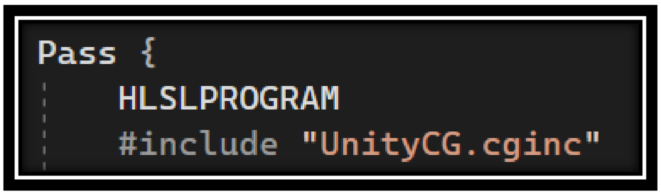


```
#pragma vertex MyVertexShader
#pragma fragment MyFragmentShader

float4 MyVertexShader(float4 position : POSITION) : SV_POSITION {
    return position;
}

float4 MyFragmentShader(float4 position: SV_POSITION) : SV_TARGET {
    return float4(0, 0, 0, 0);
}
```

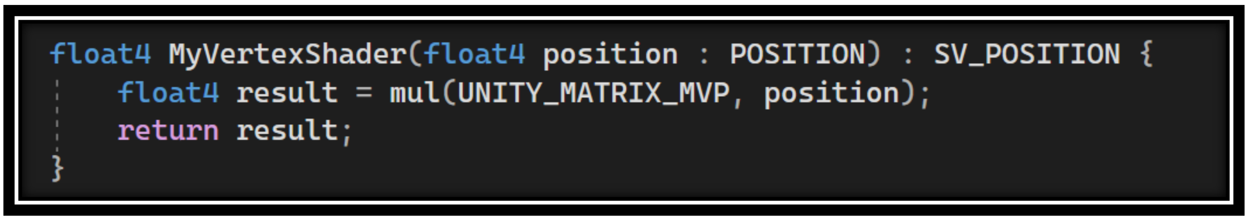
15. Save the shader file and you should see something like this. A black sphere will appear but it is slightly distorted. This is due to the fact we are in object / local space position. Moving the objects will make differences due to the shader. So we have to bring the object into screen space instead.



16. We will mulitply the raw vertex position with the model view projection matrix. Which bring the object from local space to world space then screen space using the camera transformation and projection. Change the vertex program to this. In order to use UNITY_MATRIX_MVP, we need to include load other files that contains its definition. If you see some tutorial online because they used CGPROGRAM rather than HLSLPROGRAM.



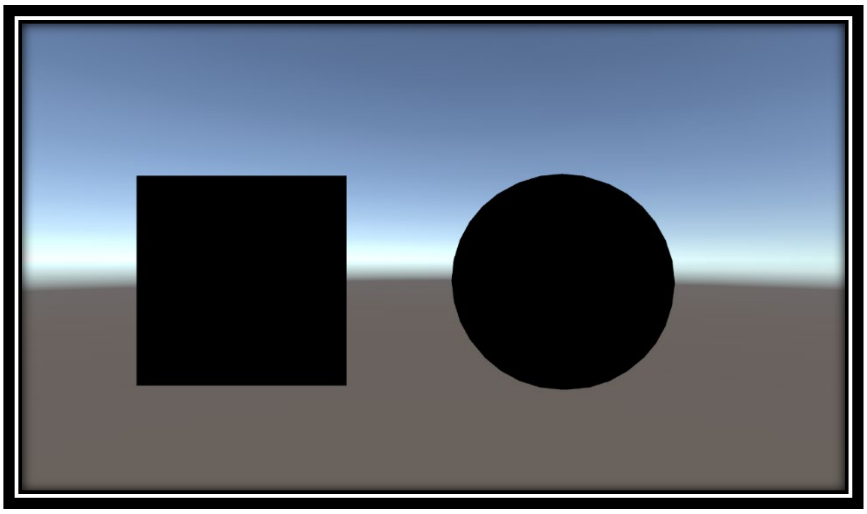
```
Pass {
    HLSLPROGRAM
    #include "UnityCG.cginc"
```



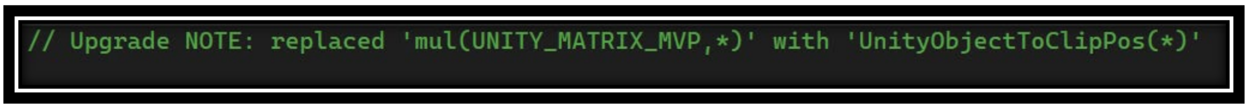
```
float4 MyVertexShader(float4 position : POSITION) : SV_POSITION {
    float4 result = mul(UNITY_MATRIX_MVP, position);
    return result;
}
```

CGPROGRAM automatically included "UnityCG.cginc".

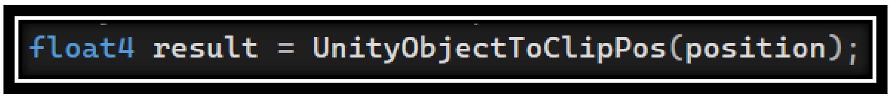
17. You should see the objects in the correct position now.



18. Some of your might notice that Unity helped you upgrade the code automatically.

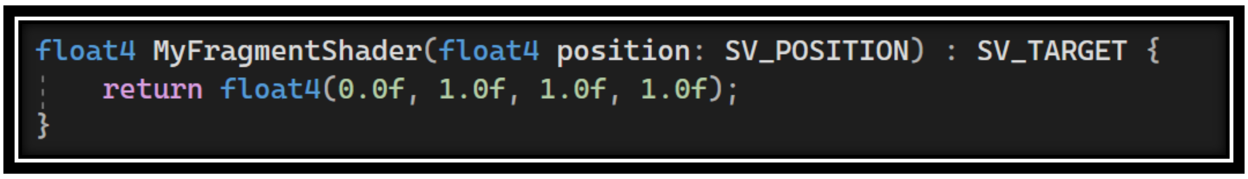


```
// Upgrade NOTE: replaced 'mul(UNITY_MATRIX_MVP,*)' with 'UnityObjectToClipPos(*)'
```

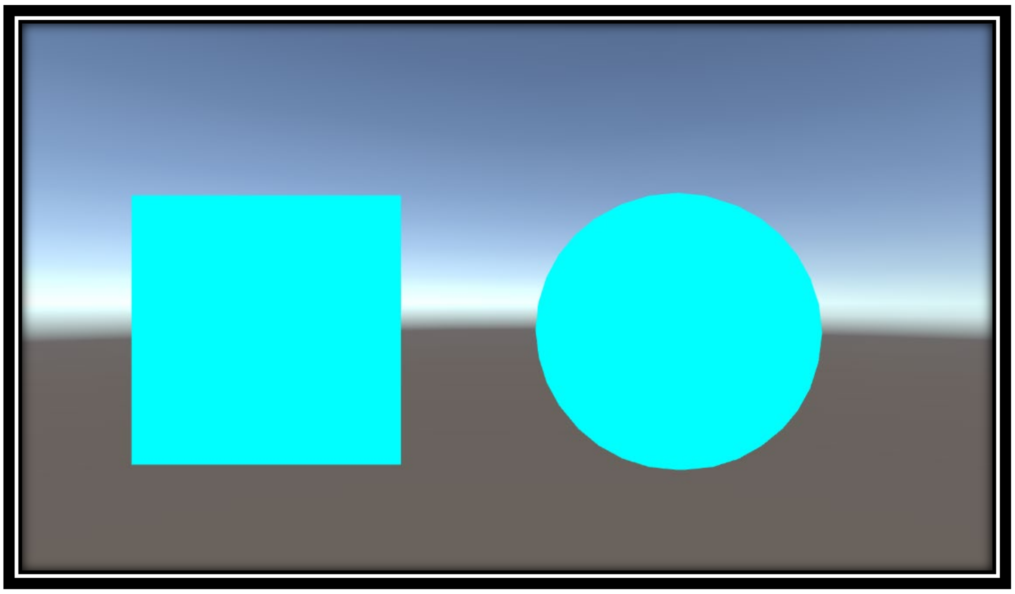


```
float4 result = UnityObjectToClipPos(position);
```

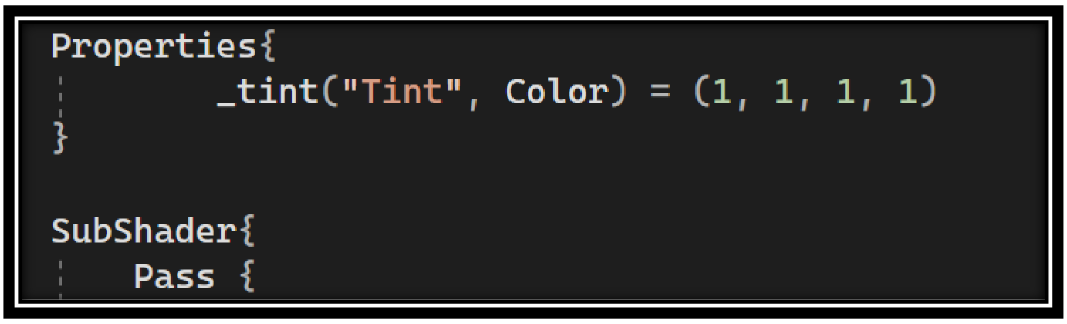
19. Let try change the object to cyan by changing the fragment shader. Play around the color if you like.



```
float4 MyFragmentShader(float4 position: SV_POSITION) : SV_TARGET {
    return float4(0.0f, 1.0f, 1.0f, 1.0f);
}
```

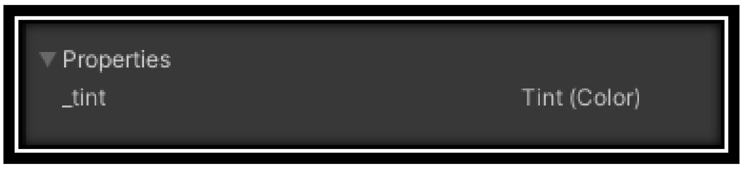


20. At this point, you have written a basic vertex and fragment shader. However, how do we write a shader that could support any color that we set through the material. We need to create via the shader properties. We need to define what type of it is. _tint is the property name in the shader. "Tint" is the property name in Unity Material inspector and Color is the type.



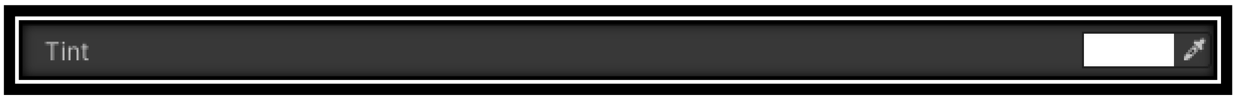
```
Properties{
        _tint("Tint", Color) = (1, 1, 1, 1)
}

SubShader{
    Pass {
```

You will see it in the shader properties.



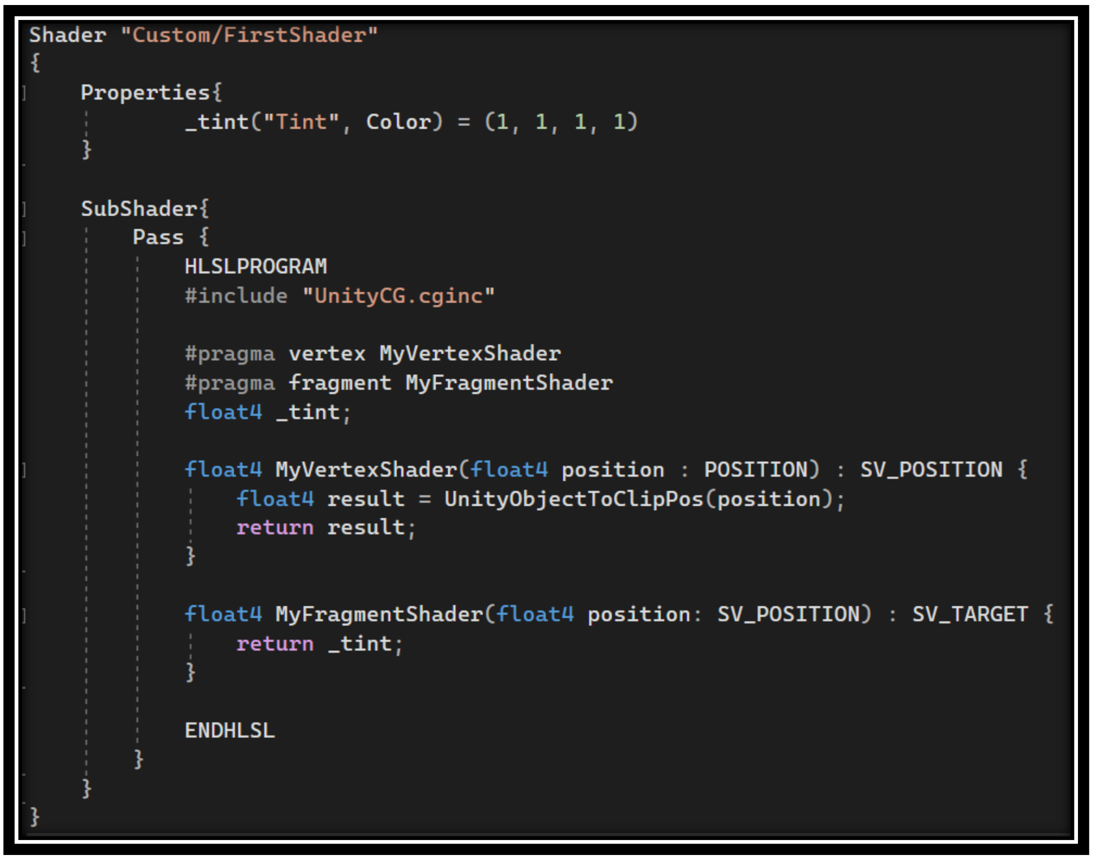
```
▼ Properties
  _tint                          Tint (Color)
```

You will see this in the material where you can change the color up to you.



```
Tint
```

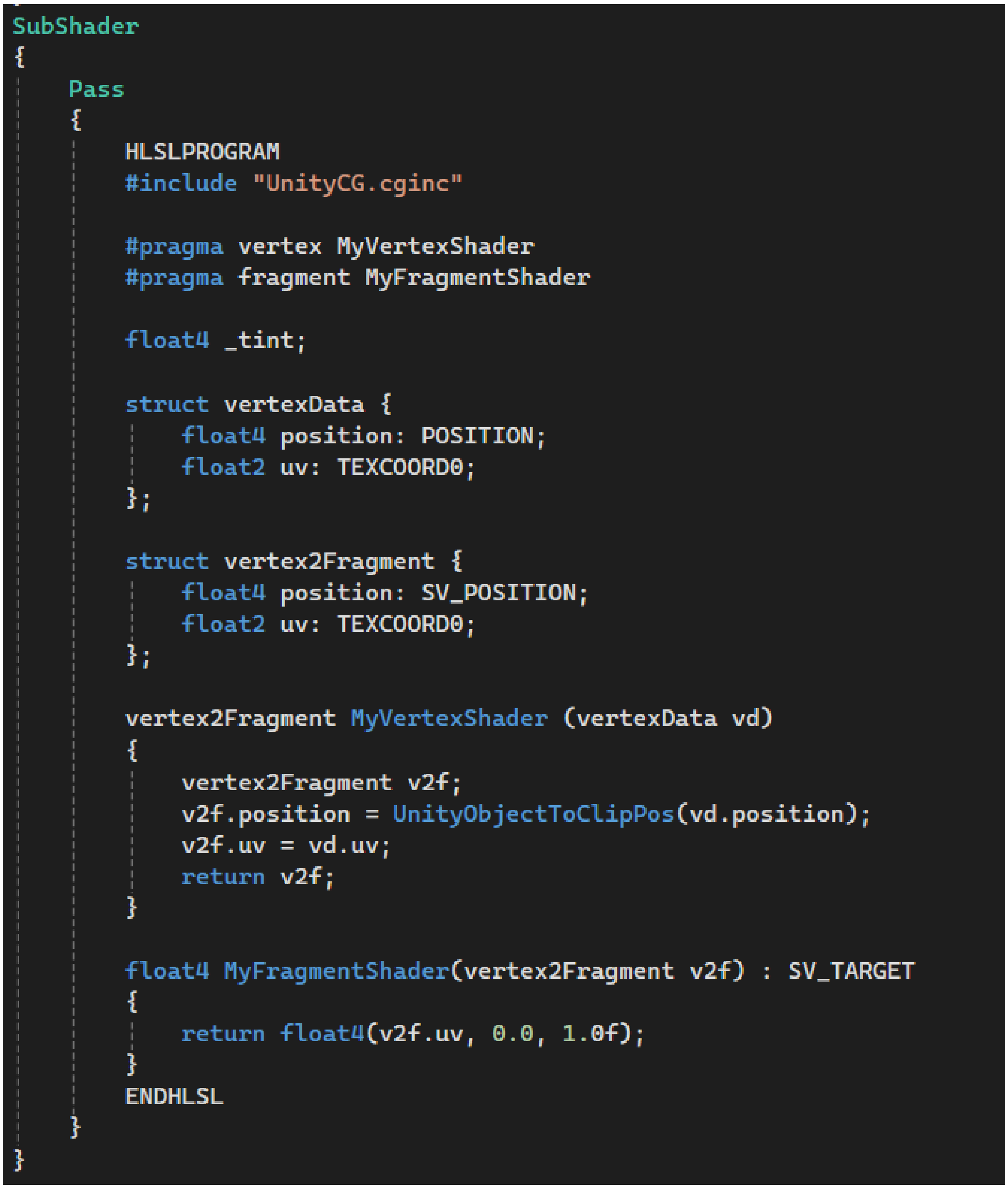
21. Update the Pass to the following. Save the file and try it out. It should allow you to change it to any color that you have set in the Tint properties.



```
Shader "Custom/FirstShader"
{
    Properties{
            _tint("Tint", Color) = (1, 1, 1, 1)
    }

    SubShader{
        Pass {
            HLSLPROGRAM
            #include "UnityCG.cginc"

            #pragma vertex MyVertexShader
            #pragma fragment MyFragmentShader
            float4 _tint;

            float4 MyVertexShader(float4 position : POSITION) : SV_POSITION {
                float4 result = UnityObjectToClipPos(position);
                return result;
            }

            float4 MyFragmentShader(float4 position: SV_POSITION) : SV_TARGET {
                return _tint;
            }

            ENDHLSL
        }
    }
}
```

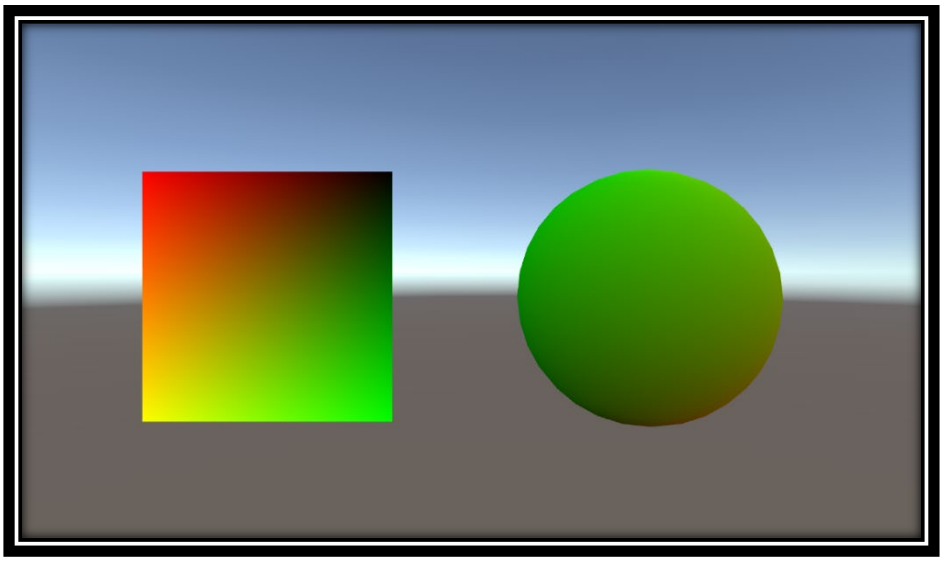
22. We have really utilised the fragment shader other than directly using the color. As we have discussed before that fragment shader will automatically do interpolation for us after the rasterization process. Now let try using the **uv** to display the **color** instead.

    We can create struct in the shader file as well. We created a 2 struct (**vertexData** , **vertex2Fragment**). **vertexData** are data that is send from the application to the **vertex shader**. Notice thatThen we are sending **vertex2Fragment** from the vertex shader to the **fragment shader** and using the uv variable to create the interpolated color.



```
SubShader
{
    Pass
    {
        HLSLPROGRAM
        #include "UnityCG.cginc"

        #pragma vertex MyVertexShader
        #pragma fragment MyFragmentShader

        float4 _tint;

        struct vertexData {
            float4 position: POSITION;
            float2 uv: TEXCOORD0;
        };

        struct vertex2Fragment {
            float4 position: SV_POSITION;
            float2 uv: TEXCOORD0;
        };

        vertex2Fragment MyVertexShader (vertexData vd)
        {
            vertex2Fragment v2f;
            v2f.position = UnityObjectToClipPos(vd.position);
            v2f.uv = vd.uv;
            return v2f;
        }

        float4 MyFragmentShader(vertex2Fragment v2f) : SV_TARGET
        {
            return float4(v2f.uv, 0.0, 1.0f);
        }
        ENDHLSL
    }
}
```

23. This is how your scene will look at the end of the practical.



24. **EXTRA**: Try using the **tint** and the **UV** to combine to make such an effect.