Computer Graphics 1

Tutorial

Assignment 3

Summer Semester 2024

Ludwig-Maximilians-Universität München

Contact

If you have any questions:

cg1ss24@medien.ifi.lmu.de

Organization

- **Theoretical part:**
  - Prepares you for the exam
  - Solve the tasks at home
  - We present the solutions during the tutorial sessions
- **Practical part:**
  - We will indicate which parts you need to do at home
  - Else this will be done & explained during the tutorial session
  - Ask questions!

Task 1): Bresenham’s Line Algorithm

- can be used for drawing lines on a raster display (the screen of the game mode in Unity) with discrete pixels
- determines the most optimal pixels to activate in order to approximate a straight line between two given points
- is especially useful because it uses only integer calculations, making it faster than other methods which use real numbers and floating-point arithmetic

Task 1): Bresenham’s Line Algorithm

- **Input:**

  The algorithm takes two points, the start point (x0, y0) and the end point (x1, y1)

- **Initialization:**

  Calculate the difference in x and y coordinates of the two points, for example

  dx = x1 - x0 and dy = y1 - y0

  Then initialize two error variables: D = 2*dy - dx and Dy = 2*dy

Task 1): Bresenham's Line Algorithm

- **Iteration:**

  Start from the start point (x0, y0) and move towards the end point (x1, y1) one pixel at a time along the x-axis. At each x-coordinate, we decide to move either straight ahead or diagonally, depending on the current error D.
- If D < 0, we move to the pixel directly to the right (increase x by 1, y stays the same)
- D = D + Dy
- If D >= 0, we move diagonally to the right (increase both x and y by 1)
- D = D + Dy - 2*dx

Task 1): Bresenham's Line Algorithm

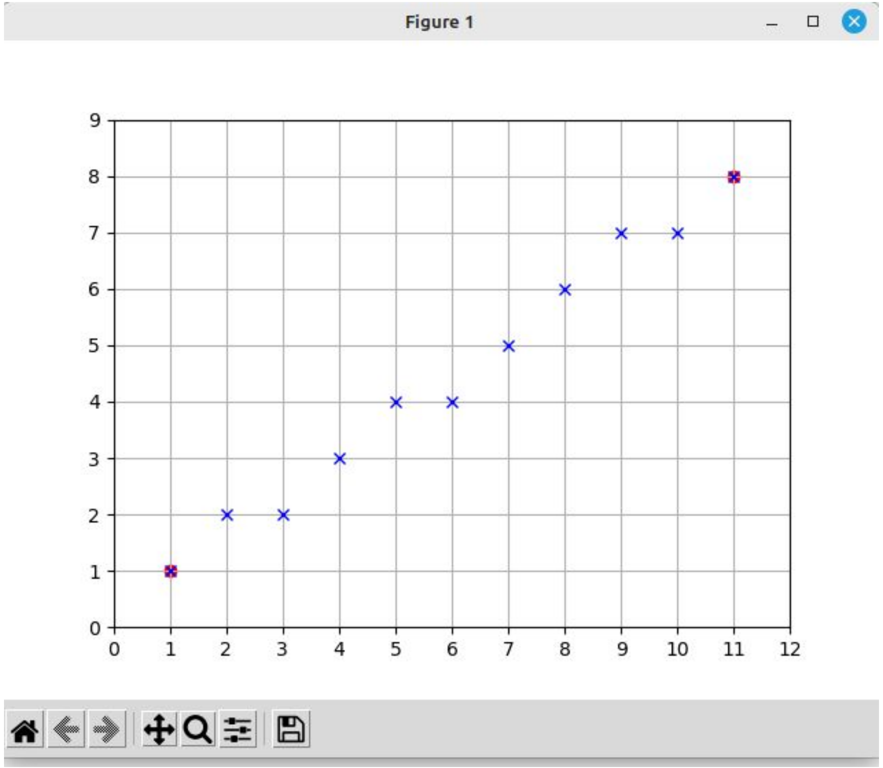
- **Termination:** The algorithm continues until it reaches the end point (x1, y1)
- **Limitation:** This explanation is for a line between 0 degrees to 45 degrees.

  The algorithm can be adjusted for different lines by switching x and y.

  We assume that x1 > x0 and y1 > y0

Task 1): Bresenham's Line Algorithm

Line plot figure with matplotlib in python:



Task 2: Cameras in CG - Background

- **View Matrix:**

  Abstract in nature, not a direct representation of the real world 3D space
- The world transformation matrix determines the position and orientation of an object in 3D space
- view matrix is used to transform vertices of a 3D model from world-space to view-space
- Always differentiate these two!

Task 2: Cameras in CG - Background

- Imagine you are holding a video camera, taking a picture of a house
- You can get a different view of the house by moving your camera around it

  => the scene appears to be moving when you view the image through your camera’s view finder
- In computer graphics the camera does not move at all !
- the world is moving in the opposite direction and orientation of how the camera moves in reality

Task 2: Why is View and Projection Transformation needed?

The Camera Transformation Matrix:

- transformation that places the camera in the correct **position and orientation** in world space
- has to be applied to 3D model of the camera if it is represented in the scene

The View Matrix:

- transform vertices from world-space to view-space
- inverse of the cameras transformation matrix

Task 2 a): Camera Translation

The translation is based on the position of the camera

$$T_{view} = \begin{bmatrix} 1 & 0 & 0 & -p1 \\ 0 & 1 & 0 & -p2 \\ 0 & 0 & 1 & -p3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Task 2 b): Camera Rotation

We can rotate the camera coordinate frame from l to -Z, u to +Y and (l x u) to +X

$$R^{-1}_{view} = \begin{bmatrix} x_{l \times u} & x_u & x_{-l} & 0 \\ y_{l \times u} & y_u & y_{-l} & 0 \\ z_{l \times u} & z_u & z_{-l} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Task 2 b): Camera Rotation

With this little trick we get the inverse matrix: $R^{-1} = R^T$

We rotate +X to (l x u), +Y to u, and +Z to-l

$$R_{view} = \begin{bmatrix} x_{l \times u} & y_{l \times u} & z_{l \times u} & 0 \\ x_u & y_u & z_u & 0 \\ x_{-l} & y_{-l} & z_{-l} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Task 2 a): Orthographic Projection

Projecting a point to the x-y plane is irrelevant to z, therefore the projected coordinates are (x, y, 0)

Task 2 b): Orthographic Projection

We translate the center of the cube to the origin, then scale its length, width and height

We rotate +X to (l x u), +Y to u, and +Z to-l

$$T_{ortho} = \begin{bmatrix} 2/(r-l) & 0 & 0 & 0 \\ 0 & 2/(t-b) & 0 & 0 \\ 0 & 0 & 2/(n-f) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -(r+l)/2 \\ 0 & 1 & 0 & -(t+b)/2 \\ 0 & 0 & 1 & -(n+f)/2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Task 2 b): Orthographic Projection

We translate the center of the cube to the origin, then scale its length, width and height

We rotate +X to (l x u), +Y to u, and +Z to-l

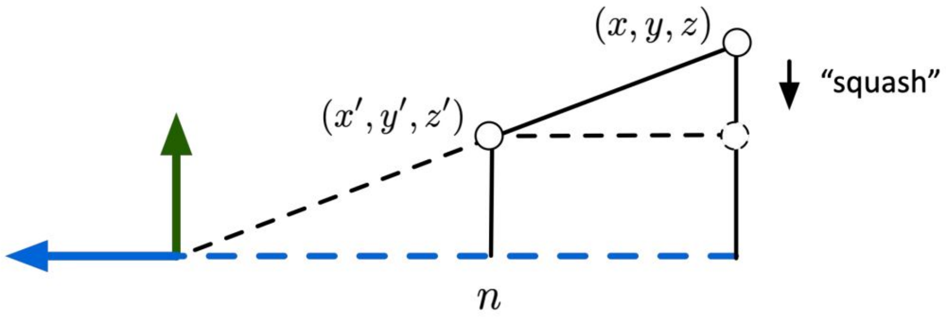
$$= \begin{bmatrix} 2/(r-l) & 0 & 0 & (l+r)/(l-r) \\ 0 & 2/(t-b) & 0 & (b+t)/(b-t) \\ 0 & 0 & 2/(n-f) & (f+n)/(f-n) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Task 2 c): Perspective Projection

If we consider (x, y, z) that projects to (x’, y’, ?), we get similar triangles:

y’/y = n/z

x’/x = n/z



Following:

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \longrightarrow \begin{bmatrix} nx/z \\ ny/z \\ ? \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ ? \\ z \end{bmatrix}$$

Task 2 c): Perspective Projection

We can observe:

$$\begin{pmatrix} nx \\ ny \\ ? \\ z \end{pmatrix} = T_{\text{persp}\to\text{ortho}} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ ? & ? & ? & ? \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Any point on the near plane will not change, when z = n,

therefore (x, y, n, 1) will not move and just transform to itself:

$$\begin{pmatrix} nx \\ ny \\ ? \\ n \end{pmatrix} \leftarrow \begin{pmatrix} x \\ y \\ n \\ 1 \end{pmatrix} = \begin{pmatrix} nx \\ ny \\ n^2 \\ n \end{pmatrix}$$

Task 2 c): Perspective Projection

Because ? is irrelevant to x and y, the transformation matrix for the near plane should be:

$$\begin{pmatrix} nx \\ ny \\ n^2 \\ n \end{pmatrix} = T_{\text{persp}\rightarrow\text{ortho}} \begin{pmatrix} x \\ y \\ n \\ 1 \end{pmatrix} = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & w_1 & w_2 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ n \\ 1 \end{pmatrix}$$

Task 2 c): Perspective Projection

It is important to understand that the center of the far plane will not change.

Following, when x = 0, y = 0 and z = f:

$$\begin{pmatrix} n \cdot 0 \\ n \cdot 0 \\ ? \\ f \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ f \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ f^2 \\ f \end{pmatrix}$$

We need to apply this for further computations:

$$\begin{pmatrix} 0 \\ 0 \\ f^2 \\ f \end{pmatrix} = T_{\text{persp}\to\text{ortho}} \begin{pmatrix} 0 \\ 0 \\ f \\ 1 \end{pmatrix} = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & w_1 & w_2 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ f \\ 1 \end{pmatrix}$$

Task 2 c): Perspective Projection

Near Plane:

$$\begin{pmatrix} nx \\ ny \\ n^2 \\ z \end{pmatrix} = T_{\text{persp}\to\text{ortho}} \begin{pmatrix} x \\ y \\ n \\ 1 \end{pmatrix} = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & w_1 & w_2 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ n \\ 1 \end{pmatrix} \implies nw_1 + w_2 = n^2$$

Far Plane:

$$\begin{pmatrix} 0 \\ 0 \\ f^2 \\ f \end{pmatrix} = T_{\text{persp}\to\text{ortho}} \begin{pmatrix} 0 \\ 0 \\ f \\ 1 \end{pmatrix} = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & w_1 & w_2 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ f \\ 1 \end{pmatrix} \implies fw_1 + w_2 = f^2$$

Task 2 c): Perspective Projection

$$\Longrightarrow \; w_1 = n + f, w_2 = -nf$$

$$\Longrightarrow \; T_{\text{persp}\to\text{ortho}} = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Task 2 d): Combined Projection Matrix

We can use our results from the previous tasks and combine the respective matrices:

$$T_{ortho} = \begin{bmatrix} 2/(r-l) & 0 & 0 & (l+r)/(l-r) \\ 0 & 2/(t-b) & 0 & (b+t)/(b-t) \\ 0 & 0 & 2/(n-f) & (f+n)/(f-n) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{\text{persp}\to\text{ortho}} = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

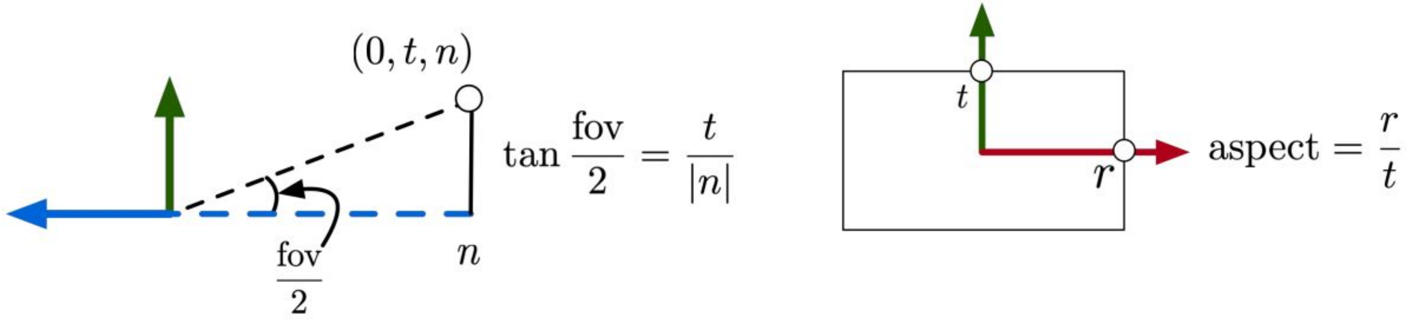
Task 2 d): Combined Projection Matrix

Combining the matrices results in:

$$\Longrightarrow T_{\text{ortho}} T_{\text{persp}\rightarrow\text{ortho}} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{l+r}{l-r} & 0 \\ 0 & \frac{2n}{t-b} & \frac{b+t}{b-t} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{f-n} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Task 2 d): Combined Projection Matrix

To figure out (l, r, b, t), we can use basic geometry:



We get:

l = -r = -λt = -λ(-n)tan θ/2 = λn tan θ/2

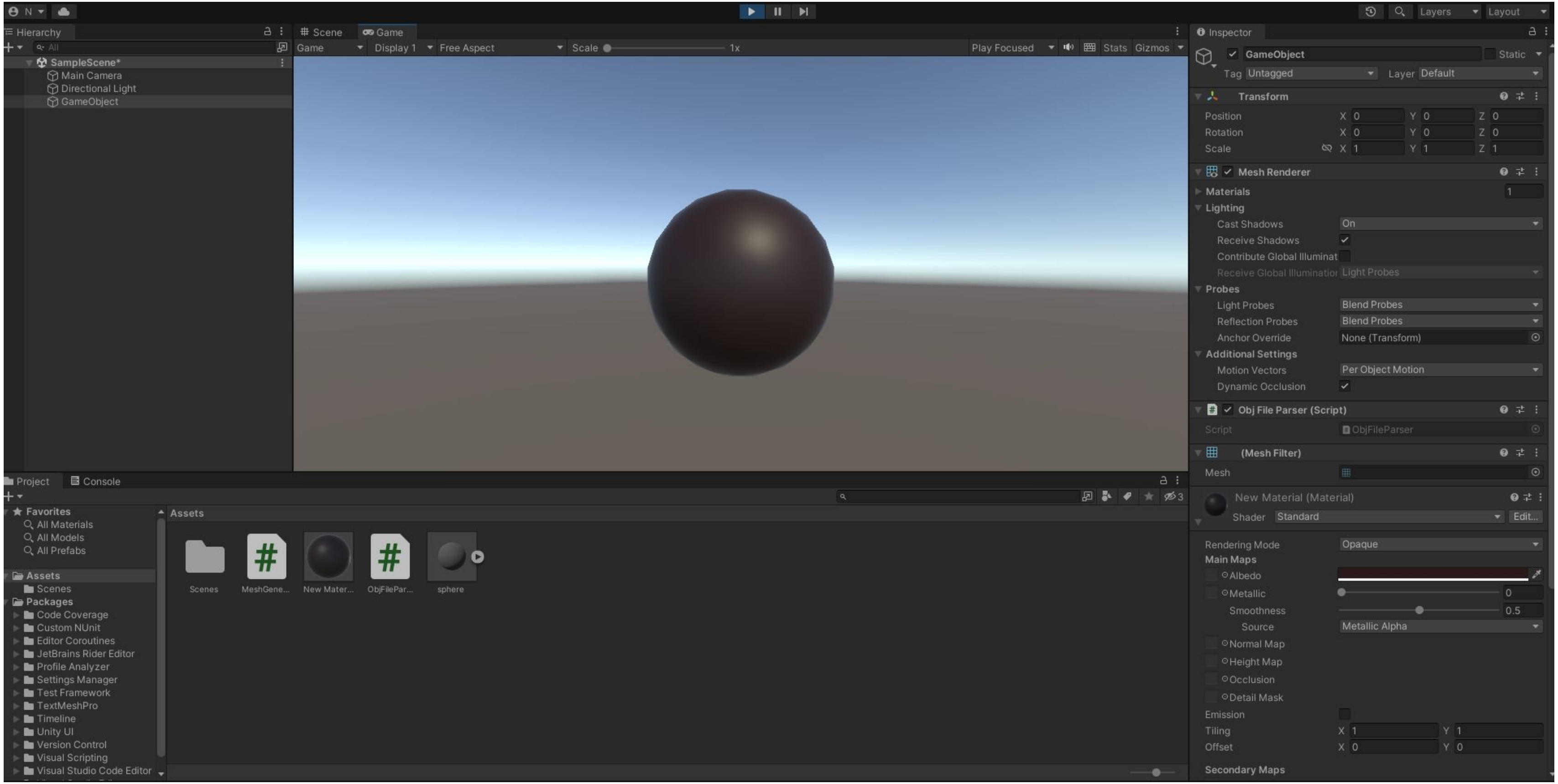
b = -t = -(-n)tan θ/2 = n tan θ/2

Task 2 d): Combined Projection Matrix

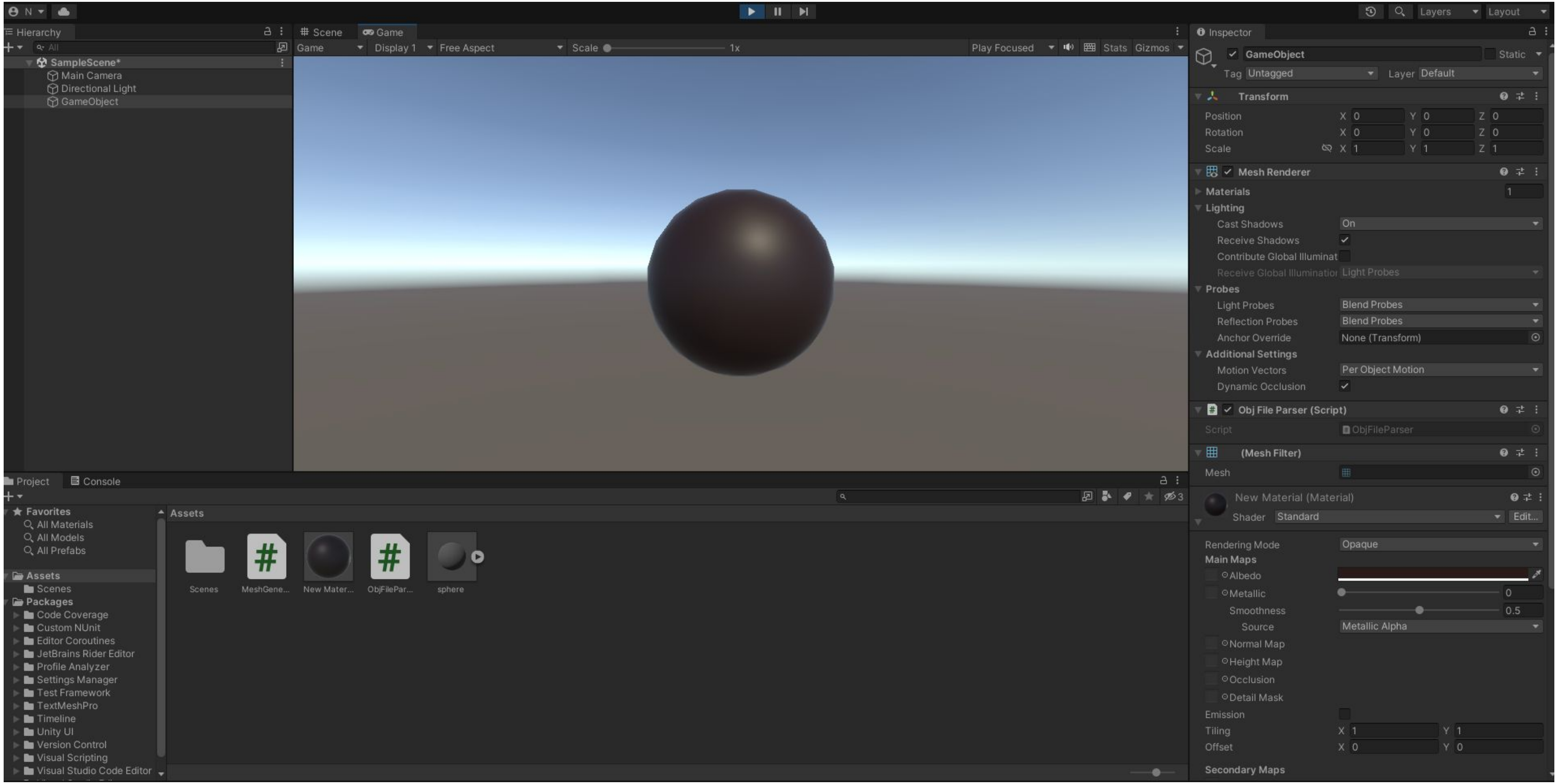
The final matrix:

$$T_{\text{persp}} = T_{\text{ortho}} T_{\text{persp}\to\text{ortho}} = \begin{pmatrix} -\frac{1}{\lambda \tan\frac{\theta}{2}} & 0 & 0 & 0 \\ 0 & -\frac{1}{\tan\frac{\theta}{2}} & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{f-n} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

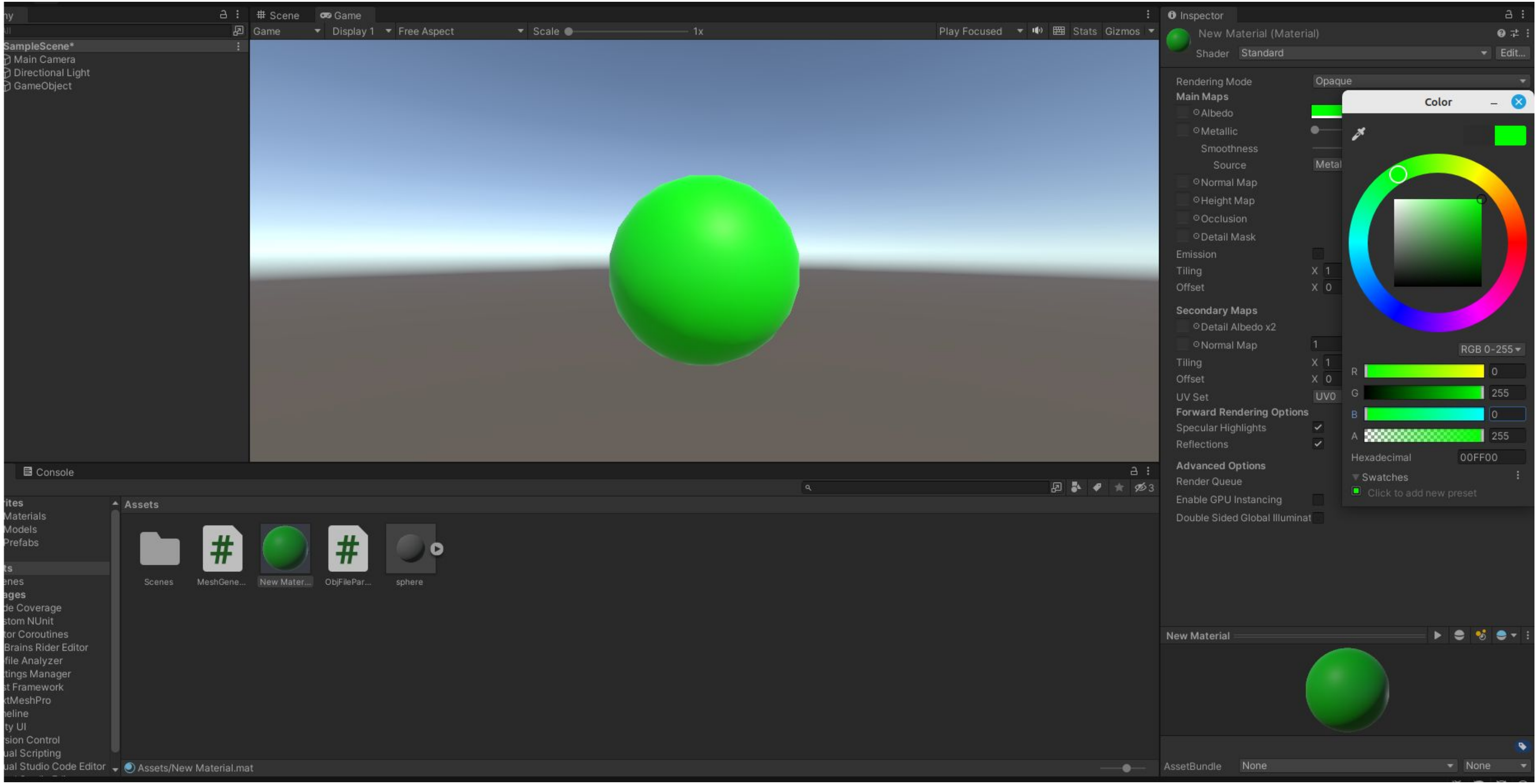
Task 2: Starting Point



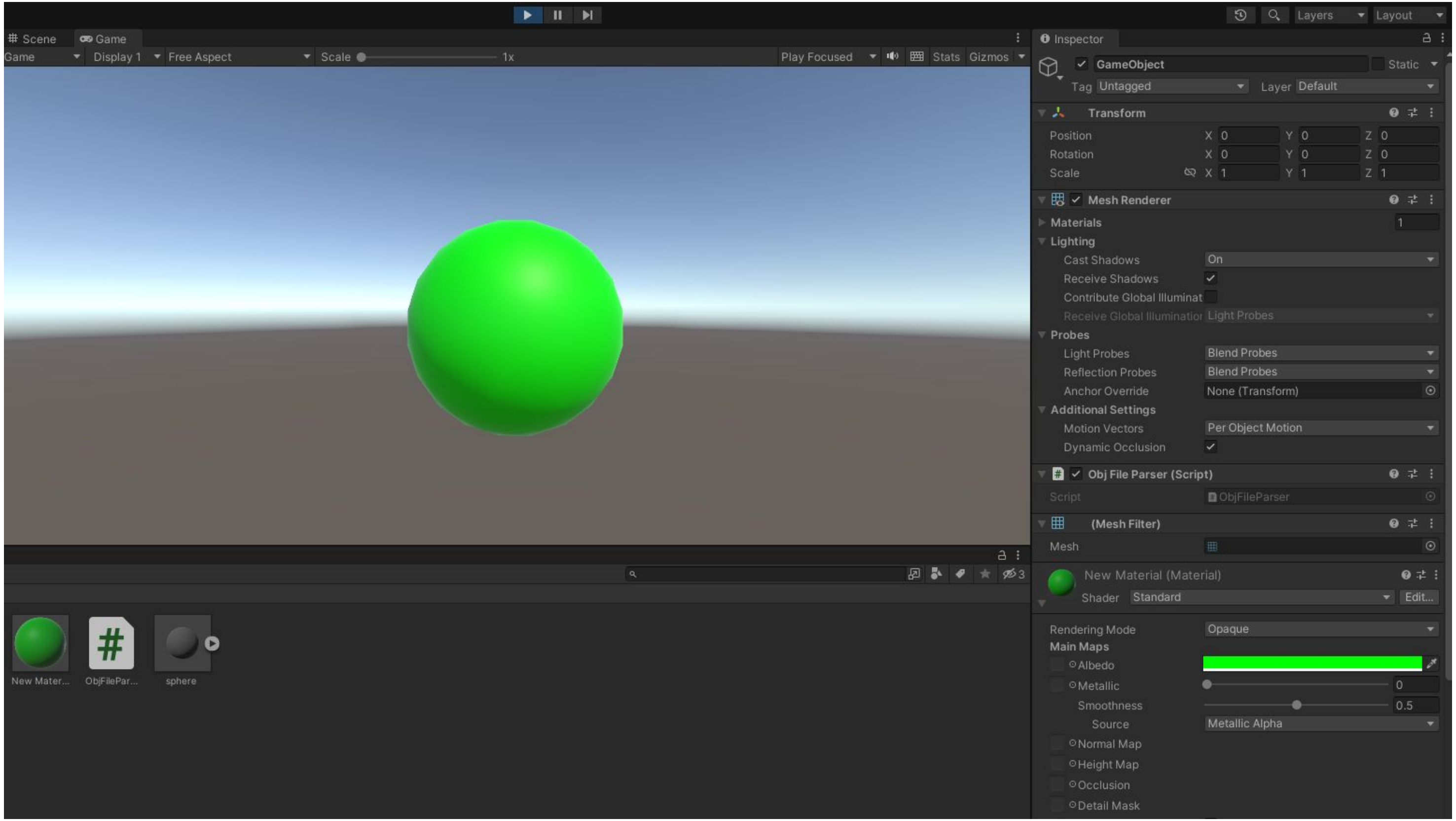
Work through Assignment 2 if you did not already



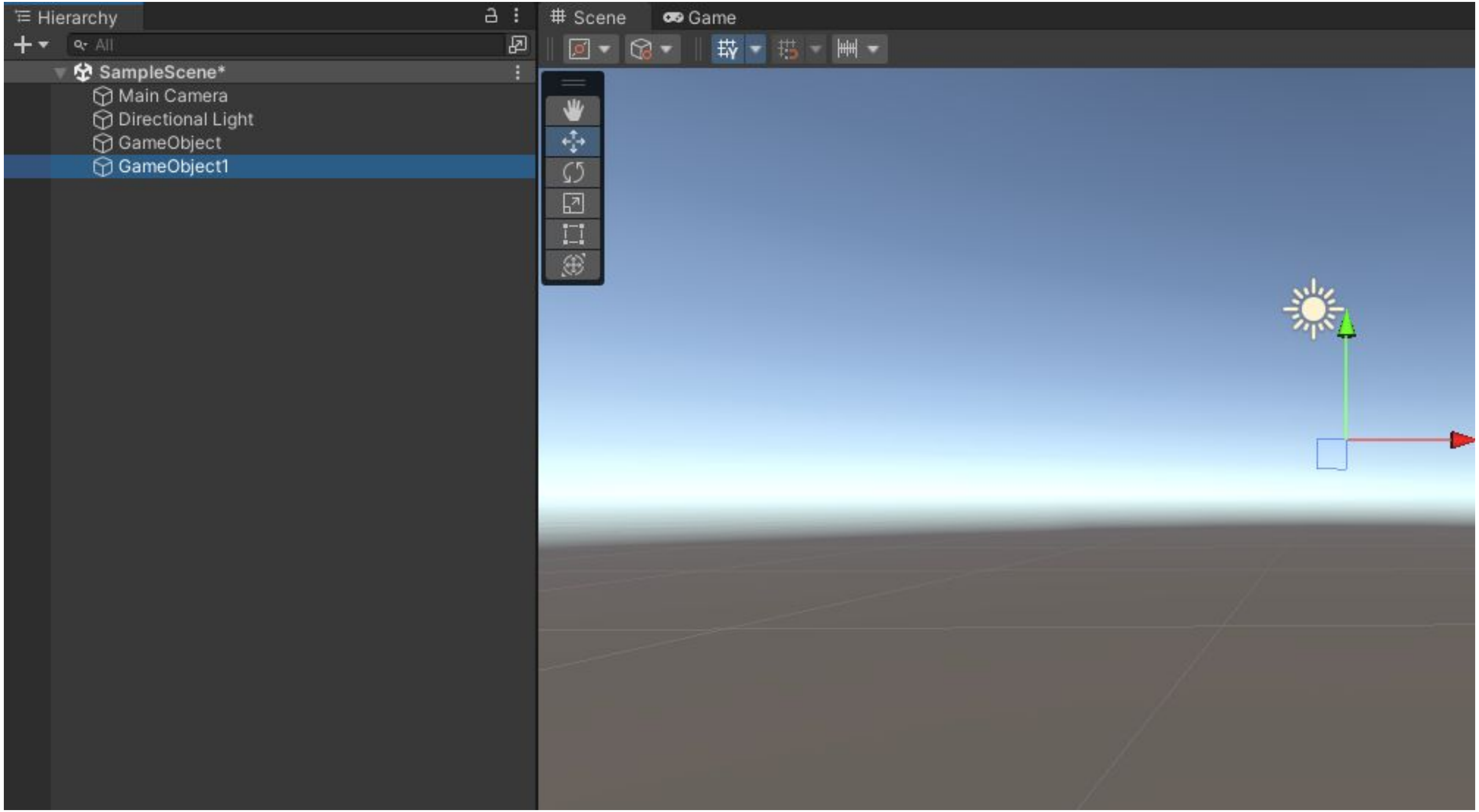
Let’s change the colour to green



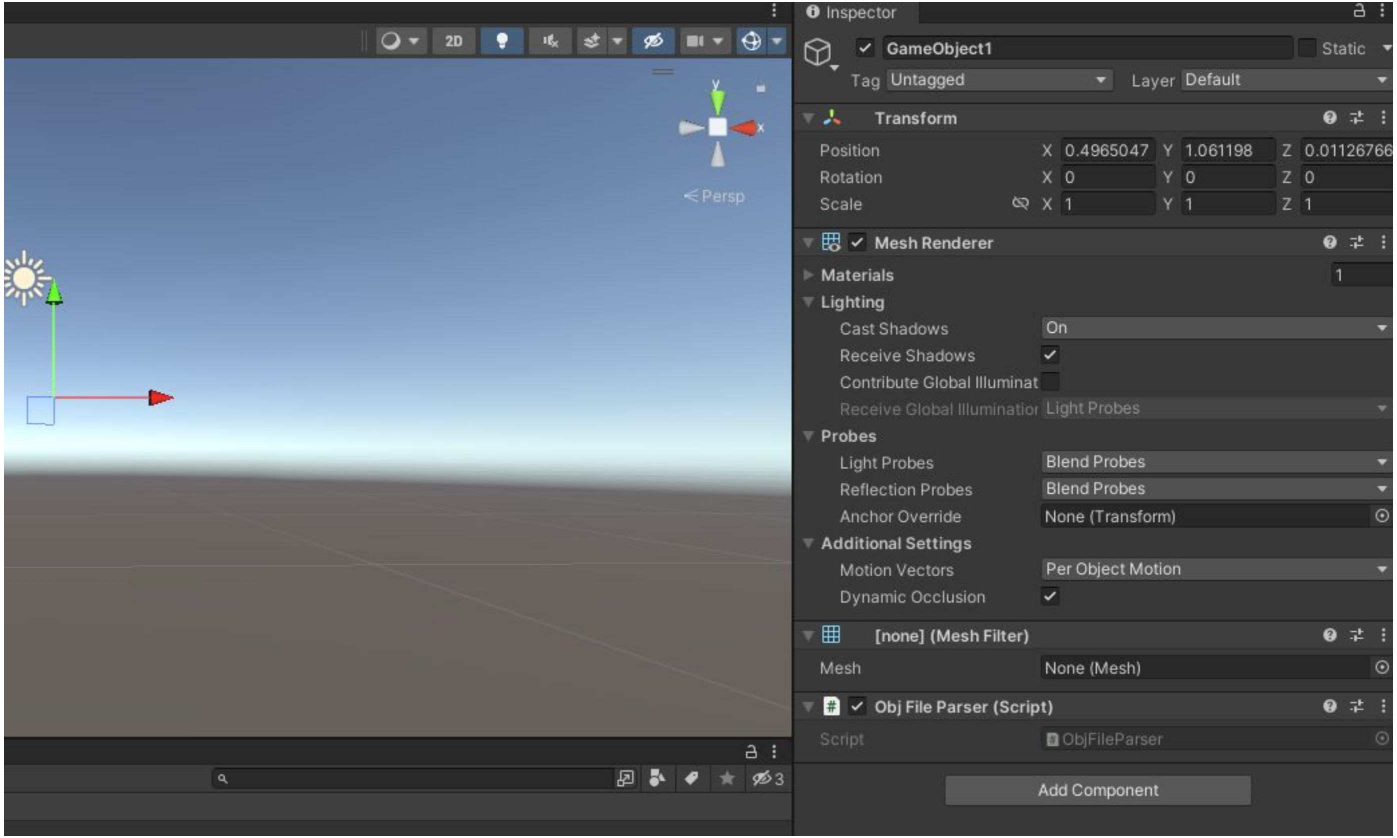
Inspector Tab of Game Object after colour change



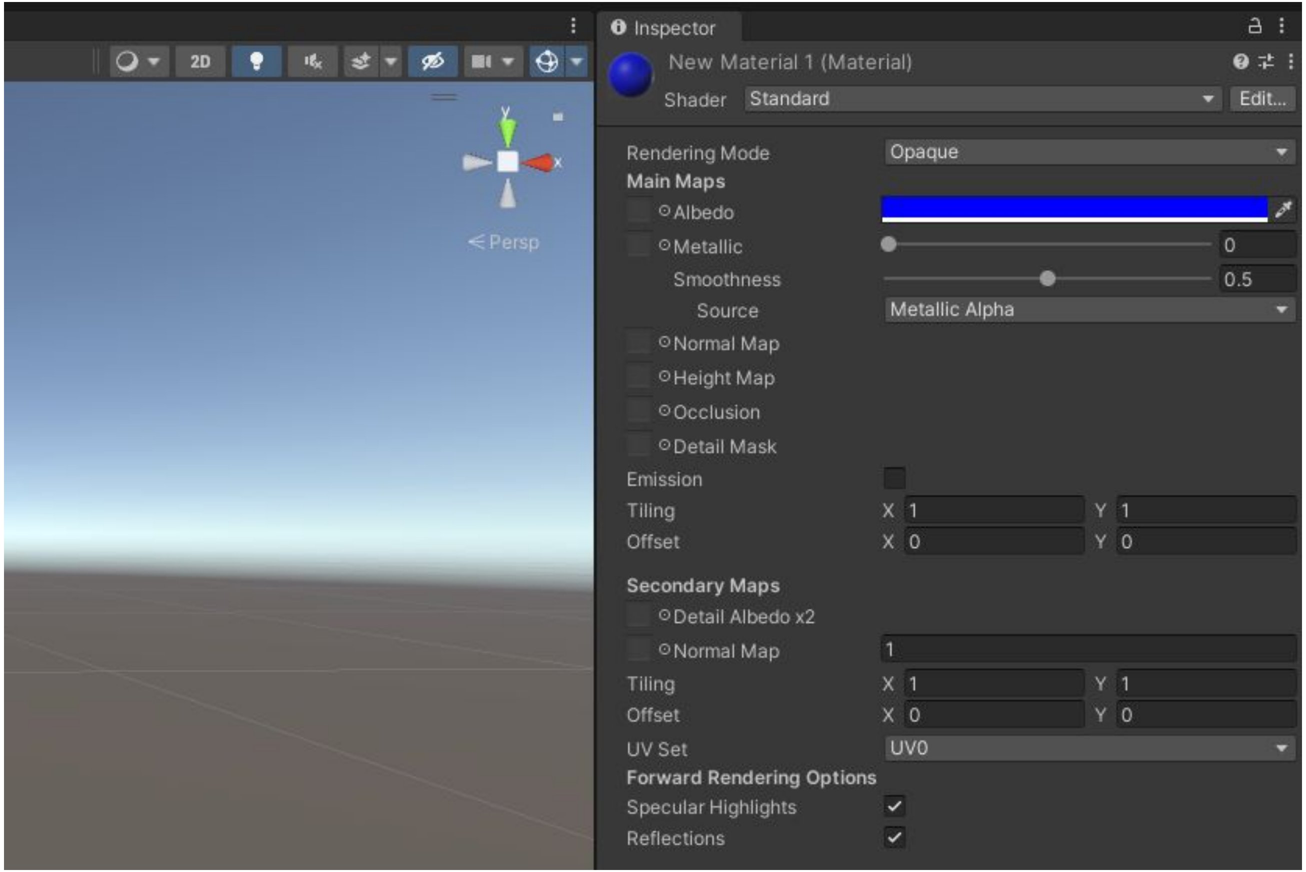
Create a second empty game object



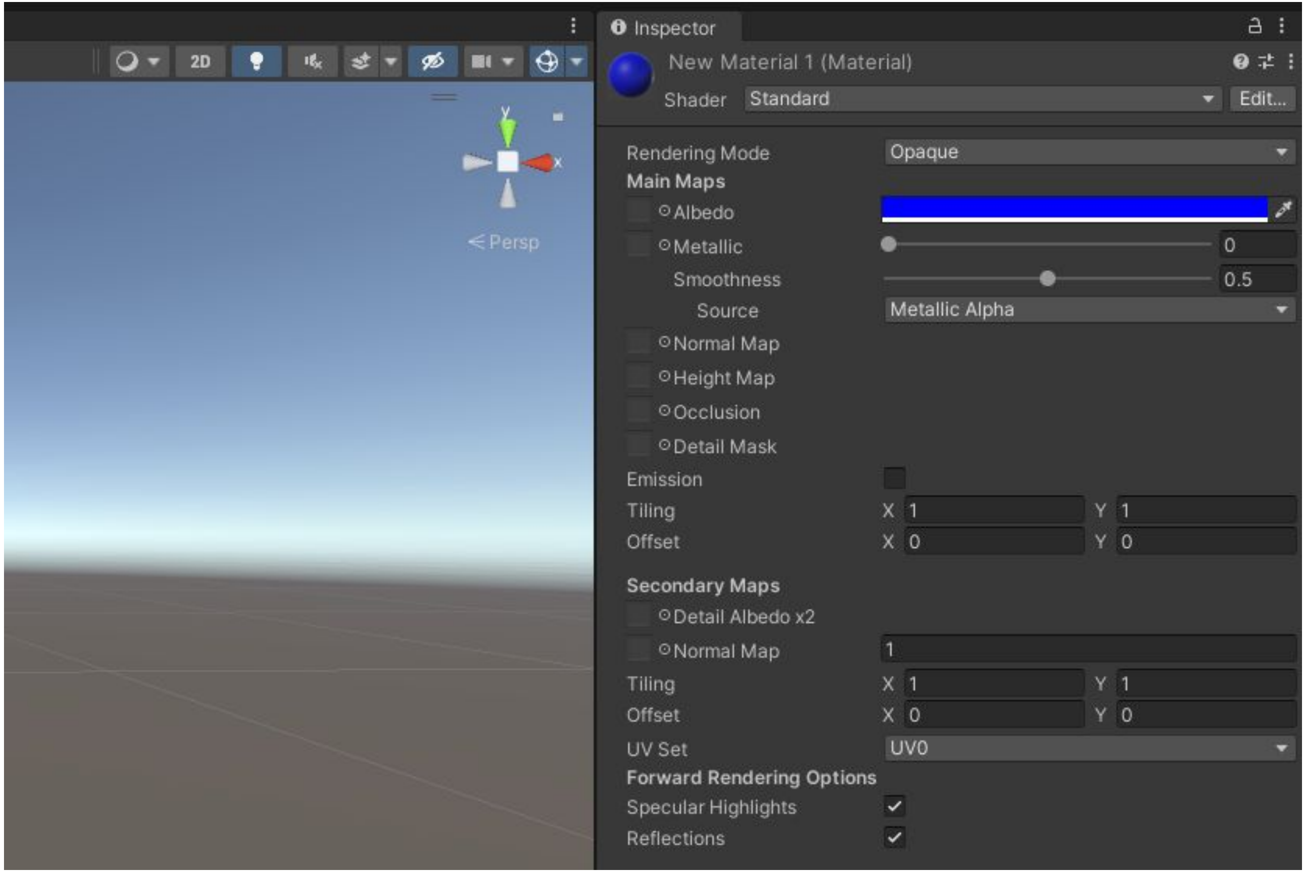
Add Mesh Renderer Component + ObjFileParser Script



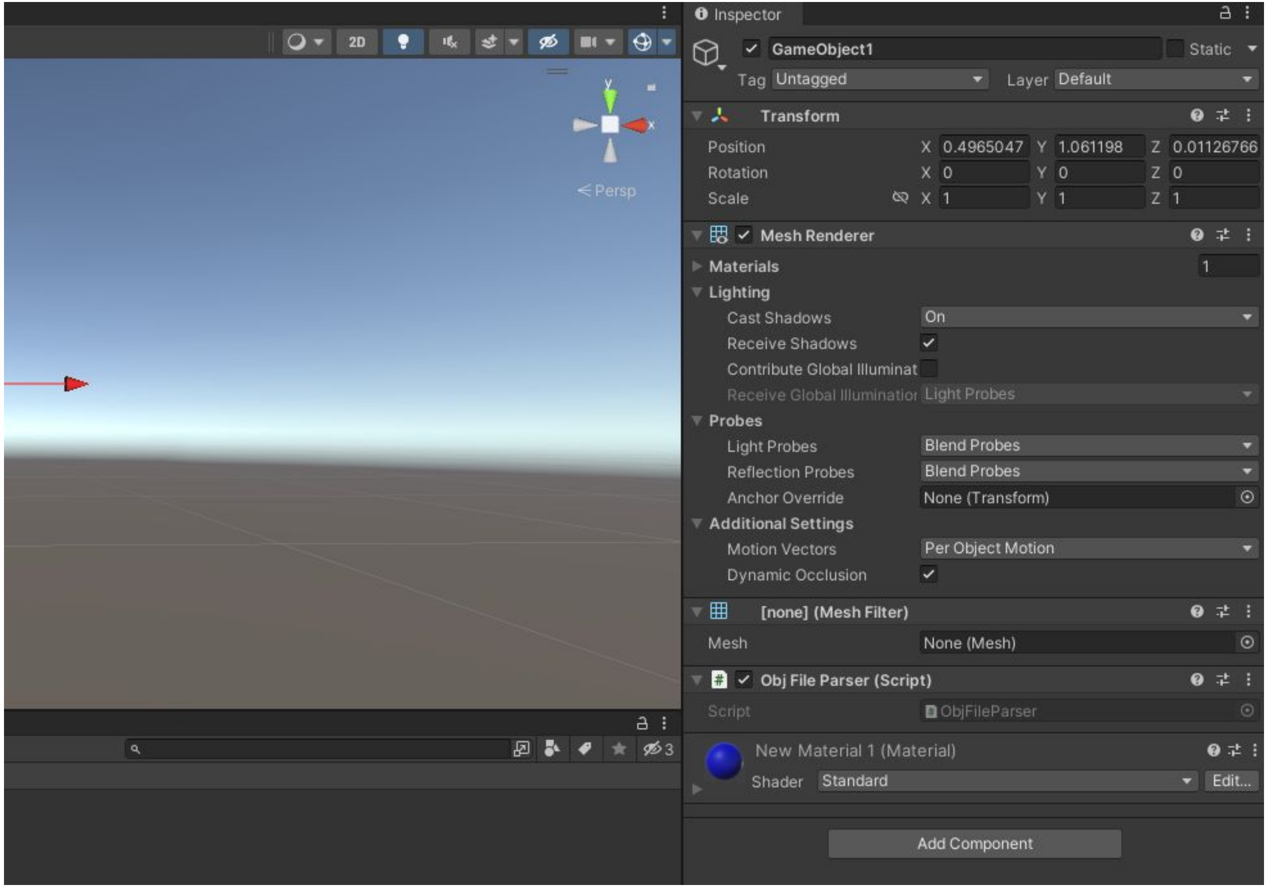
To differentiate our two objects, create a new material



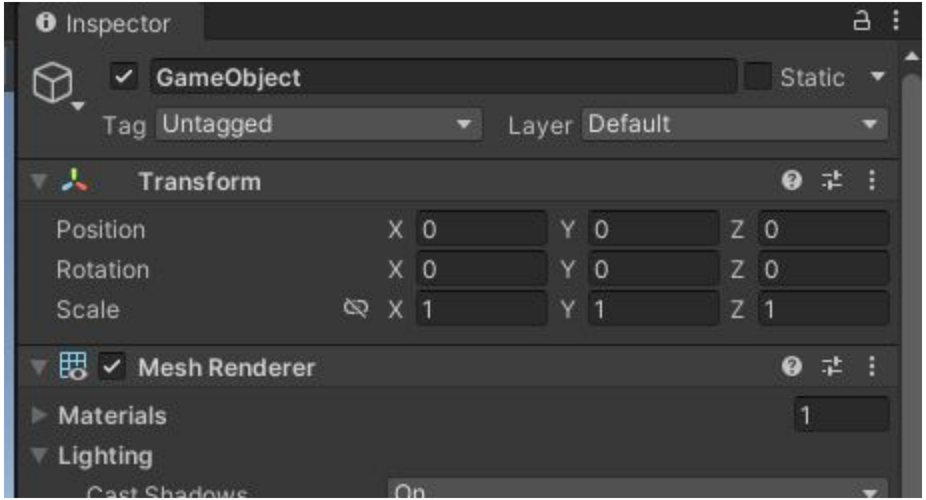
Choose any colours, as long as the two are not similar



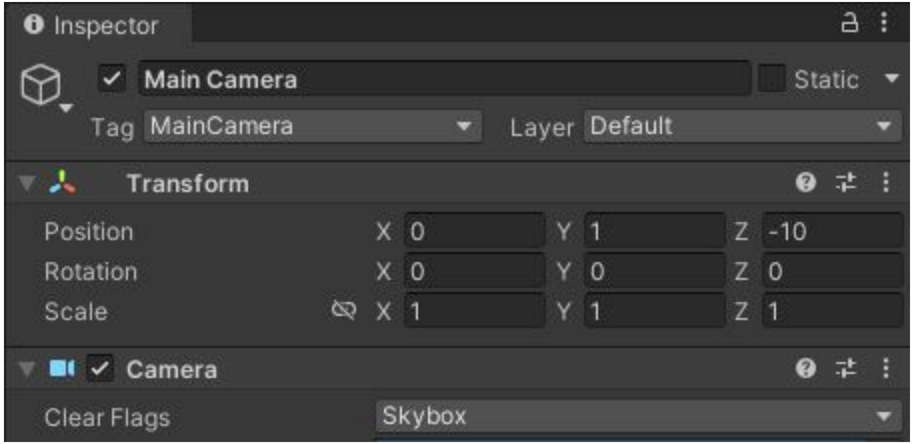
Add the new material to the second game object



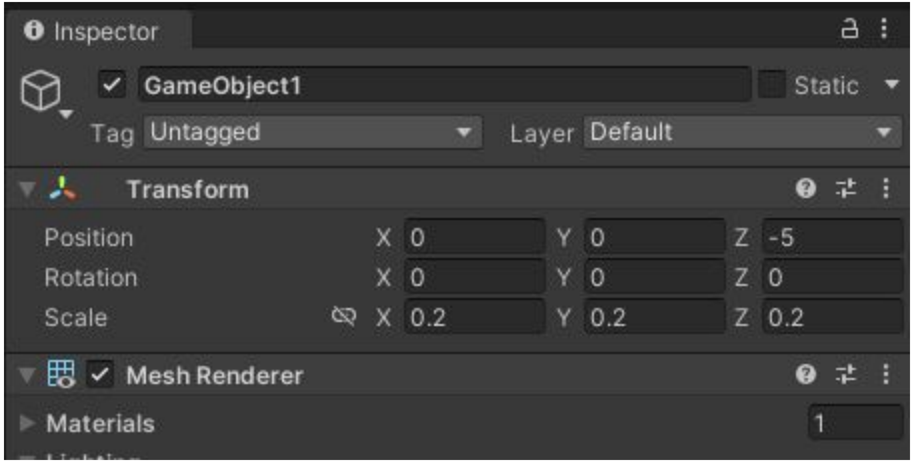
The first game object sits at (0, 0, 0)



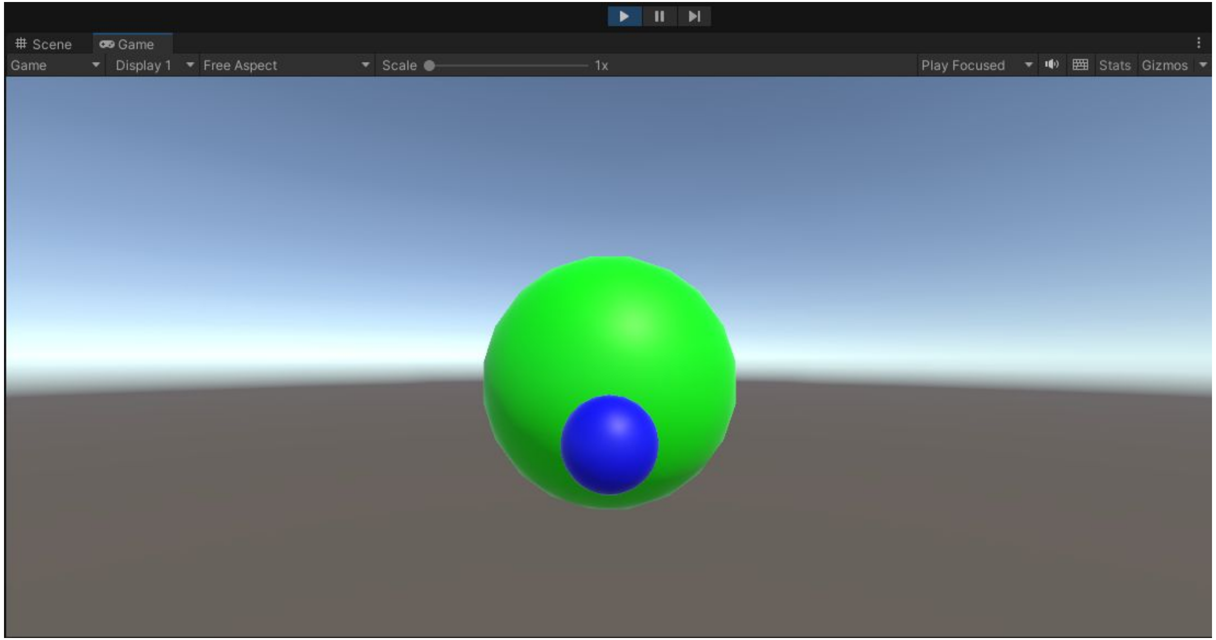
Make sure that the main camera is at (0, 0, -10)



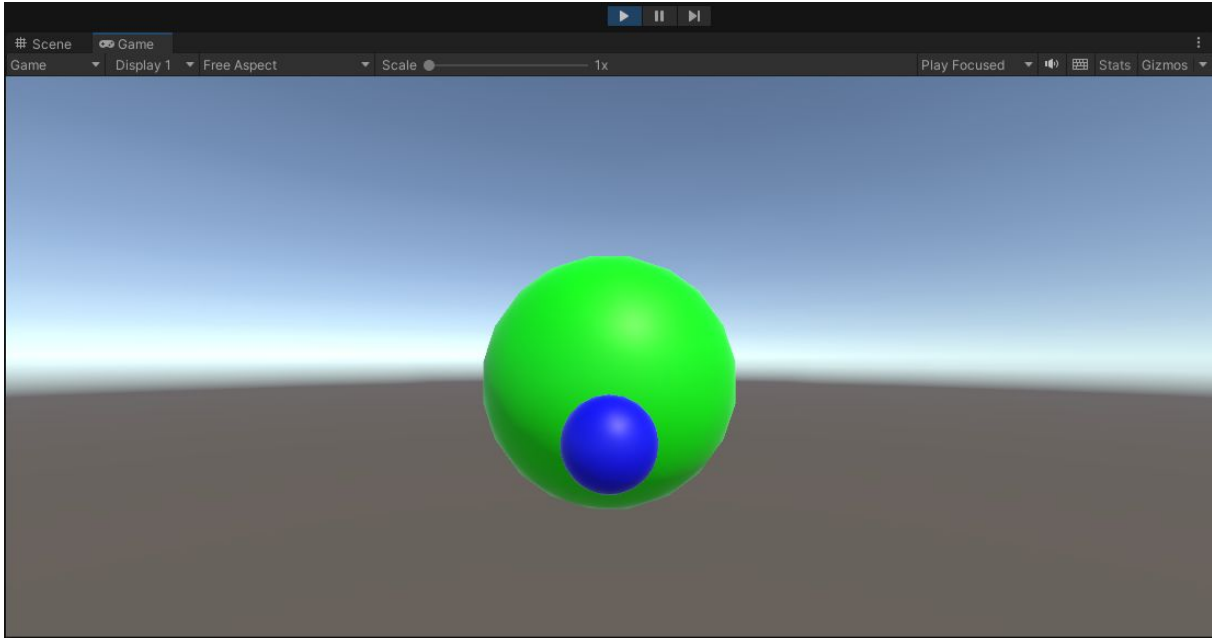
The second game object is scaled down and placed between the camera and the first game object



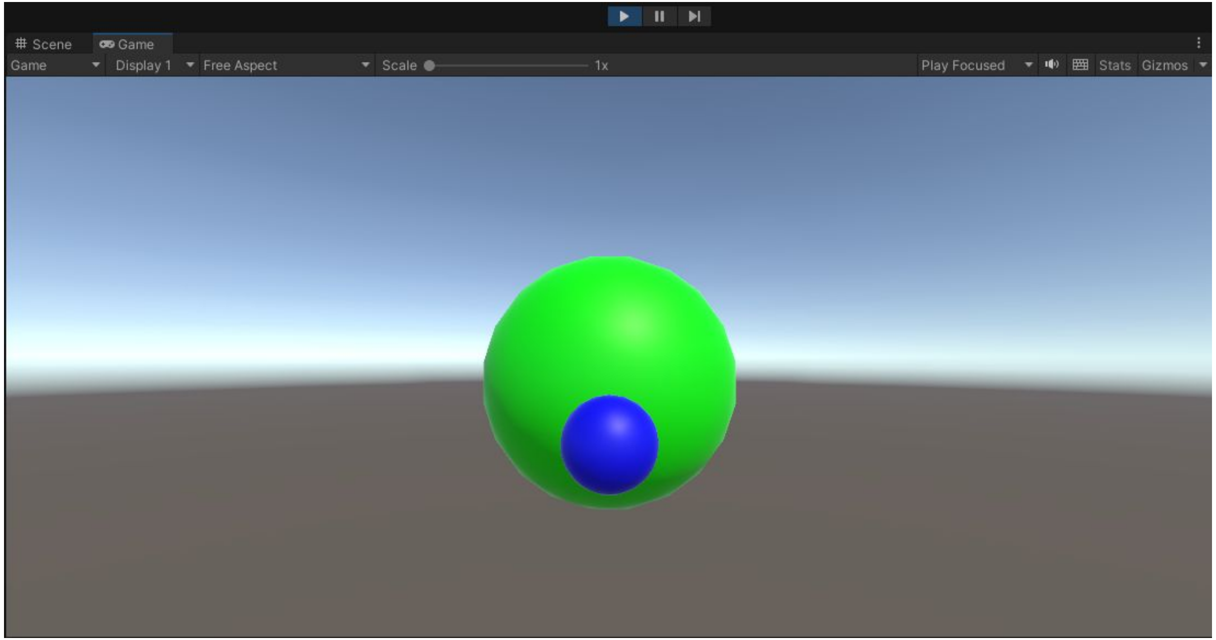
Switch to game mode and you see what you expect



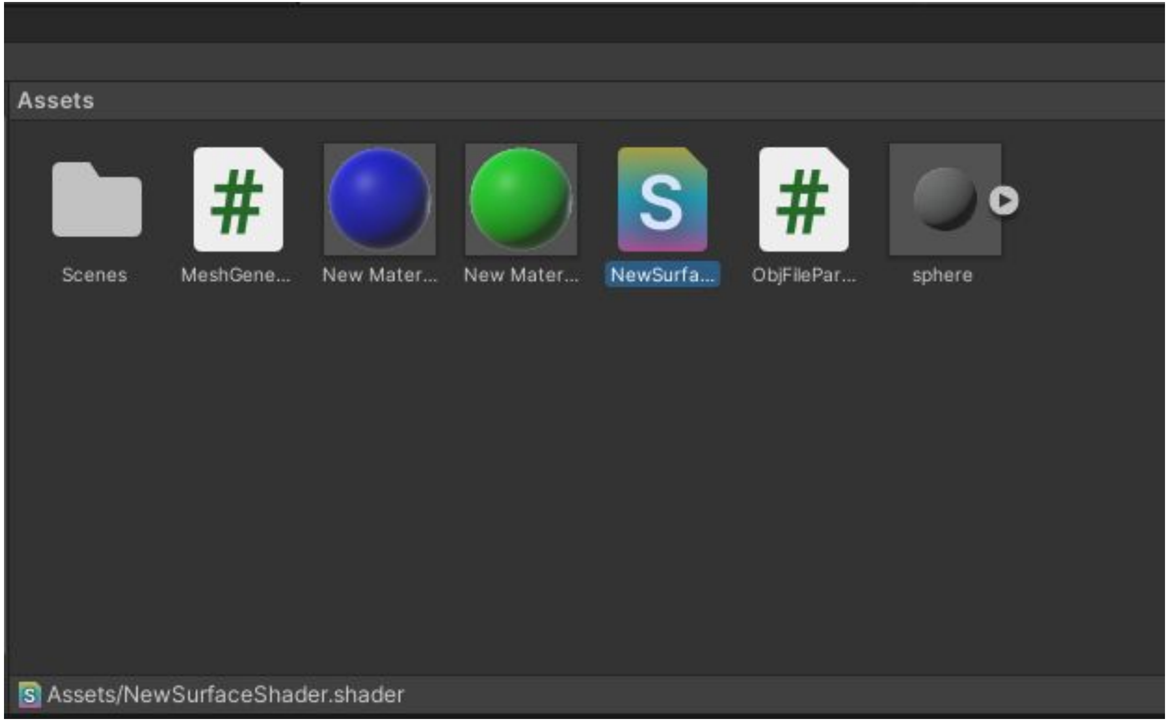
The built-in Render Pipeline takes care of Z-Buffering



Now let's see what happens if Z-Buffering is turned off



Assets -> Create -> Shader -> Standard Surface Shader



Switch to VSCode and open the .shader file

MeshGeneration.cs | ObjFileParser.cs | NewSurfaceShader.shader

Assets > NewSurfaceShader.shader

```
Shader "Custom/NewSurfaceShader"
{
    Properties
    {
        _Color ("Color", Color) = (1,1,1,1)
        _MainTex ("Albedo (RGB)", 2D) = "white" {}
        _Glossiness ("Smoothness", Range(0,1)) = 0.5
        _Metallic ("Metallic", Range(0,1)) = 0.0
    }
    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 200

        CGPROGRAM
        // Physically based Standard lighting model, and enable shadows on all light type
        #pragma surface surf Standard fullforwardshadows
```

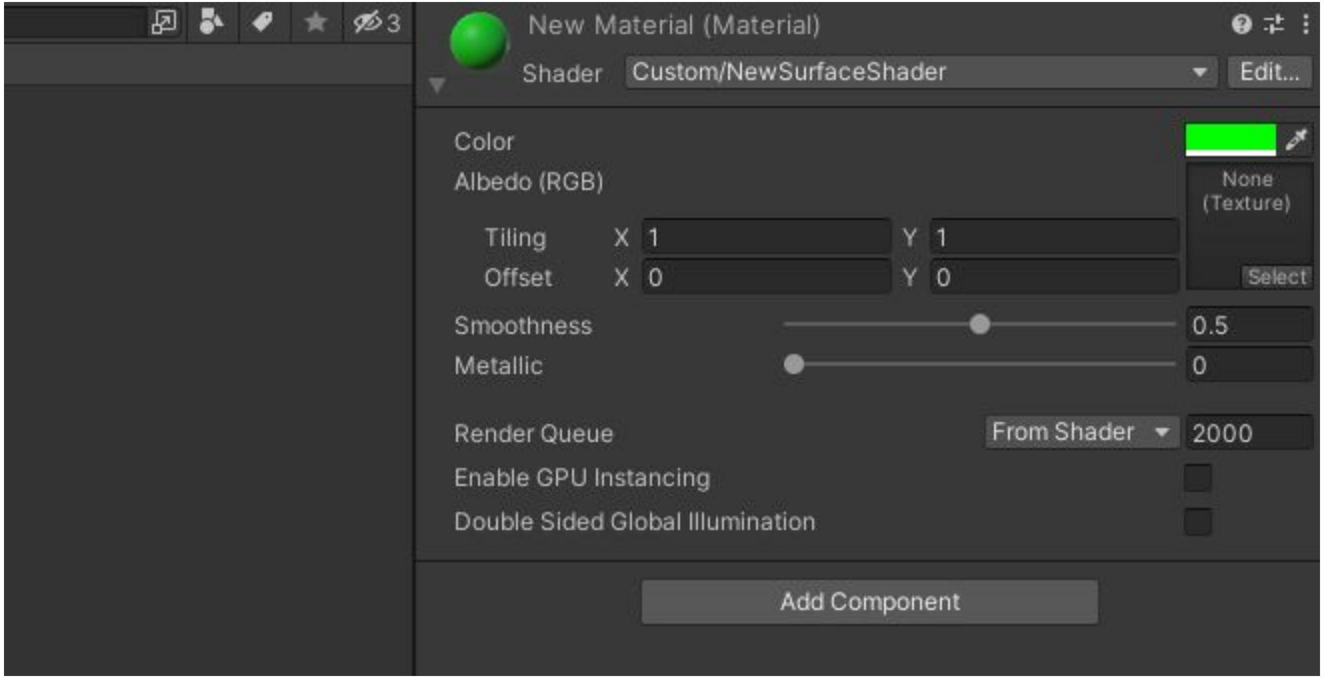
Add this line to SubShader

Assets > NewSurfaceShader.shader

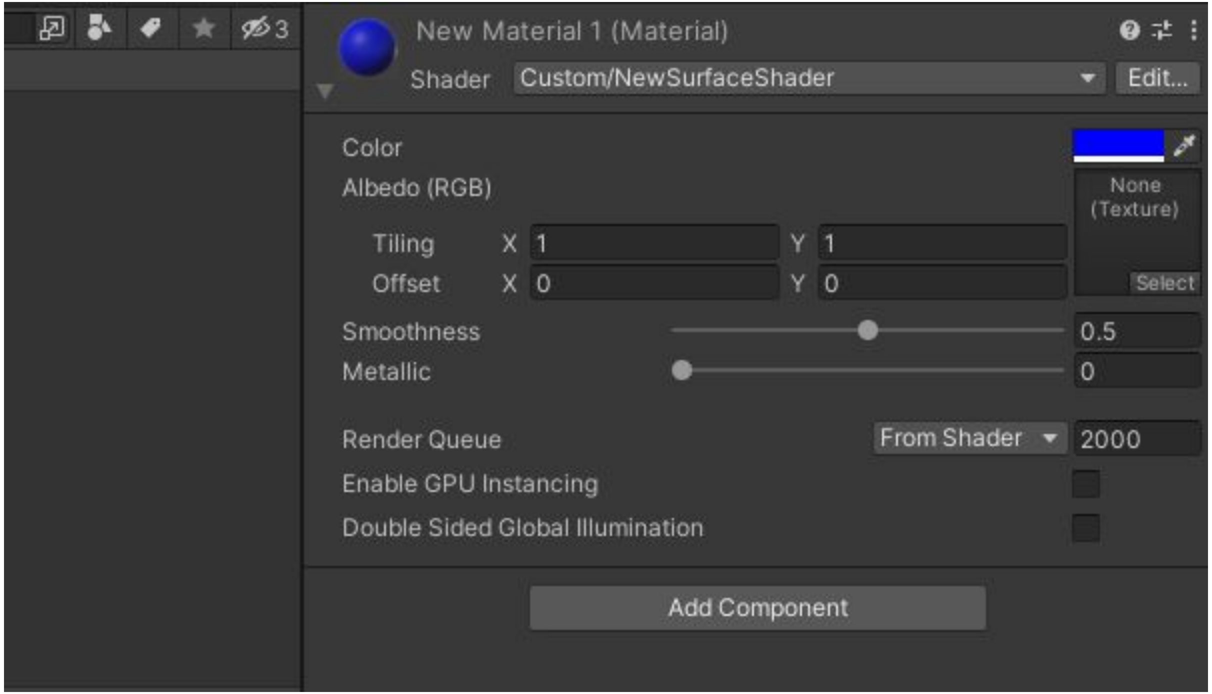
```
Shader "Custom/NewSurfaceShader"
{
    Properties
    {
        _Color ("Color", Color) = (1,1,1,1)
        _MainTex ("Albedo (RGB)", 2D) = "white" {}
        _Glossiness ("Smoothness", Range(0,1)) = 0.5
        _Metallic ("Metallic", Range(0,1)) = 0.0
    }
    SubShader
    {
        //Turns off Z-Buffering
        ZWrite Off

        Tags { "RenderType"="Opaque" }
        LOD 200
```

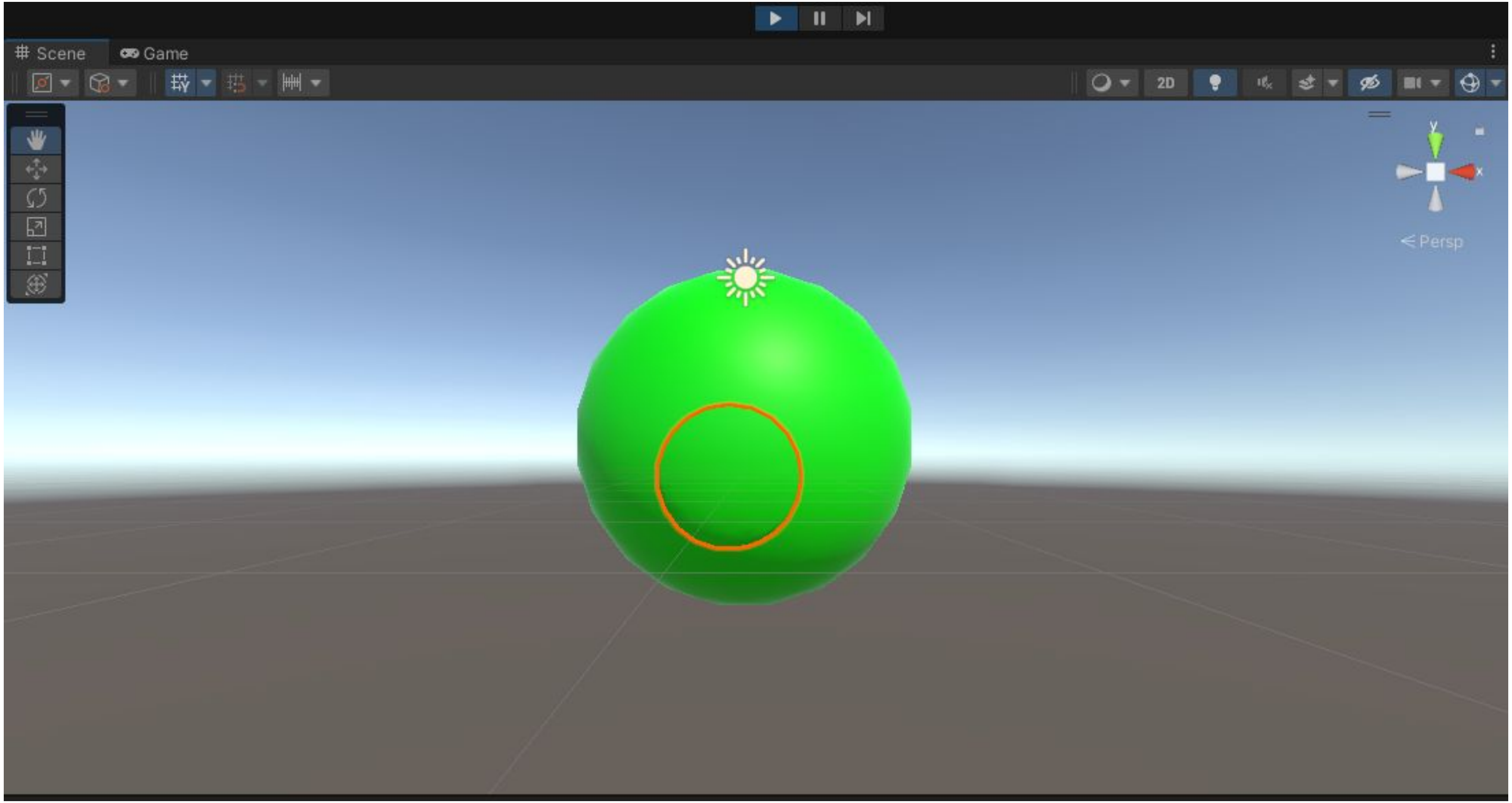
Open the game object in the Inspector and drag and drop the surface shader to the Shader field of the Material component



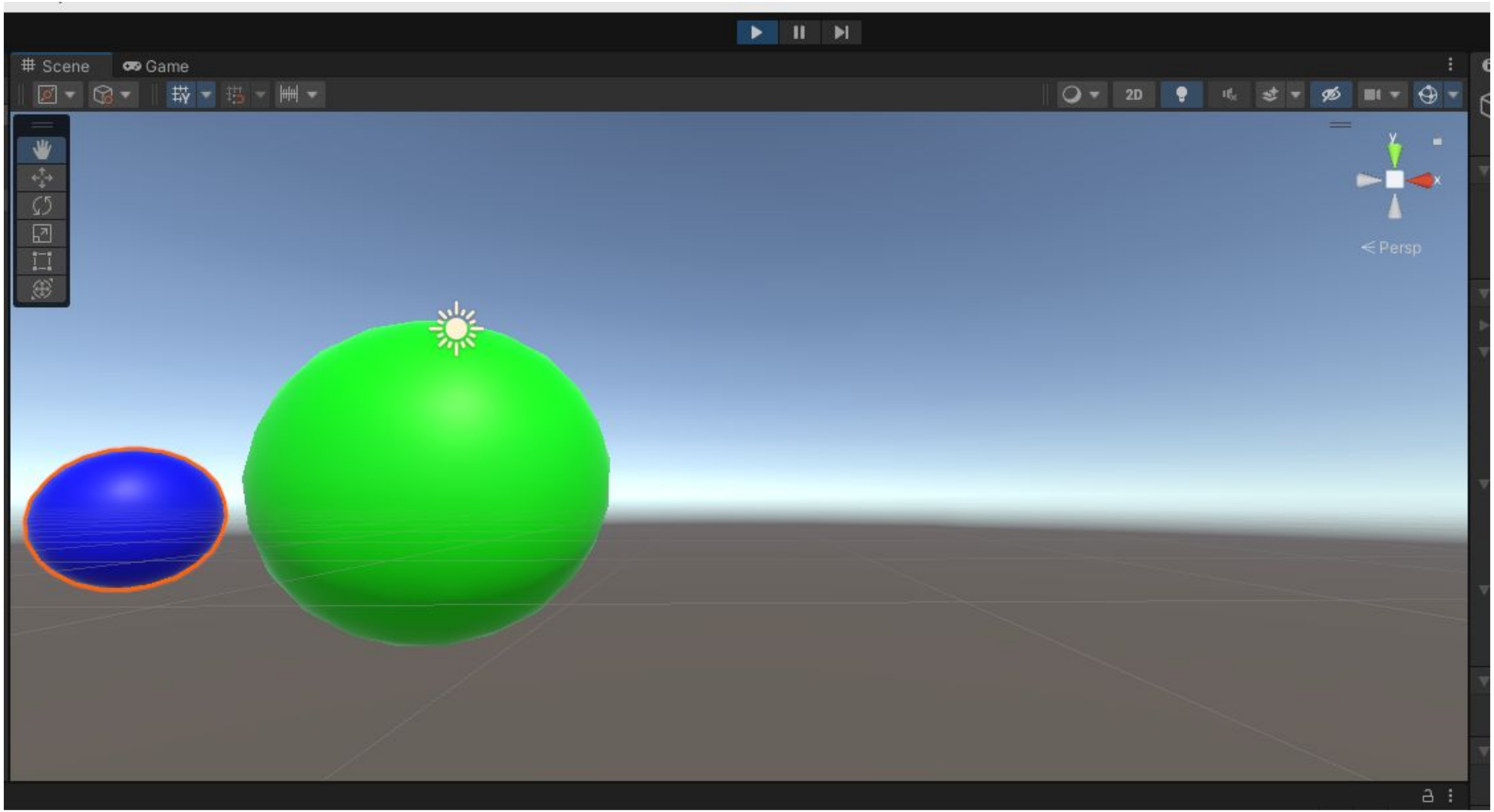
Do this for both game objects



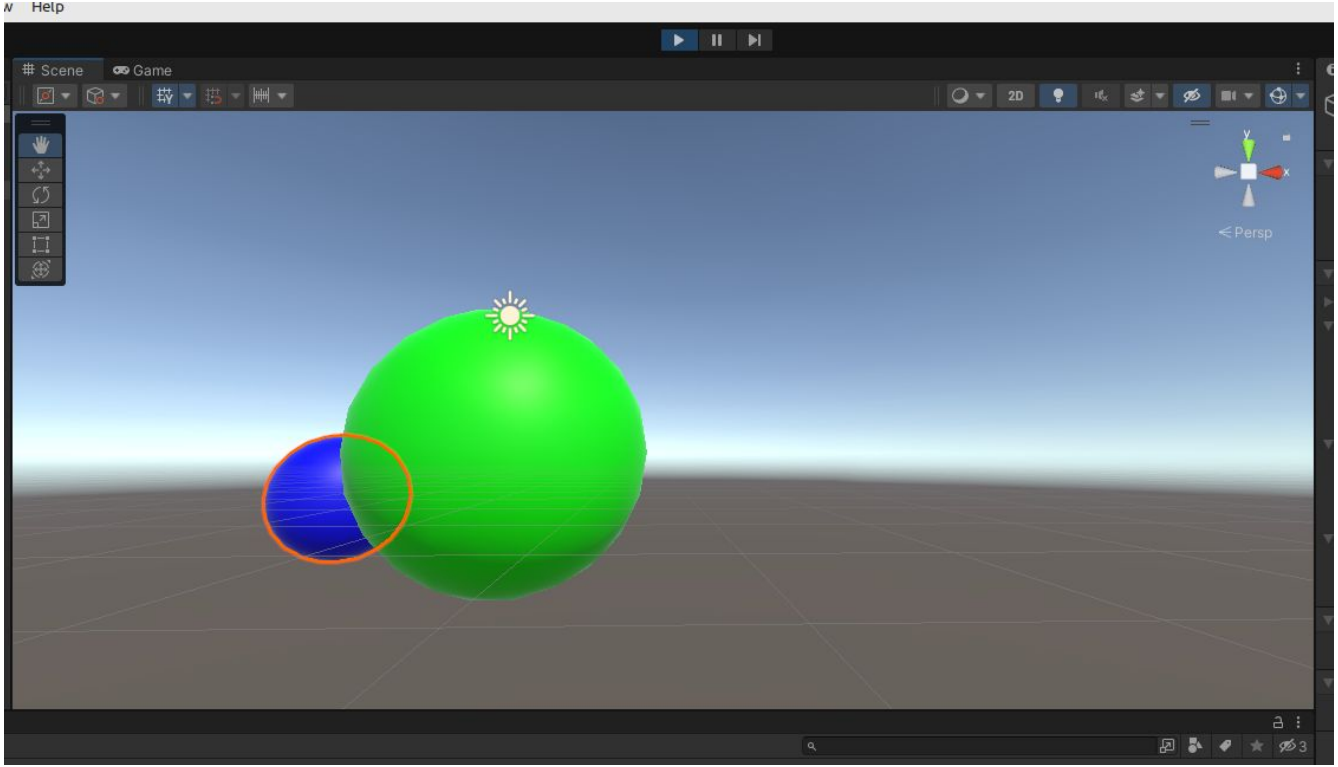
Switch to game mode and observe the change



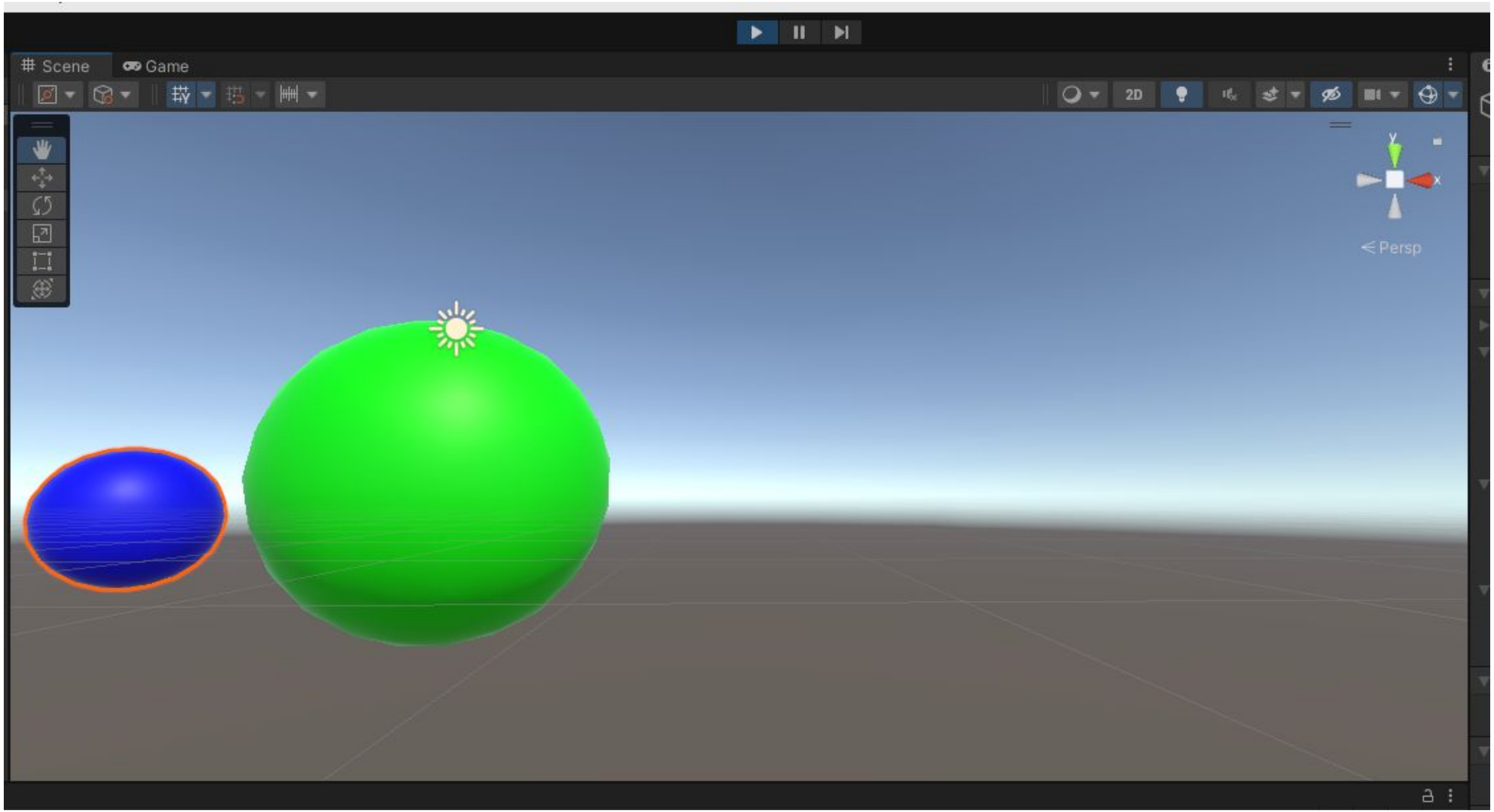
Investigate the effects from different viewing angles



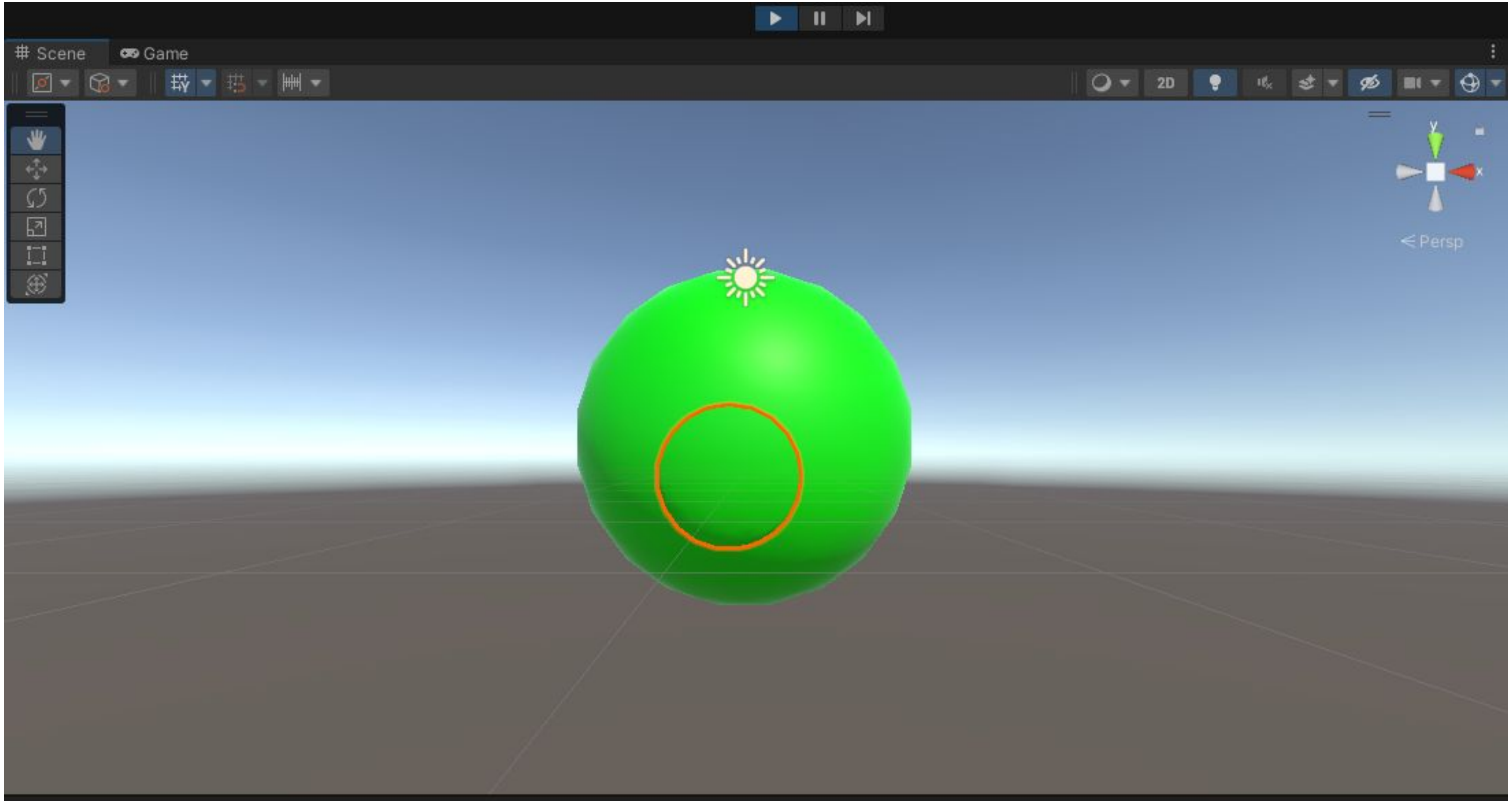
The small sphere disappears despite being placed in front of the big cube



Investigate the effects from different viewing angles



Why does this happen?



Core Idea of Z-Buffering Pseudocode

- **We maintain a two-dimensional array (the Z-buffer) that stores the depth (Z-value) of the closest object seen at each pixel so far**
- **When we draw an object, we calculate the depth of the object at each pixel. If this depth is less than the stored depth in the Z-buffer, we update the Z-buffer and draw the object. If it's greater, we skip drawing the object.**

Z-Buffering Pseudocode

Assets > zbuffer.cs

```
public class ZBuffering
{
    private float[,] zbuffer;
    private Color[,] framebuffer;
    private int width;
    private int height;

    public ZBuffering(int width, int height)
    {
        this.width = width;
        this.height = height;
        zbuffer = new float[width, height];
        framebuffer = new Color[width, height];

        //initialize Z-buffer to a large number
        for (int x = 0; x < width; x++)
        {
            for (int y = 0; y < height; y++)
            {
                zbuffer[x, y] = float.MaxValue;
            }
        }
    }

```

Z-Buffering Pseudocode

```
    public void RenderObject(MyObject obj)
    {
        foreach (Pixel pixel in obj.Pixels)
        {
            if (pixel.depth < zbuffer[pixel.x, pixel.y])
            {
                zbuffer[pixel.x, pixel.y] = pixel.depth;
                framebuffer[pixel.x, pixel.y] = pixel.color;
            }
        }
    }
}

public class Pixel
{
    public int x;
    public int y;
    public float depth;
    public Color color;
}

public class MyObject
{
    public List<Pixel> Pixels;
}
```