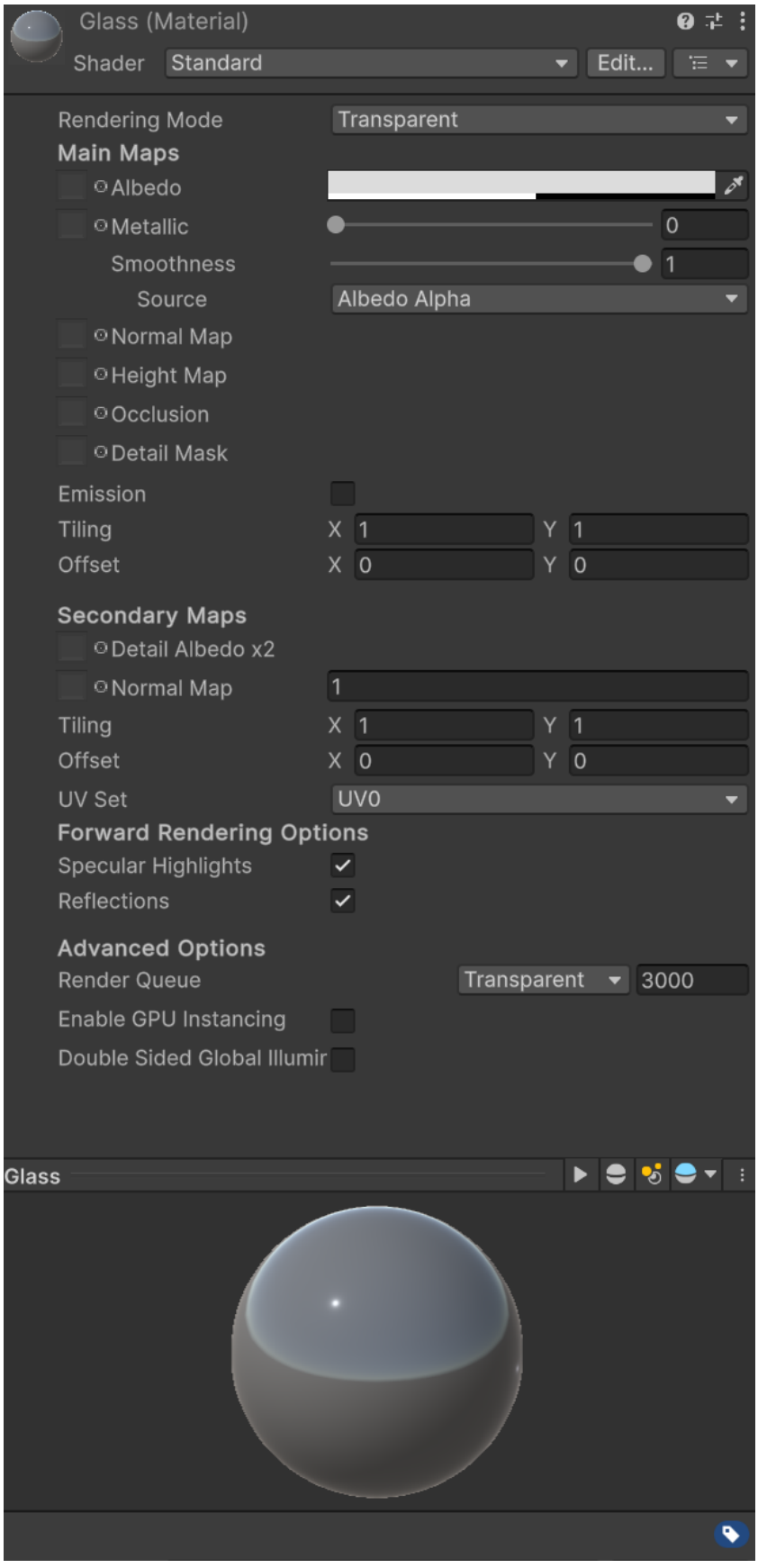
Read Me

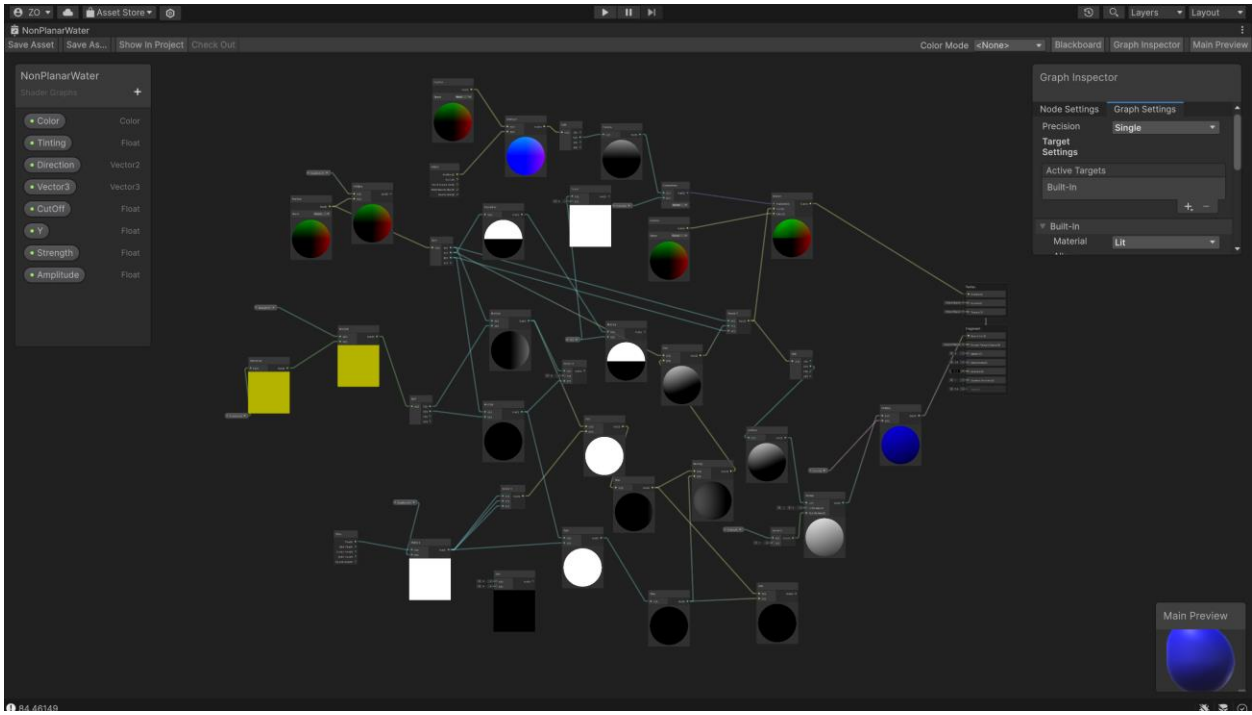
- Car
  - I made a simple script for the car that allows it to move forward and back and steer. It simply gets w and s key inputs and adds force to the front of the car and gets a and d input to change the rotation of the car.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

1 asset usage
public class Movement : MonoBehaviour
{
    [SerializeField] private float accelerationStrength = 5;  Unchanged
    [SerializeField] private float steeringStrength = 5;  "0.2"

    private float steeringInput;
    private float accelerationInput;
    private Rigidbody rb;
    Event function
    void Start()
    {
        rb = GetComponent<Rigidbody>();
    }

    // Update is called once per frame
    Event function
    void Update()
    {
        steeringInput = Input.GetAxis("Horizontal");
        accelerationInput = Input.GetAxis("Vertical");

        AddForce();
        AddSteering();
    }

    Frequently called  1 usage
    void AddForce()
    {
        Vector3 force;

        force = transform.forward * (accelerationInput * accelerationStrength);
        rb.AddForce(force, ForceMode.Force);
    }

    Frequently called  1 usage
    void AddSteering()
    {
        //transform.rotation = Quaternion.Euler(0, (transform.rotation.y * Mathf.Rad2Deg), 0);
        print(transform.rotation.eulerAngles.y);
        transform.rotation = Quaternion.Euler(0, transform.rotation.eulerAngles.y + (steeringInput * steeringStrength), 0);
    }
}
```

  - I wanted to make the car look like a fish tank so I added two shaders on it: a semi-transparent reflective shader used to simulate basic glass and a non-planar water shader that simulates waves. The glass uses unity's built-in shaders and is made by decreasing the alpha channel on a transparent shader and setting the smoothness to max. The water shader is a

modification of the water shader we did in class (though in shader graph). Unlike the class shader, this shader can work on objects that are not a plane as you can indicate what vertices you want the shader to affect. You can also change the direction and strength of the waves. It uses the same basic principles of the shader in class and oscillates the object's vertices based on sine waves. It ignores vertices below a threshold that you can set so that it does not affect the entire object. I did it this way so that you can view that water from a side angle like you would in a fish tank.


Glass (Material)
Shader
Standard
Edit...
Rendering Mode
Transparent
Main Maps
Albedo
Metallic
0
Smoothness
1
Source
Albedo Alpha
Normal Map
Height Map
Occlusion
Detail Mask
Emission
Tiling
X 1
Y 1
Offset
X 0
Y 0
Secondary Maps
Detail Albedo x2
Normal Map
1
Tiling
X 1
Y 1
Offset
X 0
Y 0
UV Set
UV0
Forward Rendering Options
Specular Highlights
Reflections
Advanced Options
Render Queue
Transparent
3000
Enable GPU Instancing
Double Sided Global Illumir
Glass

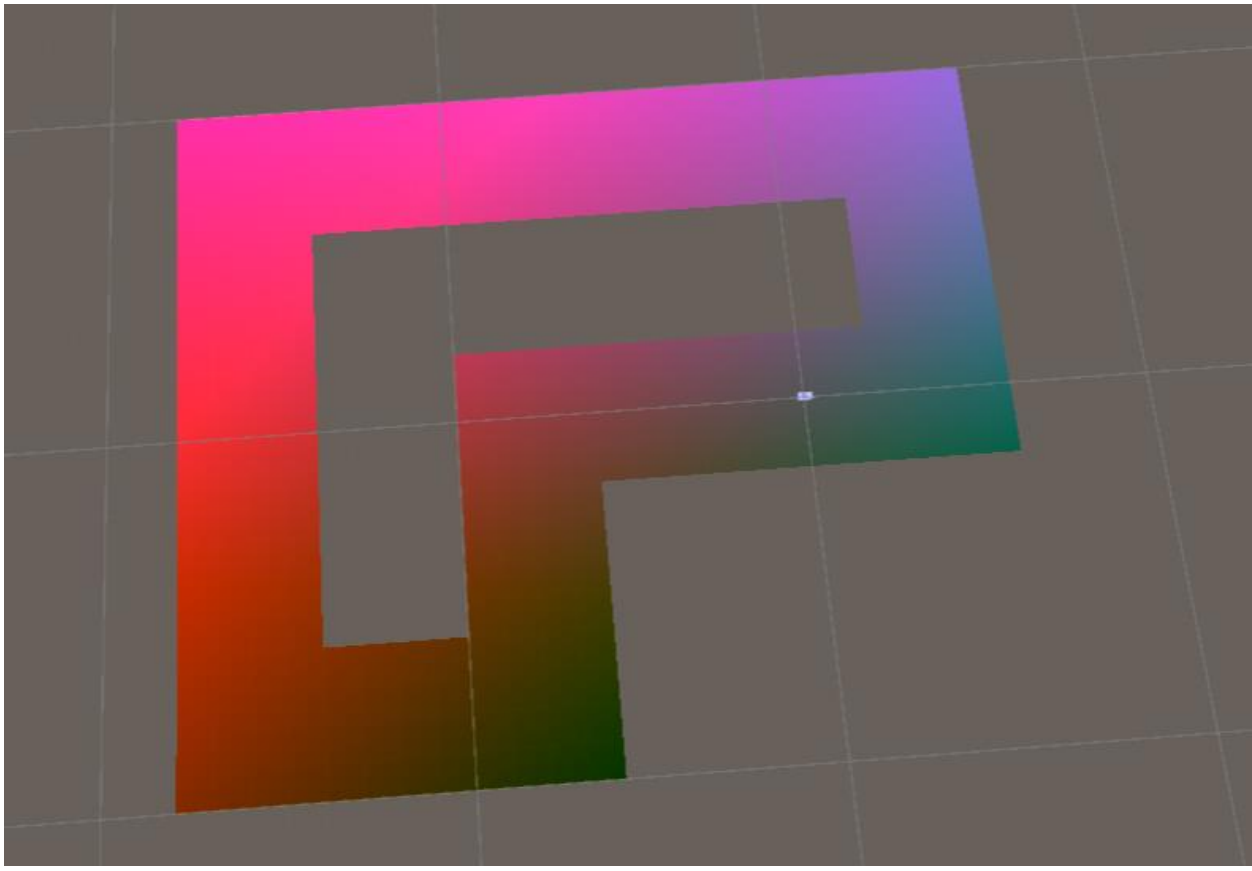


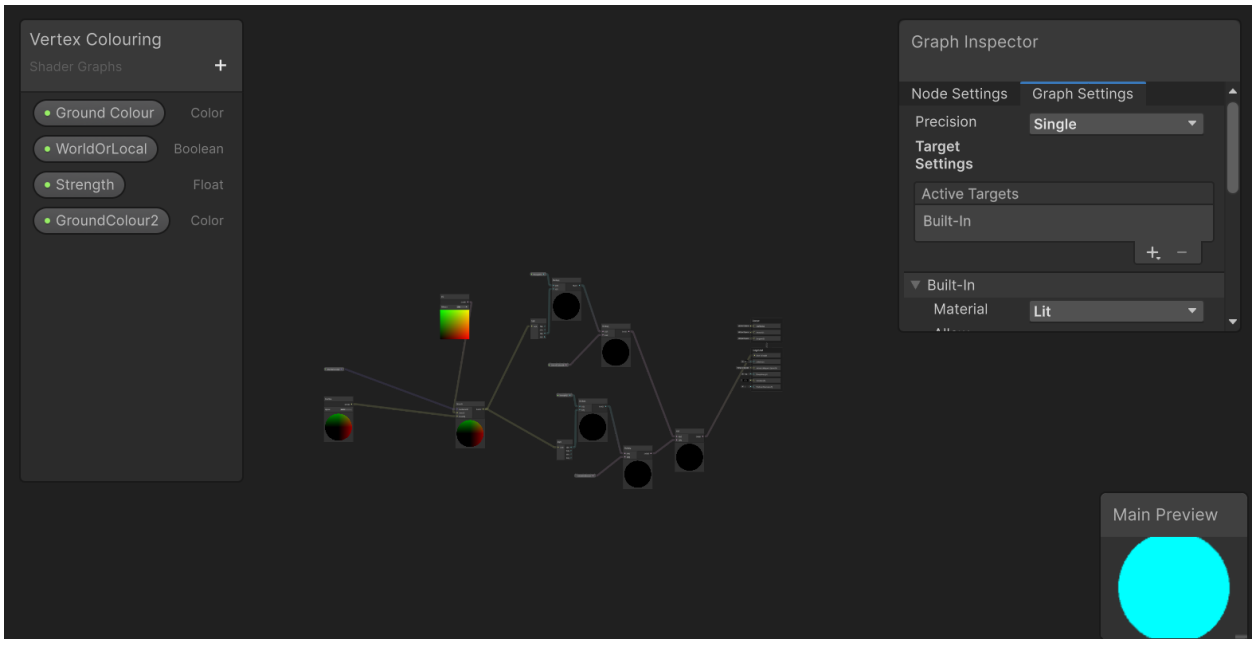

- Player
    - The player has a toon shader on it. The toon shader uses the same method we learned in class where it steps diffused lighting according to values sampled from a toon ramp. There is a slight modification so that you can control the strength of the toon ramp. I decided to add this to make the toon shading more subtle. I decided to use a toon shader for this as I wanted to go for a simple look on my player model so it fit well with the rest of the game.

```
Shader "Custom/ToonShader"
{
    Properties
    {
        _Color ("Color", Color) = (1,1,1,1 )
        _RampStrength ("_Ramp Strength", Float) = 1
        _RampTex ("Ramp Texture", 2D) = "white" {}
    }
    SubShader
    {
        CGPROGRAM
        // Physically based Standard lighting model, and enable shadows on all light types
        #pragma surface surf ToonRamp

        float4 _Color;
        float _RampStrength;
        sampler2D _RampTex;

        float4 LightingToonRamp (SurfaceOutput s, fixed3 lightDir, fixed atten)
        {
            float diff = dot (s.Normal, lightDir);
            float h = diff * 0.5 + 0.5;
            float2 rh = h;
            float3 ramp = tex2D(_RampTex, rh).rgb * _RampStrength;

            float4 c;
            c.rgb = s.Albedo * _LightColor0.rgb * (ramp);
            c.a = s.Alpha;
            return c;
        }

        struct Input
        {
            float2 uv_MainTex;
        };

        void surf (Input IN, inout SurfaceOutput o)
        {
            o.Albedo = _Color.rgb;
        }
        ENDCG
    }
}
```

- Road
    - The road uses vertex colouring. My first iteration of this used the same method we learned in class where you sampled the local position of the vertices and multiplied it by a colour but I noticed an issue when I placed multiple road tiles they wouldn't line up. So I changed my approach to use

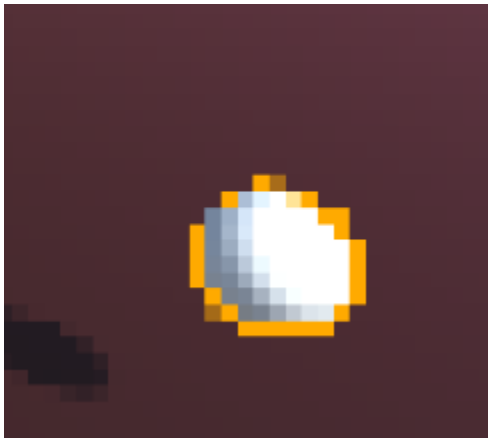
the global vertex positions so that I could combine multiple planes smoothly together. I also add a value that allowed you to adjust the strength of the gradient as I found it was far too steep. I used this technique as I wanted my ground to have a similarly colourful ground like reference image.






- Coins

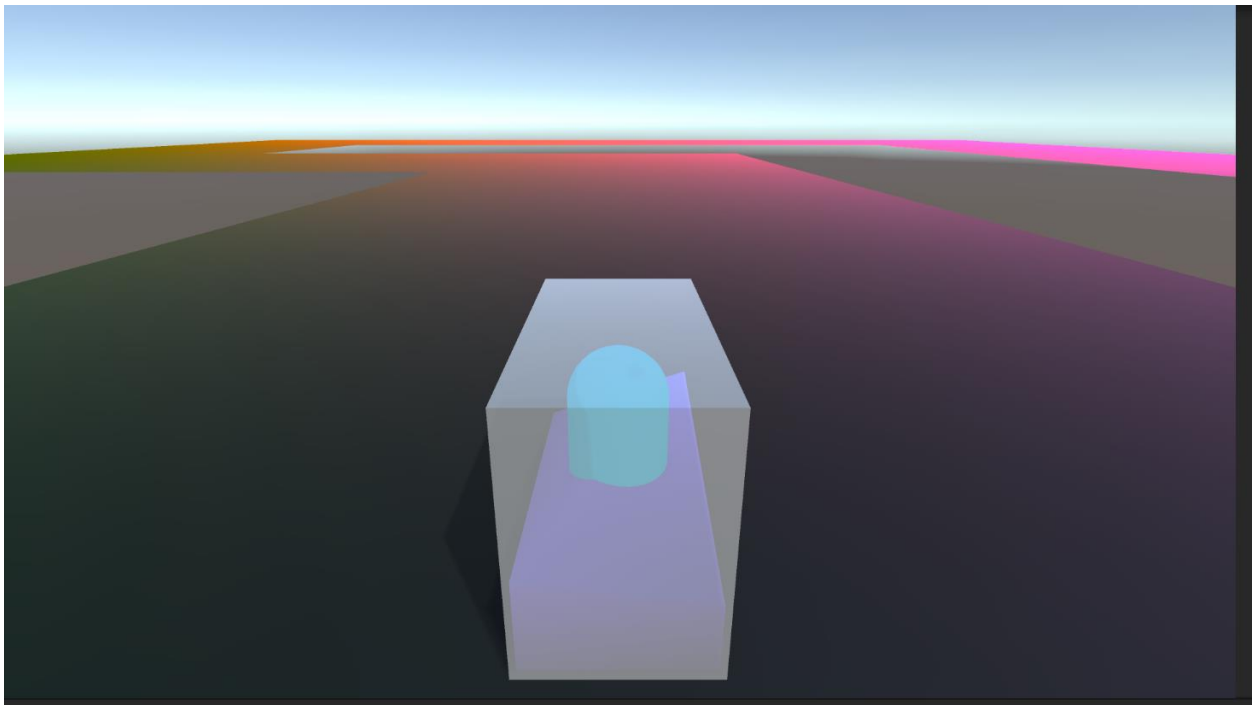
- I added spheres with an outline shader to replicate coins in Mario cart. Adding an outline shader helps give it a 2D look while actually being 3D. I used the more complicated outline shader we used in class to create this effect.



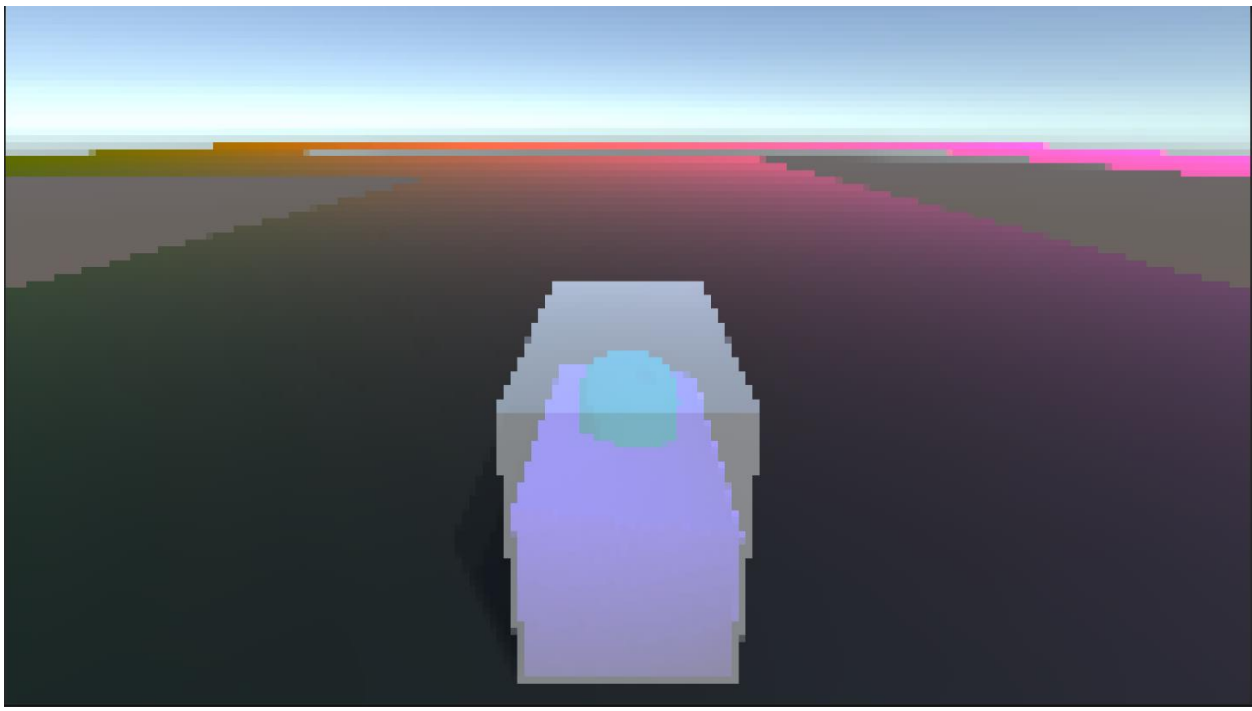
```
Shader "Custom/Outline"
{
    Properties {
    _MainTex ("Texture", 2D) = "white" {}
    _OutlineColor ("Outline Color", Color) = (0,0,0,1 )
    _Outline ("Outline Width", Range (0.002, 0.1)) = 0.005
}

SubShader {
    CGPROGRAM
    #pragma surface surf Lambert
    struct Input {
        float2 uv_MainTex;
    };

    sampler2D _MainTex;
    void surf (Input IN, inout SurfaceOutput o) {
        o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
    }
    ENDCG

    Pass {
    Cull Front

    CGPROGRAM
    #pragma vertex vert
    #pragma fragment frag

    #include "UnityCG.cginc"

    struct appdata {
        float4 vertex : POSITION;
        float3 normal : NORMAL;
    };

    struct v2f {
        float4 pos : SV_POSITION;
        fixed4 color : COLOR;
    };

    float _Outline;
    float4 _OutlineColor;

    v2f vert (appdata v) {
        v2f o;
        o.pos = UnityObjectToClipPos(v.vertex);

        float3 norm = normalize(mul((float3x3)UNITY_MATRIX_IT_MV, v.normal));
        float2 offset = TransformViewToProjection(norm.xy);

        o.pos.xy += offset * o.pos.z * _Outline;
        o.color = _OutlineColor;
        return o;
    }

    fixed4 frag (v2f i) : SV_Target {
```

- Resolution
    - I wanted to maintain the same pixelated look of the original game while using 3D so I added a script that makes the camera render at a lower resolution. This script is a modification of the pixel boy script.

      Without

      With

Vertex Colouring Diagram