POC Writeup - Non-Euclidean World

Kevin Kwan
4/12/2021
Game Design 4th Period

Summary:

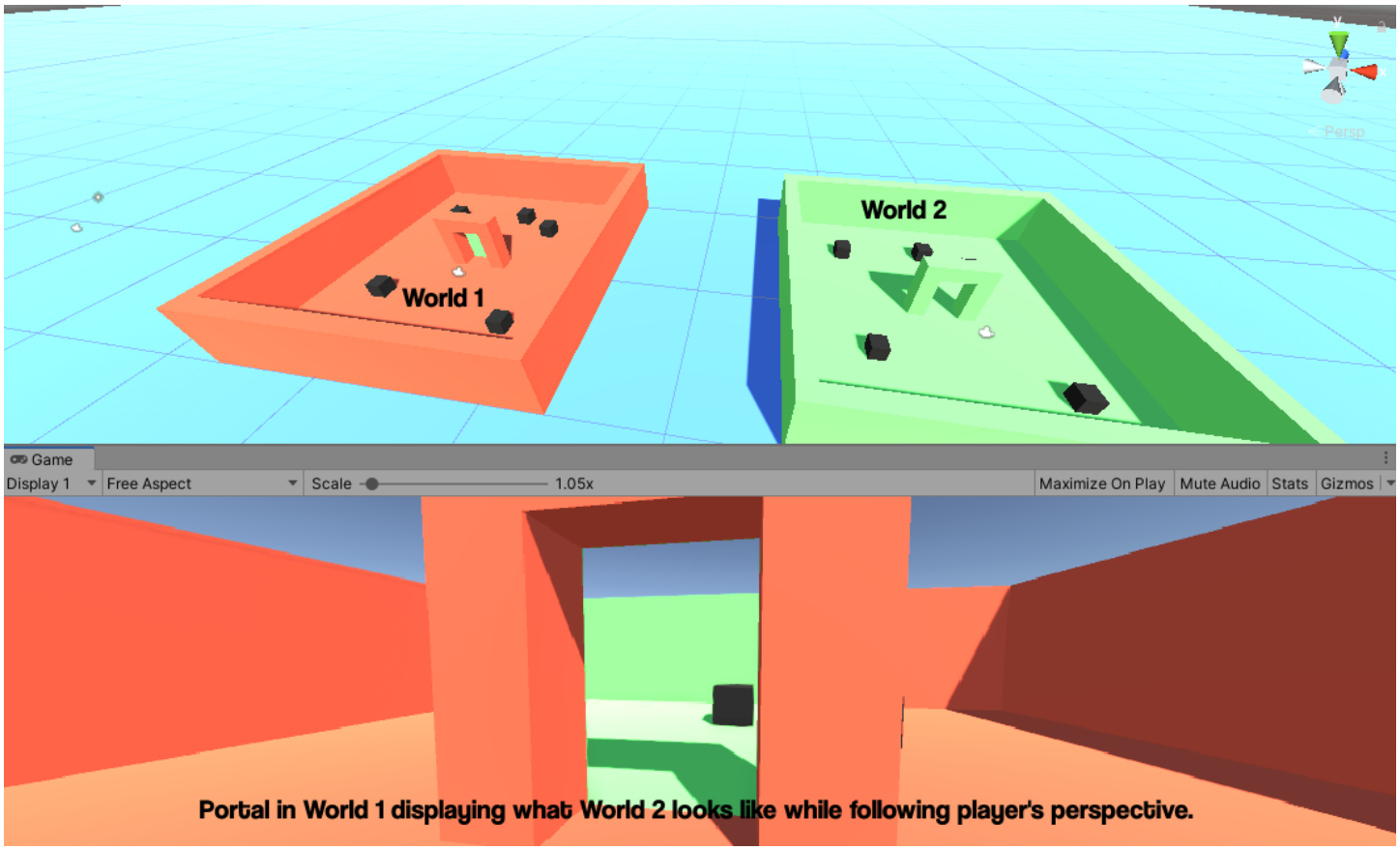
I will be creating a non-euclidian environment that players can interact with, look at, move around in, and interact with objects in them. The reality of the world will look warped and defy spatial reality as the player plays in a forced perspective where objects are not as they seem. Basically, the world is focused around convincing optical illusions to confuse the player's brain and eyes. For example, a player might walk through a simple doorway, but when they come out of it and look at the doorway that they came from, they will see that they came out of a long hallway instead of a doorway. If they look down the hallway, they will see the previous area/world that they came from that the doorway was in. Another example would be when a player walks around a wall corner and sees an object, like a ball. However, when the player continues walking and rounds the corner again, they will see a different object like a cube, even though the player thinks that nothing in the world has changed. This game mechanic can be used in games involving puzzle-solving, horror, mystery, and more. The following are three features/aspects of the topic that I attempted to implement.

1. Using smooth portals to help create convincing optical illusions between multiple worlds to make them seem like one world. (player's view and moving the player, basic functionality)
2. Allow the player to bring and interact with objects in the different "worlds." (special functionality)
3. Come up with convincing optical optical illusion concepts by building the worlds and using 3D objects (modeling, world design)

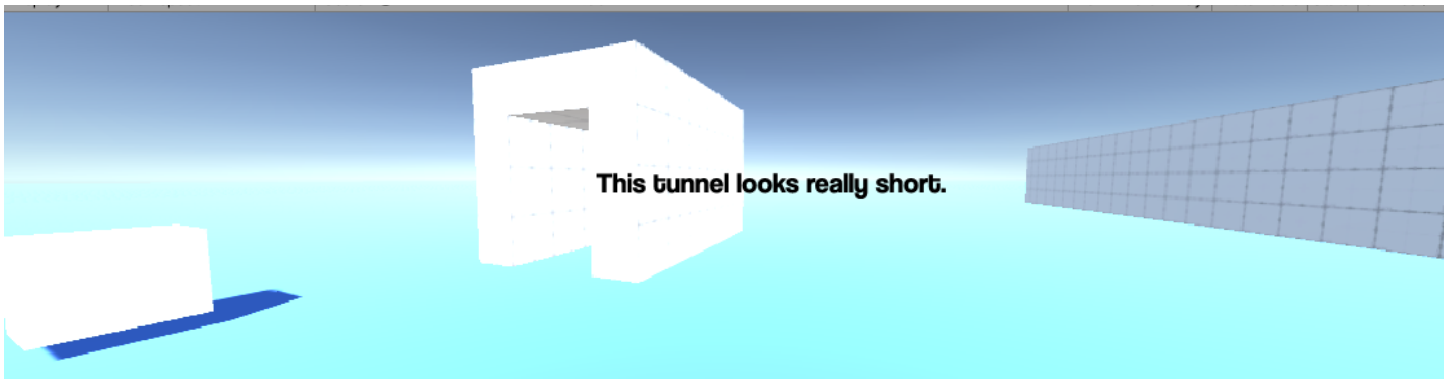
Implementation of the POC:

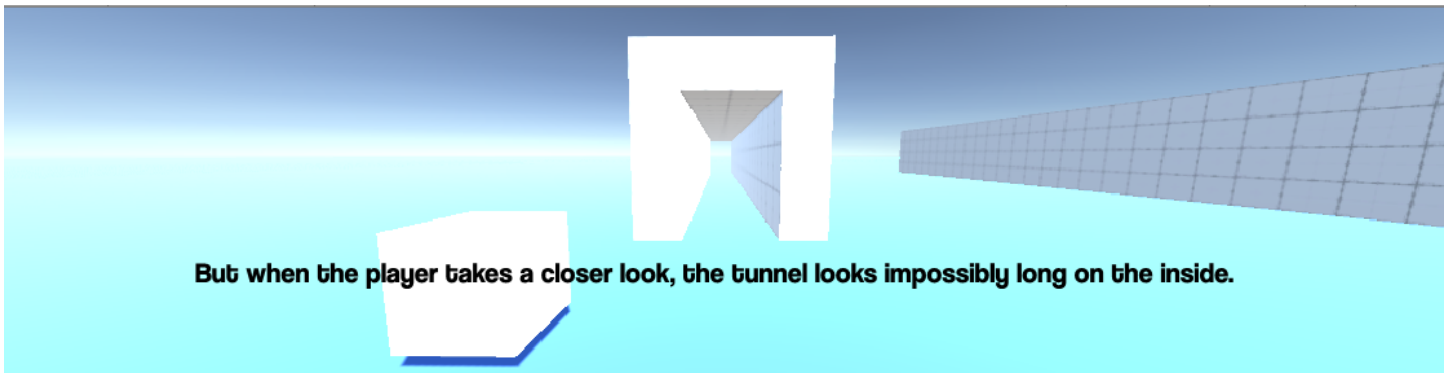
Unique Implementation Showcase:

Besides using the referenced tutorials I expanded upon the concepts and implementation. For example, Brackey's tutorial demonstrated using smooth portals to display the opposite world between World 1 and World 2 while allowing the player to seamlessly "move" between the two worlds by using the portal.

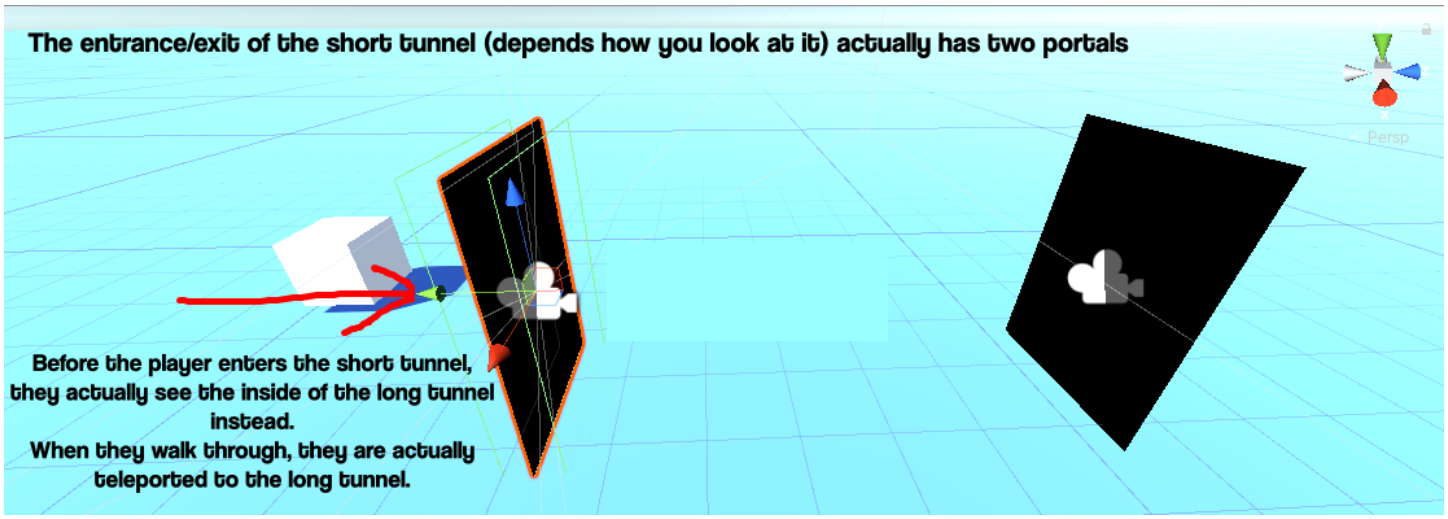


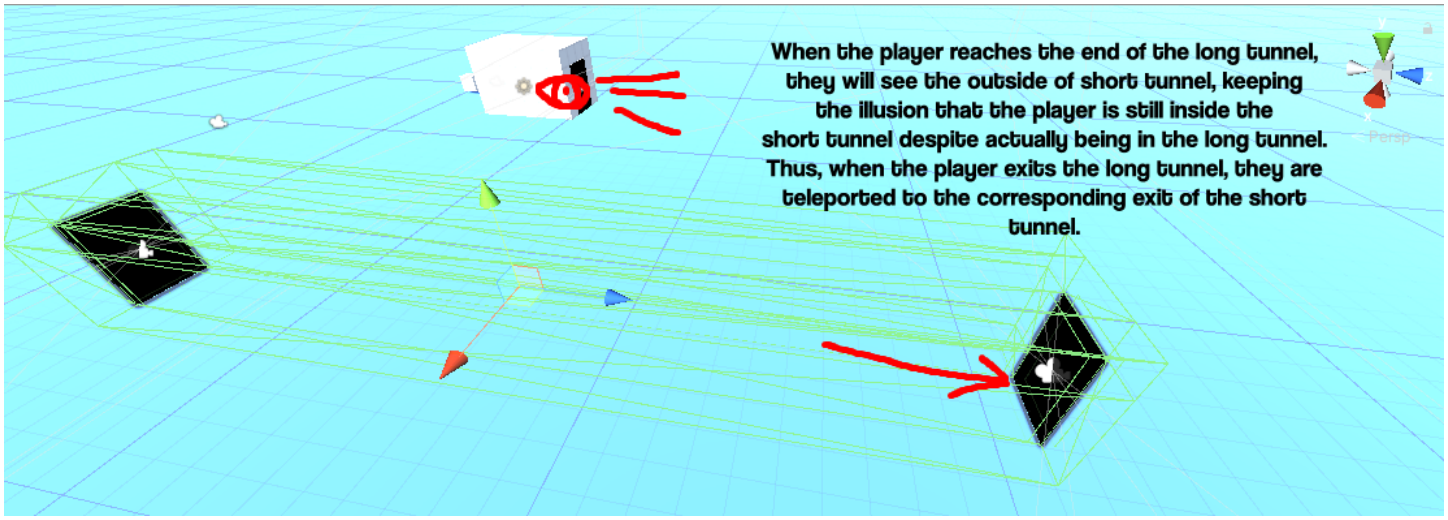
I used the basic functionality of these portals to create optical illusion rooms that I made myself. For example, here is a room where there are two tunnels. On the outside of one tunnel, it looks like a really short tunnel that the player can go through. However, when the player walks to the entrance of the tunnel, the tunnel on the inside looks way longer. There is actually a portal at the doorway that is showing the inside of the other tunnel. The player can walk through the portal without knowing it, and walk through the long tunnel. When they exit and look at the tunnel that they just walked out of from the outside, the tunnel will still seem short, which is impossible according to physics and logic. This is a non euclidean world example. The opposite occurs regarding the long tunnel.



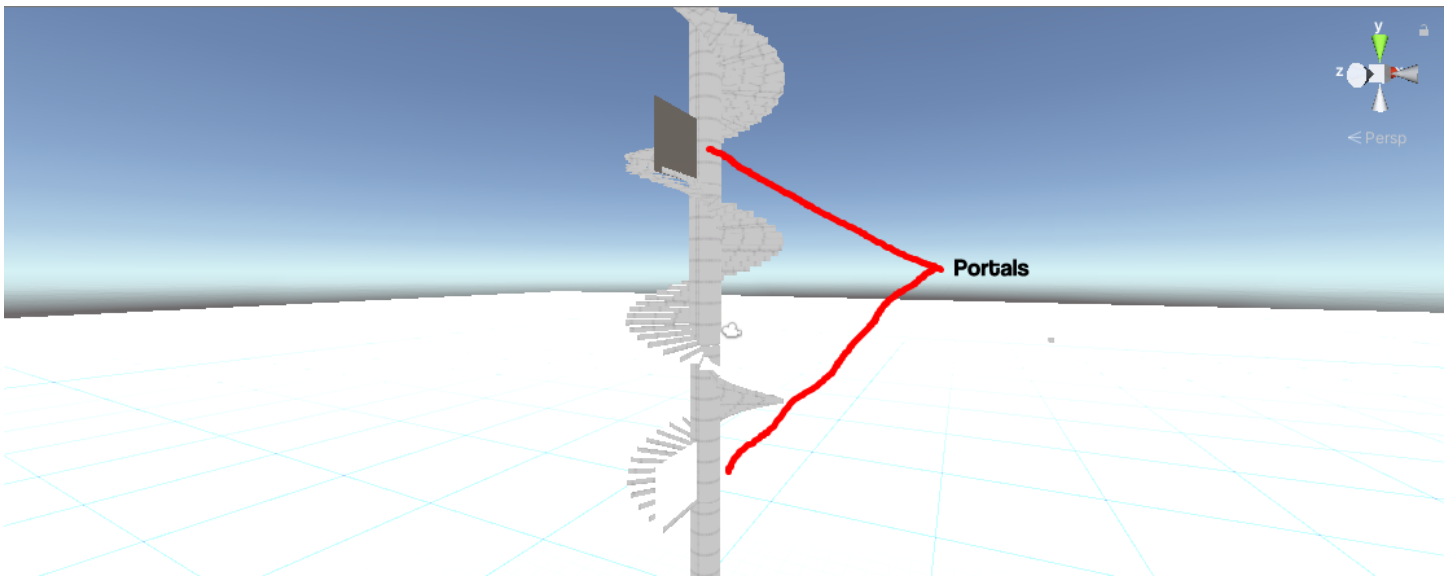


I also implemented a new functionality by using two portals at one location to display what the player sees from outside the tunnel looking at the portal and what the player sees from inside the tunnel looking at the portal. It's two portals facing away from each other but at the same location. There are two portals at each of the ends of the portals, for a total of 4 portals per tunnel. This allows the optical illusion to work even when the player enters and exits one of the tunnels.

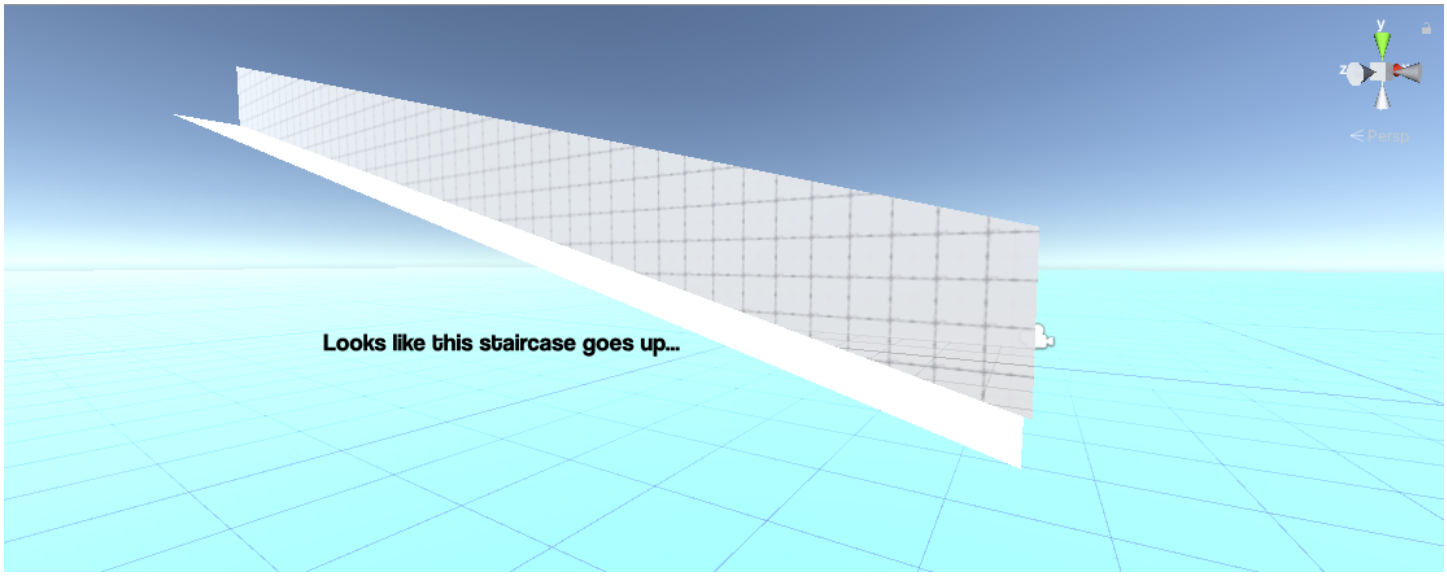


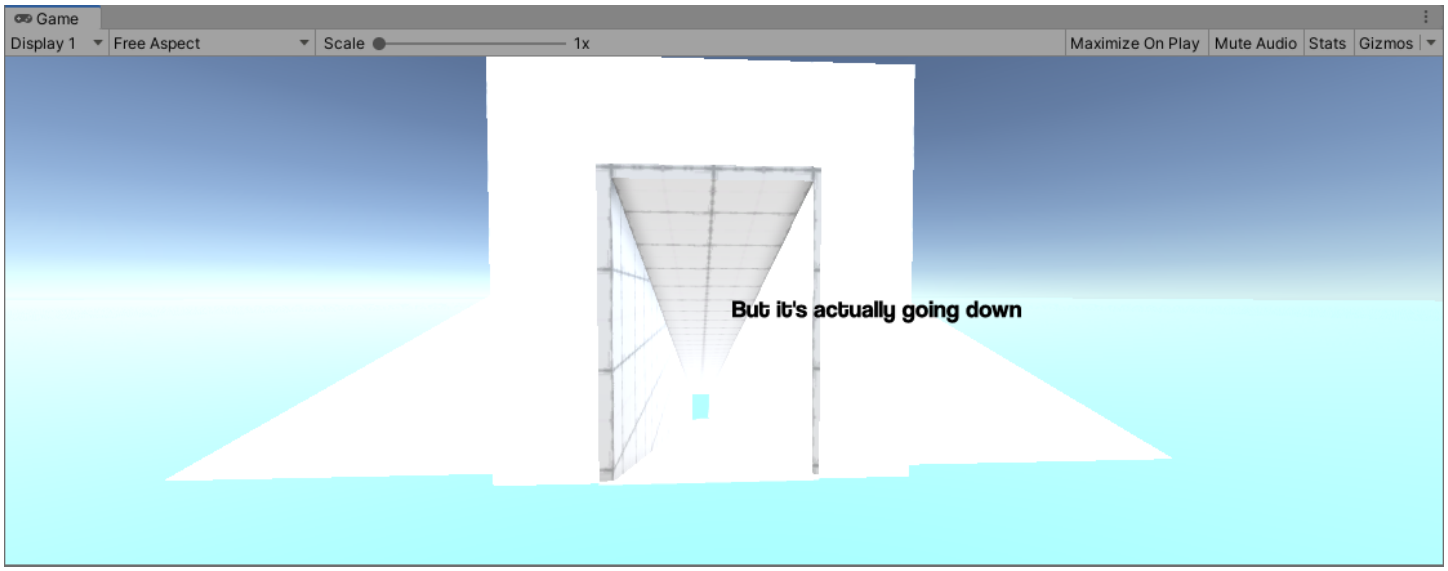


Another optical illusion/non-euclidean world that I made was the infinite spiral staircase. While on the outside, the spiral staircase is not infinite, and players usually expect stairs to go from point A and terminate at point B, with the use of portals at the top and bottom of the staircase, However, the player will find that if they take these flight of stairs, they are trapped in an infinite loop of going up or going down the stairs. The render planes of the portals handle the visual side of the optical illusion and hide the fact that there are portals that are doing this. The player unknowingly walks through these portals, teleporting to the opposite portal, and continue their journey.



Another unique implementation of these portals is in another optical illusion/non-euclidean world that I made. It involves a staircase tunnel that looks like it's going upwards on the outside when the player is on the ground floor.

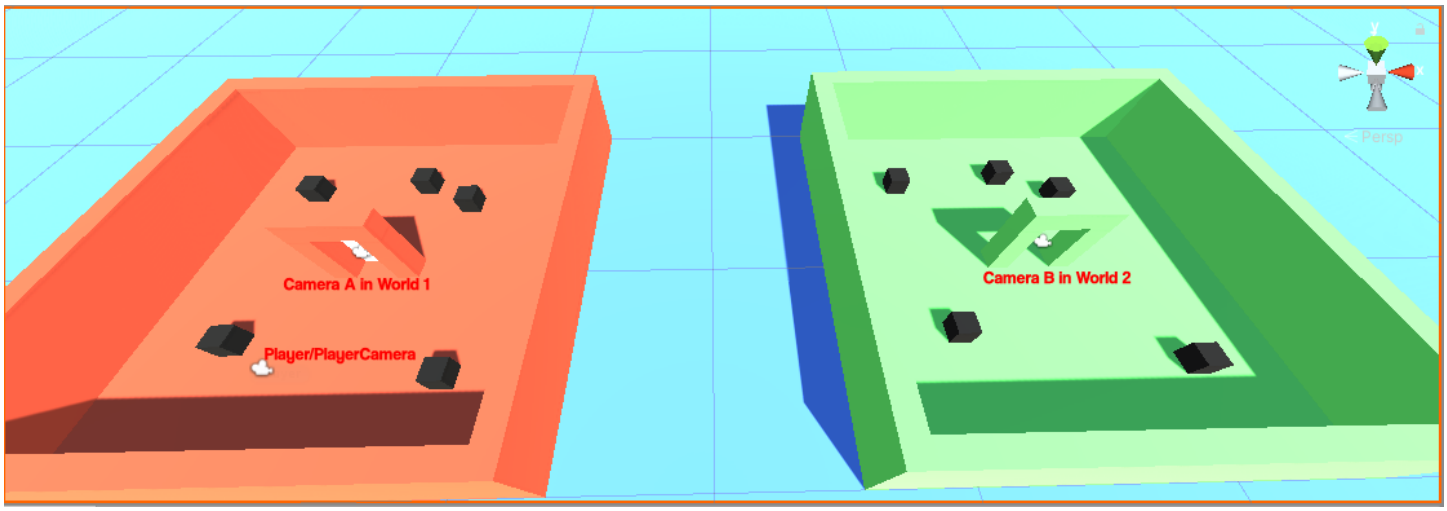


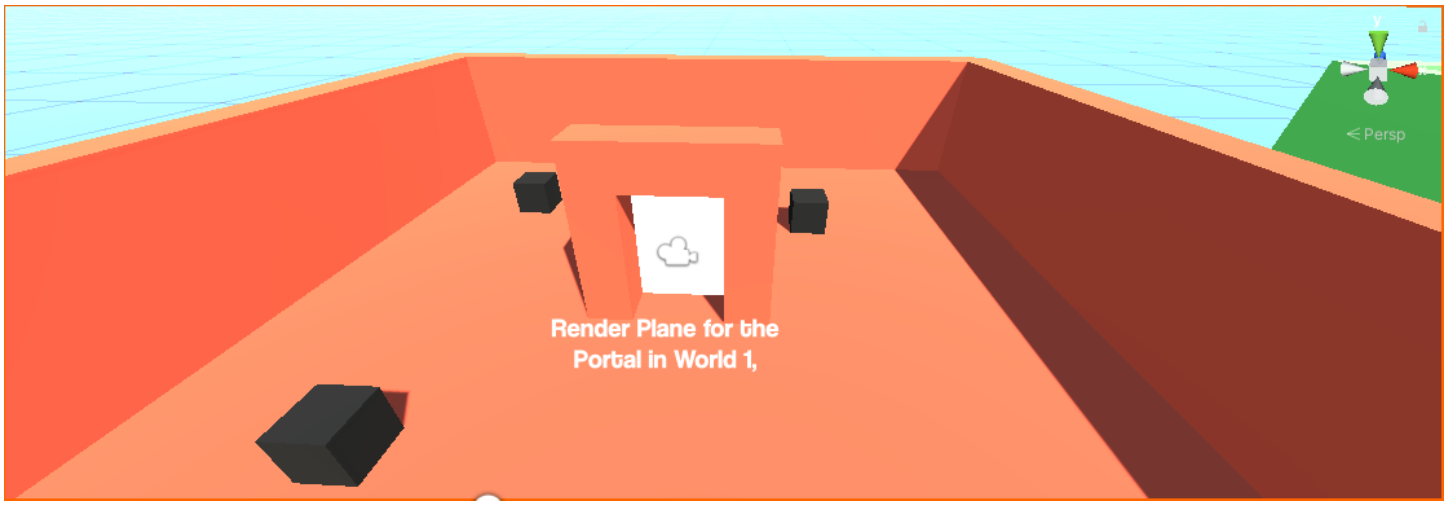


The vice versa occurs when the player is at the top of the staircase. They would assume that the stairs would go down, the stairs go up instead.

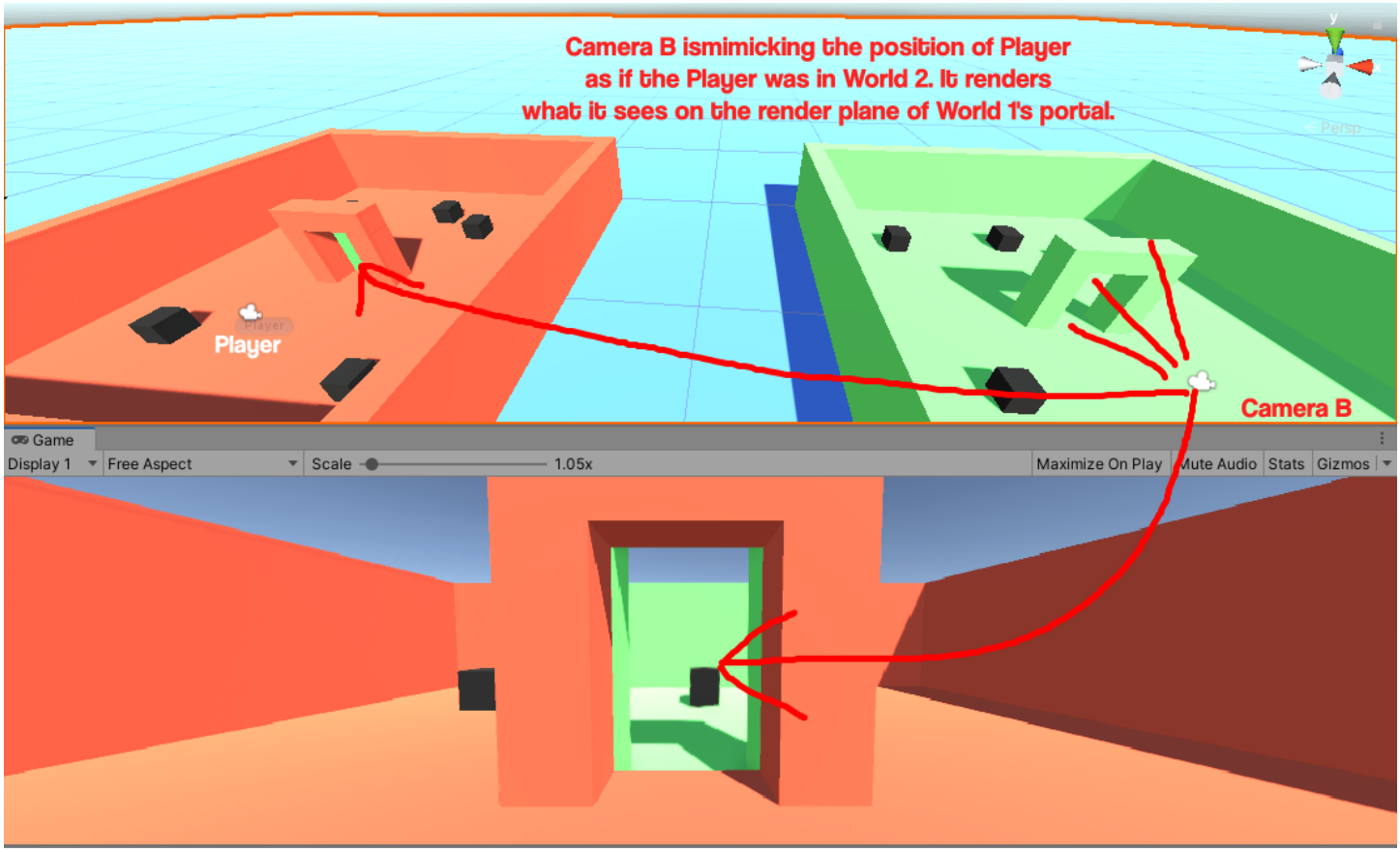
1. Implementation of Smooth Portals

First, I followed Brackey's tutorial on how to create smooth portals in Unity, and then, I expanded from it. Smooth portals are visual portals that players can look through and see the other location while following their perspective without having to walk through it. By walking through the portal, the player seems to smoothly "enter" another location. Unbeknownst to the player, they actually were teleported to a new location. Smooth portals are really good for creating optical illusions. In Brackey's tutorial, smooth portals are used between two worlds, with one being a mirror of the other one but a different color. The player starts off in one of the worlds. Each world has the identical objects and enclosed environments, including portal frames. There are two cameras, one for each world, which will be used to capture and display what they see in each world. Inside the portal frames, there is a render plane with an ordinary material that uses a shader provided by Brackey that renders a cutout (only the part that the player sees through the portal) of what the camera sees in the second world using screen coordinates.





We want the camera in the 2nd world to follow the position of the player as if they were in the 2nd world and render what it sees on the render plane of the portal in the 1st world.

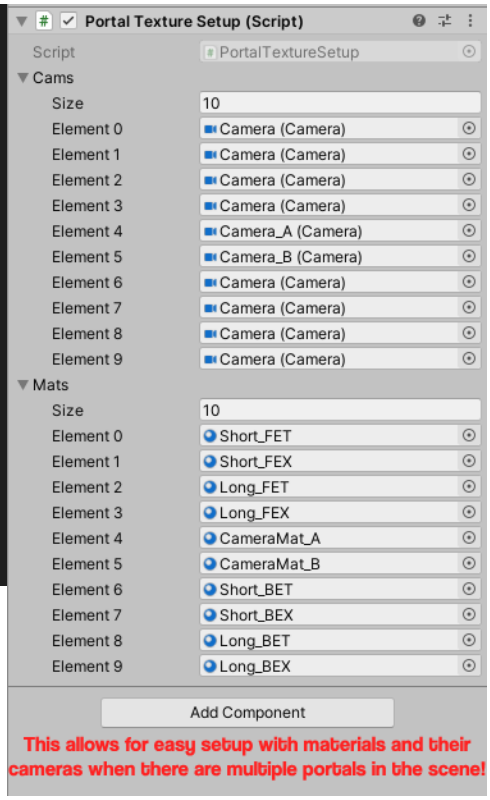


In order to do this, a script is needed to have the portal cameras follow the position of a player in relation to its own portal. The script also offsets the rotation of the portal cameras to match the player's camera rotation in order to keep the player's perspective as if they were in the opposite world.

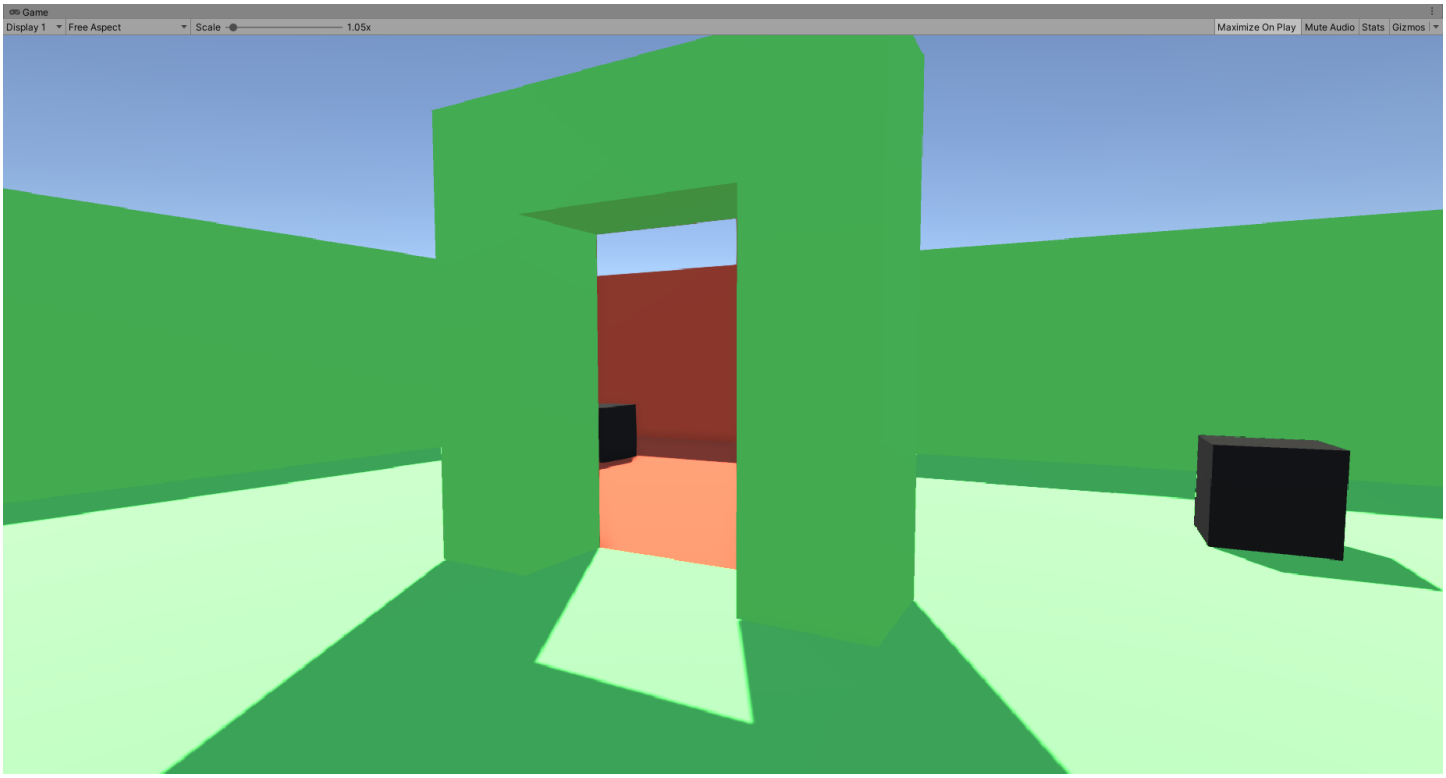
```
void LateUpdate()
{
    Vector3 playerOffsetFromPrtal = player_cam.position - otherPortal.position;

    if (!neg)
        transform.position = portal.position + playerOffsetFromPrtal;
    else
    {
        transform.position = new Vector3(portal.position.x, -portal.position.y, portal.position.z) - new Vector3(playerOffsetFromPrtal.x, -playerOffsetFromPrtal.y, playerOffsetFromPrtal.z);
    }
    float angularDiff = Quaternion.Angle(portal.rotation, otherPortal.rotation);

    Quaternion portalRotDiff = Quaternion.AngleAxis(angularDiff, Vector3.up);
    Vector3 newCamDir = portalRotDiff * player_cam.forward;

    transform.rotation = Quaternion.LookRotation(newCamDir, Vector3.up);
```

The following script allows the developer to manually assign and set up the corresponding cameras and what they see to their complementary render plane materials. For example, Camera B will correspond to the material that is displayed on the render plane in World 1, while Camera A will correspond to the material that is displayed on the render plane in World 2.

```
public class PortalTextureSetup : MonoBehaviour
{

    public List<Camera> cams = new List<Camera>();
    public List<Material> mats = new List<Material>();

    Unity Message | 0 references
    void Start()
    {
        for (int i = 0; i < cams.Count; i++)
        {
            if (cams[i].targetTexture != null)
            {
                cams[i].targetTexture.Release();
            }
            cams[i].targetTexture = new RenderTexture(Screen.width, Screen.height, 24);
            mats[i].mainTexture = cams[i].targetTexture;
        }
    }
```



Here is what World 1 looks like through the portal in World 2.



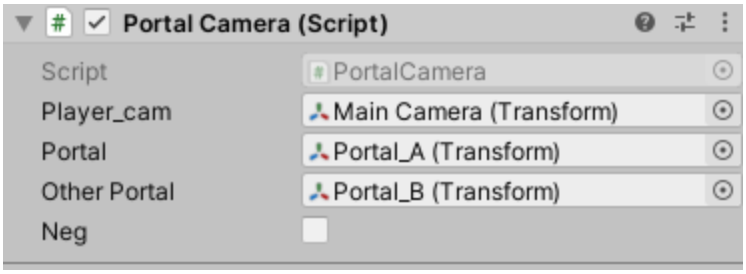
That is the visual aspect of the portal implemented!

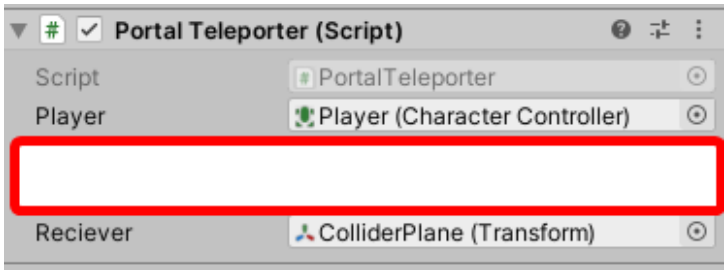
Now, regarding the teleportation, an invisible collider exists at each portal at the same location as the render plane. A script is used to check if the player is overlapping or exiting the collider, check whether or not they moved across the portal, and teleport the player to the opposite portal while keeping their direction and rotation using dot products to get vectors and offset calculations.

```
void Update()
{
    if (playerIsOverlapping)
    {
        Vector3 portalToPlayer = player.transform.position - transform.position;
        float dotProduct = Vector3.Dot(transform.up, portalToPlayer);

        // If this is true: The player has moved across the portal
        if (dotProduct < 0f)
        {
            // Teleport him!
            float rotationDiff = -Quaternion.Angle(transform.rotation, reciever.rotation);
            rotationDiff += 180;
            player.transform.Rotate(Vector3.up, rotationDiff);
            //Keep the player's rotation orientation
            Vector3 positionOffset = Quaternion.Euler(0f, rotationDiff, 0f) * portalToPlayer;

            player.transform.position = reciever.position + portalToPlayer;


            playerIsOverlapping = false;
        }
    }
```

Then, since there are public variables, the developer can assign and link up the portals and cameras by simply dragging them into the inspector.

Examples include:



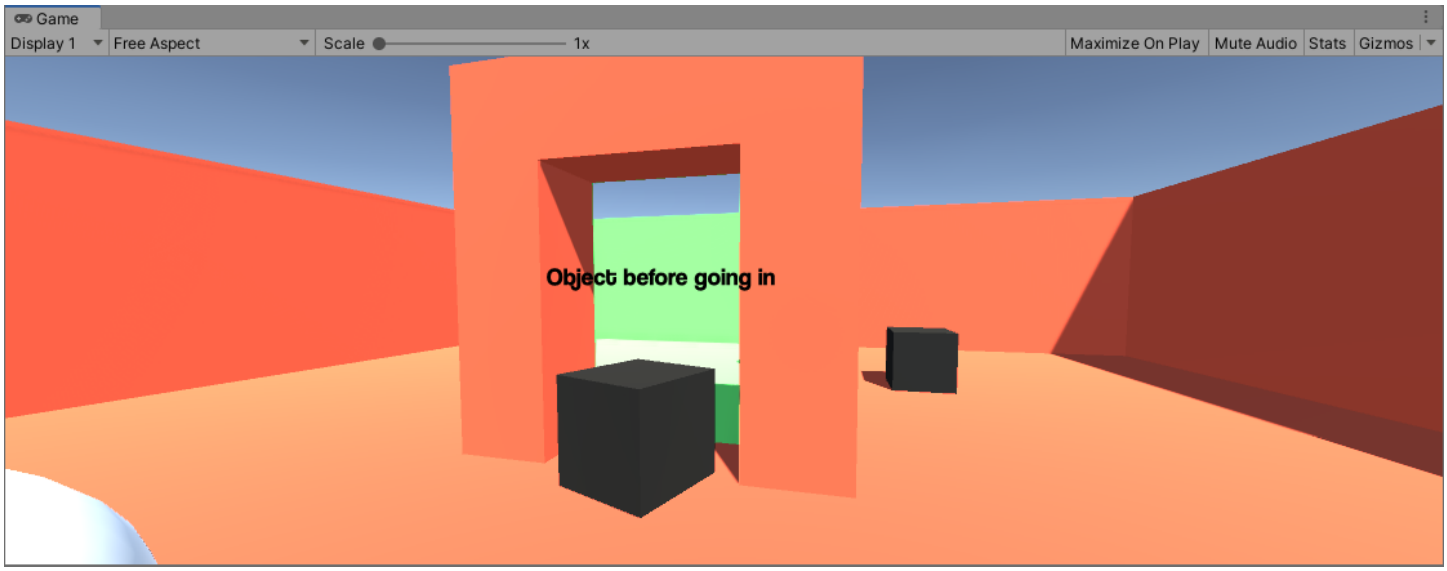


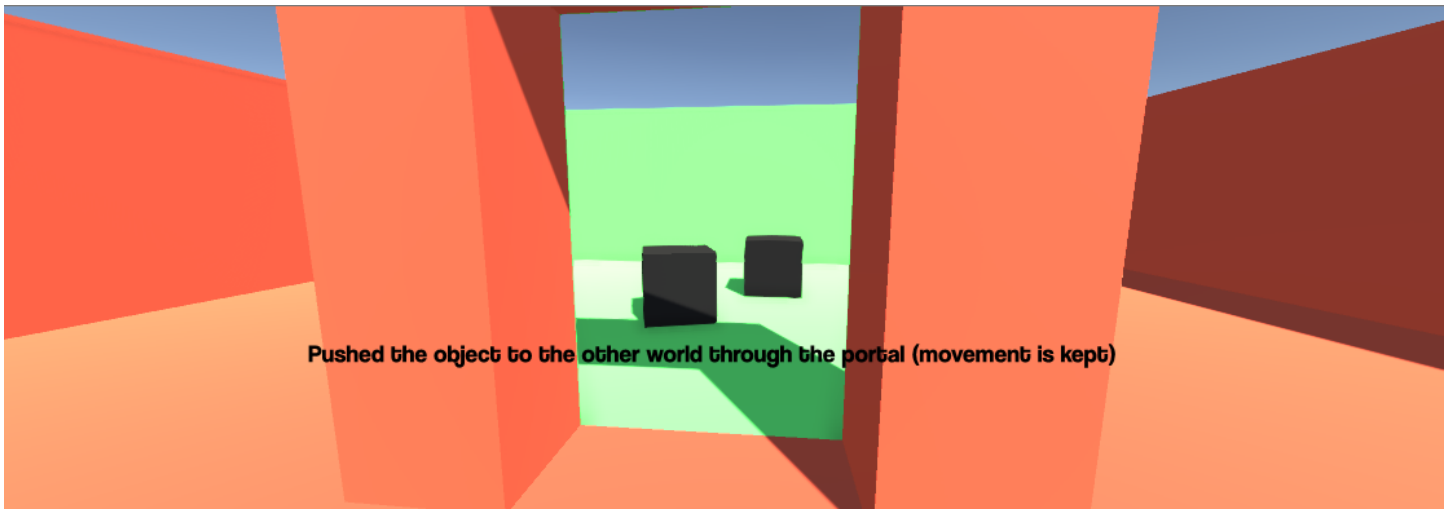
2. Implementation of Interactable Objects going through Portals

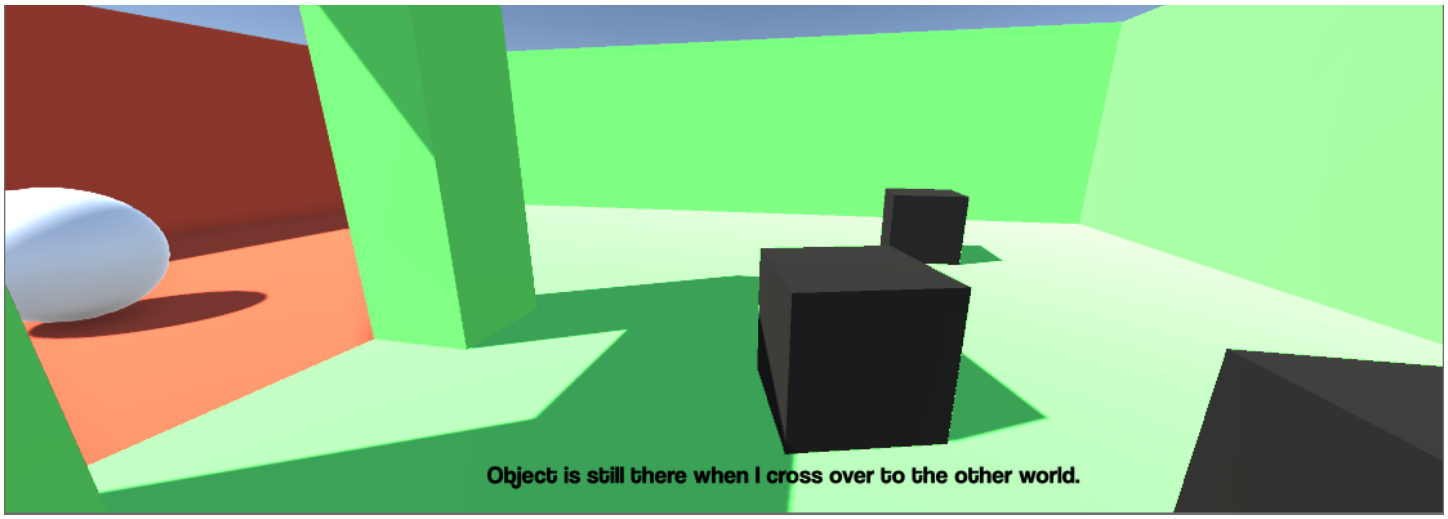
Using the same concepts of Portal Teleportation, I modified the code used to smoothly teleport the player when they go through portals to include objects as well.

```
if (blockIsOverlapping)
{
    Vector3 portalToBlock = block.transform.position - transform.position;
    float dotProduct = Vector3.Dot(transform.up, portalToBlock);

    // If this is true: The block has moved across the portal
    if (dotProduct < 0f)
    {
        // Teleport him!
        float rotationDiff = -Quaternion.Angle(transform.rotation, reciever.rotation);
        rotationDiff += 180;
        block.transform.Rotate(Vector3.up, rotationDiff);
        //Keep the block's rotation orientation
        Vector3 positionOffset = Quaternion.Euler(0f, rotationDiff, 0f) * portalToBlock;

        block.transform.position = reciever.position + portalToBlock;


        blockIsOverlapping = false;
    }
}
```

The blocks flawlessly travel through the portal and show up visually on the render plane well. Their movement is kept as well.







3. Implementation of Optical Illusions using Smooth Portals

Using Brackey's tutorial world and portals as inspiration, I thought of the common optical illusions that I have seen in other non-euclidean games and tried to mimic them using the tools available. I created the structures like the tunnels and stairs using Unity's ProBuilder tool to model them. The implementation of the non-euclidean world concept and optical illusions are discussed earlier in the writeup in the "Unique Implementation" section.

References:

Brackeys - Smooth PORTALS in Unity: https://www.youtube.com/watch?v=cuQao3hEKfs

CodeParade - Non-Euclidean Worlds Engine: https://www.youtube.com/watch?v=kEB11PQ9Eo8

JayCode - Non-Euclidean Game using Unity: https://www.youtube.com/watch?v=rvAhM9ynbSc