Portals

Kento Nishi, Hanna Pak
CS 175 Final Project (Spring 2023)

Level Building

This level was built mostly in Unity, but the portal was made using Blender. To create the portal, we first modified the standard “torus” shape in blender, then created a second circular plane to act as the “mirror” through which the player can observe the other side of the portal. The portal was then exported as an .fbx to Unity.

In Unity, to build the level, we first created the outline of the space with various planes and modified cubes, drawing inspiration from the first level in the Portal game series on which this project is based. The player was represented by a small capsule object, as is standard game development practice. We then created the various materials to be used around the level. The glass, player capsule, and toxic waste materials were all made from scratch. The concrete materials came from the Unity asset store, since they had matching albedo, normal, height, and occlusion maps to create a realistic concrete effect. The materials used were modified slightly for color grading purposes, and to ensure uniform tiling across game objects of varying size. Hanna had a magic mesh effect asset pack downloaded from a previous Unity project, so we heavily manipulated one of the effects from that pack to create the fire-like animation on the border of the portals. Finally, we imported fences from the asset store to serve as barriers in the second half of the level.

Player Movement

Player movement is split into two components (and their respective scripts): Player Movement and Player Look. For movement, we made use of Unity's character controller system, which detects WASD key presses as inputs to vertical or horizontal axes. We created a Vec3 (PlayerInput) from these inputs with the vertical world motion set to 0, since the player cannot jump with WASD. We then normalized this vector and used it to create another Vec3 (MoveVector) that stores what direction the player will move in relative to the world frame. MoveVector was then used to create yet another Vec3 (CurrentMoveVelocity), which takes into account walking speed and smooths the motion. Finally, the player's location is updated using CurrentMoveVelocity and multiplying it by the time per frame, ensuring uniform motion. To keep the player on the ground, we also implemented gravity. Checking whether the player is on the ground is done by emitting a ray cast going straight down from the player's current position, yielding a boolean indicating whether that ray has hit another rigid body (i.e. the ground). If the ray cast returns true, we simply apply a small constant force to ensure the player remains on the ground. However, if the ray cast returns negative and the player is in the air, then the y-coordinate of the Vec3 CurrentForceVelocity is set to gravity * time * time, realistically implementing the time^2 component of gravity. This goes unused most of the time, except for when the user "pops out" of a portal (as described in the next section).

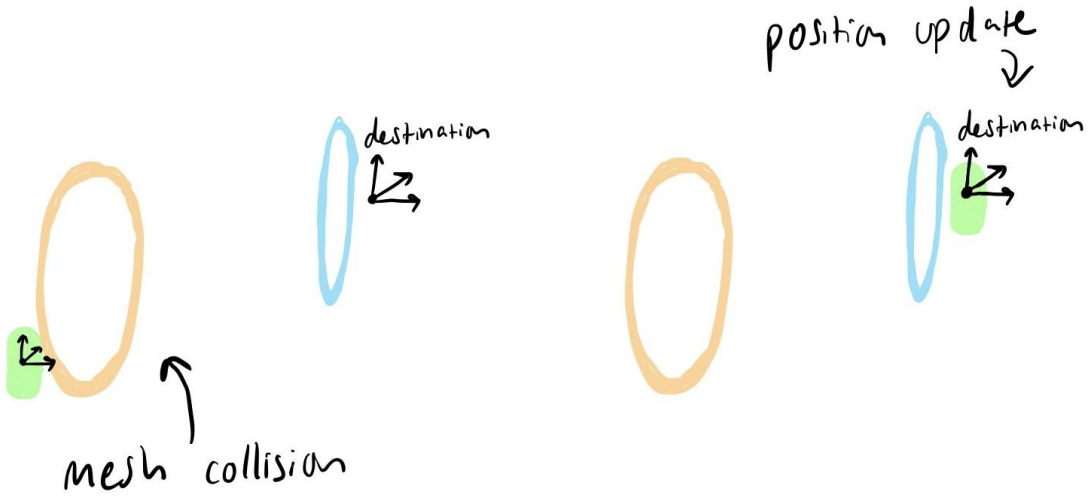
Teleportation Logic

To begin teleportation implementation, we first created "destination" game objects. Each destination is the child of its corresponding portal, and while these are empty game objects, their location is slightly in front of and above the center of the portal's location in the game.

The script applied to each portal detects when the player collides with the portal's mesh, upon which the player game object is momentarily set to inactive. The player's position is then updated to equal the position of the corresponding portal's "destination" game object.

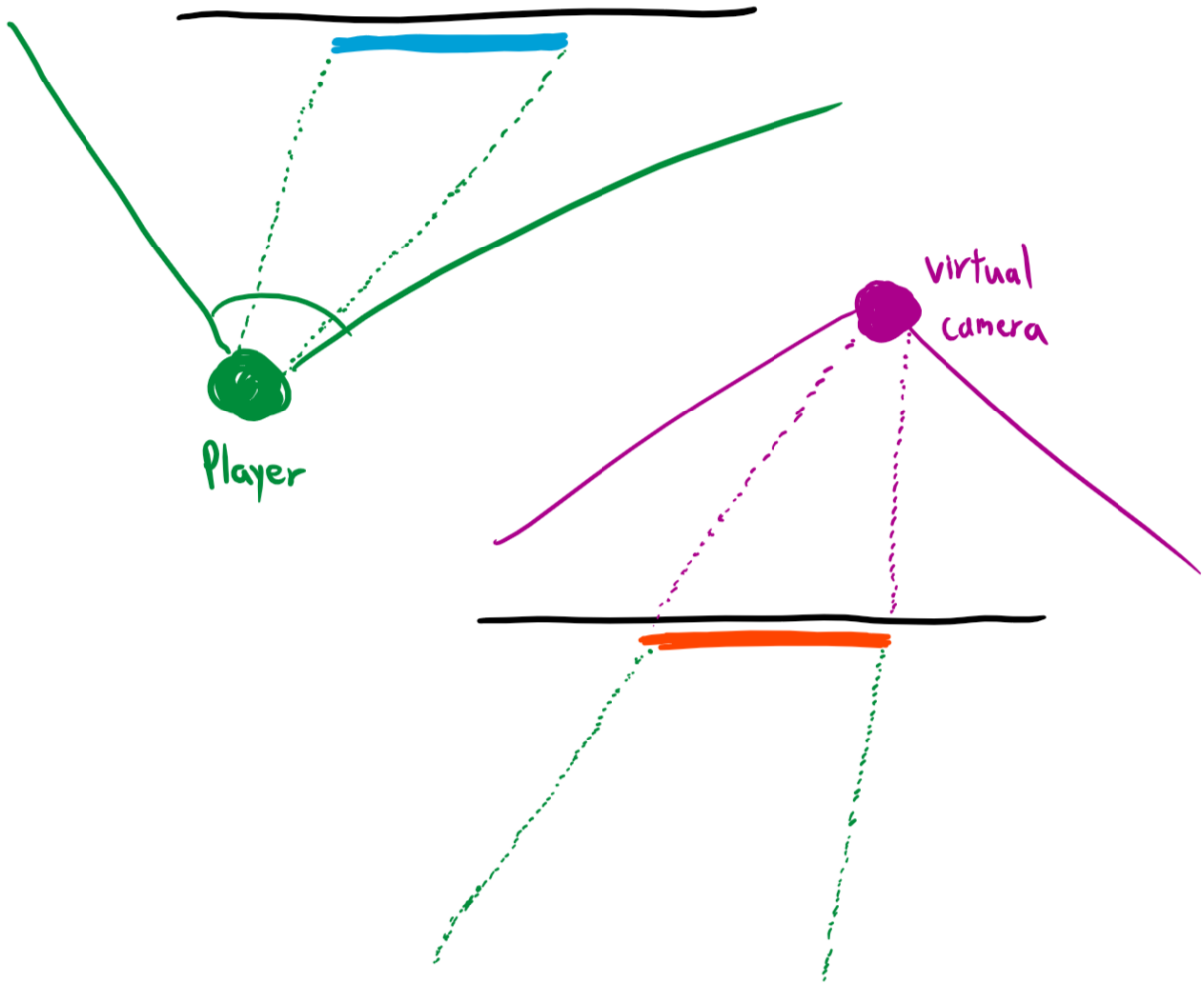
Additionally, the player's rotation relative to the entry portal is recorded and reapplied relative to the exit portal to preserve the player's perceived rotation (so that they are looking the same way relative to the portal they exit). Finally, the player object is reactivated and the player has teleported from one portal to the next.

The "destination" game object teleportation was implemented to prevent against clipping through planes; if the player is teleported so that their mesh intersects with or crosses through a plane, they could simply fall through the world or clip through walls, so setting the destination in front and above the portal ensures that the player will teleport successfully. The height of the destination is also not exactly calibrated to the height of the player, so the gravity term from the previous section keeps the player on the ground.

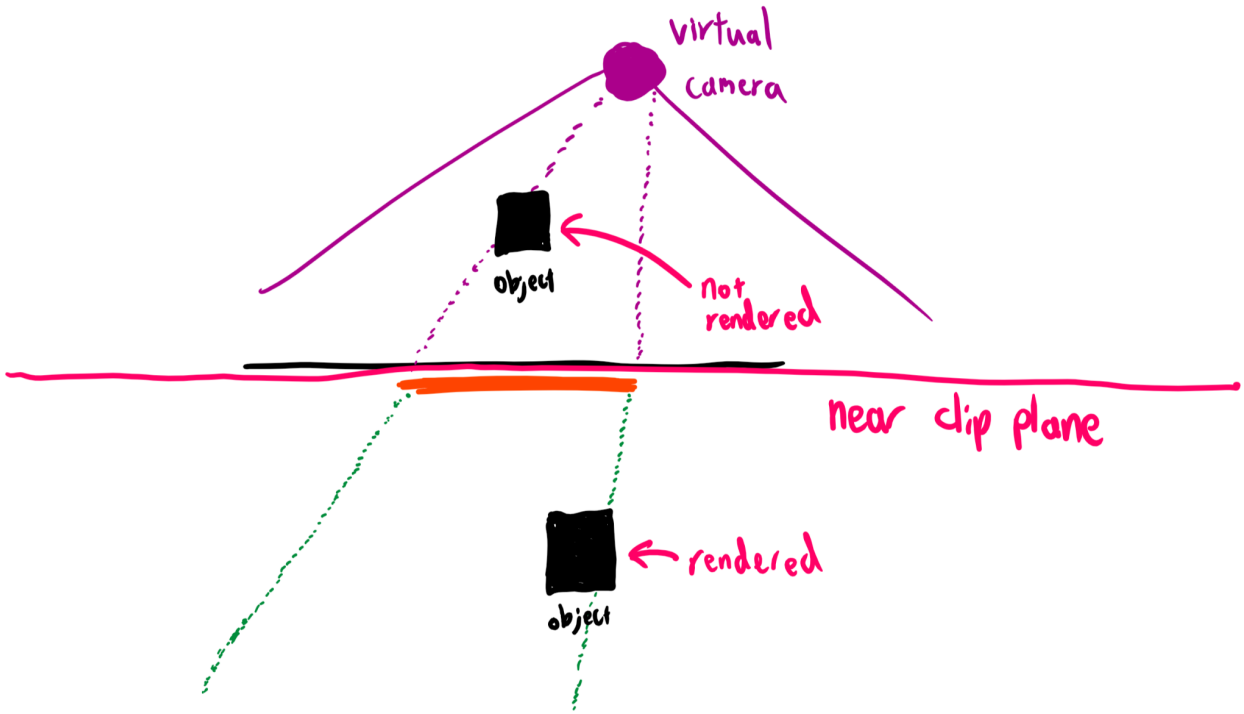


Portal Rendering

To render portals, we wrote a shader which effectively "fakes" viewing through the portal by painting an image on the surface of the portal. To achieve this effect, we use a set of virtual cameras which render directly into a texture. As the player moves through the world, the virtual camera position is set relative to the exit portal based on the player's camera position relative to the entry portal. Below is a diagram of how this logic works:



When rendering the virtual camera's view to a texture, we also need to get rid of objects which lie closer to the camera than the portal surface as they may obstruct the camera's view. To do so, we use an "oblique projection matrix," which effectively adjusts the virtual camera's near-clip plane to remove all objects which lie between the camera and the portal surface. We followed this tutorial for implementing the oblique projection matrix. Below is a diagram for visualizing this behavior:



It is important to note that placing the image captured from the virtual camera directly onto the portal surface does not result in a convincing portal effect. This is because the camera's field of view does not correspond to what would be in view when looking through a portal. To correctly account for the field of view, we use screen-space coordinates to effectively crop the camera's captured image to the dimensions of the portal. Then, using a mesh shader, we can directly render this cropped image onto the portal surface. With this, a basic implementation of portals is complete.

Potential Improvements

While our project is (in our humble opinion) pretty cool, there are notable issues with our implementation. First, because we don't precisely teleport the player by measuring their position relative to the entry portal, there is a somewhat noticeable "jump" in the visuals when walking through a portal. Some amount of visual jumping is unavoidable, but this can certainly be reduced with better teleportation logic. On the visual side, we did not implement recursive

rendering in the interest of time; therefore, when looking through a portal through a portal, the surface is pitch black. This can be fixed by implementing a loop which iteratively renders multiple passes through a portal. Finally, portals are statically placed—our renderer is placement-agnostic, so a “portal gun” can be added with some additional work. While these features are not included in our submission, we may explore adding these mechanics for fun down the line.

The End

PS. We want to say thank you to Schlomo and the TFs of this course! We learned a lot throughout the semester, and we had a lot of fun putting what we learned to good use with this final project. Have a great summer break :)

- Kento, Hanna