Rendering Portals

Reed Metzler-Gilbertz

Rensselaer Polytechnic Institute

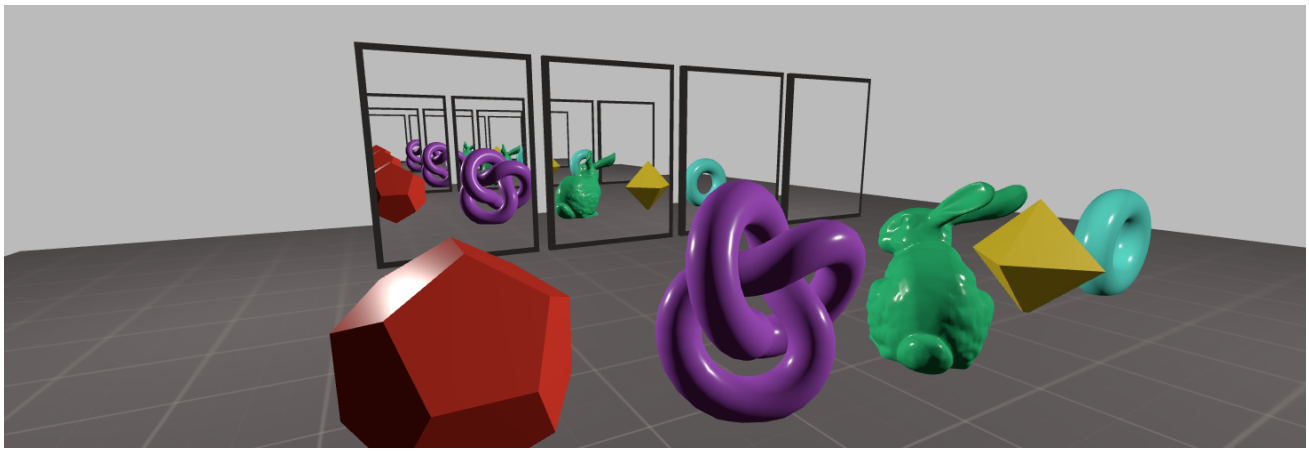


Figure 1: *A scene with several recursive portals*

Abstract

In this paper, I describe a method for accurately rendering interactive planar portals, as well as a method to detect and execute seamless teleportation of a camera from a portal to its destination. I also briefly provide some examples as to how portals can be used to create illusions of non-euclidean geometry.

1 Introduction

This project focuses on a method for accurately rendering planar portals in a traditional, OpenGL derived rendering environment. A portal is a surface within a scene that acts as a "window" into another part of the scene. Rendering portals using standard, rasterized rendering can be very tricky, as multiple render passes need to be carefully executed in order to create the illusion that a hole leading to some other area has been cut into the world. There are several challenges and special cases that the rendering method must also be able to handle in order to maintain the believability of the effect, such as portal recursion (seeing one portal through another portal).

In addition to rendering portals, I explore how the portals themselves can be made "functional", meaning the viewer is able to seamlessly walk into the area that is being rendered on the surface of a portal. Both the rendering of the portals and the method of teleporting the player between the different areas a portal divides must ideally be subtle enough that the viewer may be unaware they have even moved through a portal at all.

This project was implemented using THREE.js and WebGL, and runs in any modern web browser that supports WebGL.

2 Related Work

Much of the prior academic work on rendering portals relates to a technique known as portal culling, a method that divides a scene up into connected "cells" separated by portals. This partitioning of the scene can then be exploited to reduce the amount of geometry that needs to be rendered. [Aliaga and Lastra 1997] presents a technique that renders the visible geometry in cells adjacent to the viewer using a texture displayed on the portal's surface. This use case for portals, however, is not very relevant to this project as that method sacrifices accuracy of the geometry rendered to the portal in favor of performance, whereas my goal is for portals to render geometry with as much accuracy as possible.

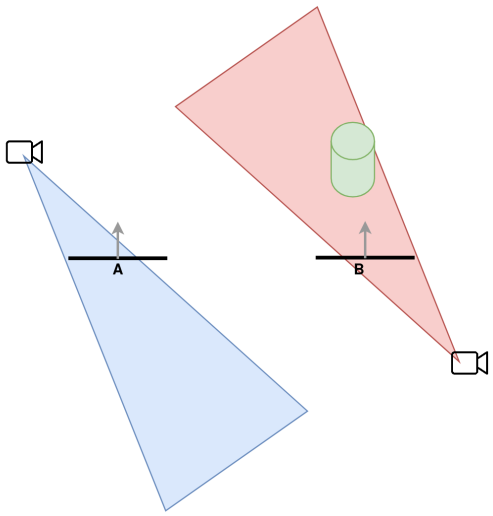
[Lowe and Datta 2005] presents a generalized method for rendering non-planar and non-convex transformative portals. My method was heavily inspired by their algorithm, with some of the main differences being due to the fact that I am only rendering planar portals.

Video games are another area where portals have made several appearances. The Portal series by Valve is one example of a set of games based around interactive portals. Some of the details and challenges associated with the portal system created for those games are discussed in a lecture for Harvard's CS50 course given by two developers who worked on the games [Kircher and Kohli]. Much of that talk is focused on how they managed to accurately simulate physics through portals, however this is beyond the scope of my project and not necessary for the results I'm hoping to achieve.

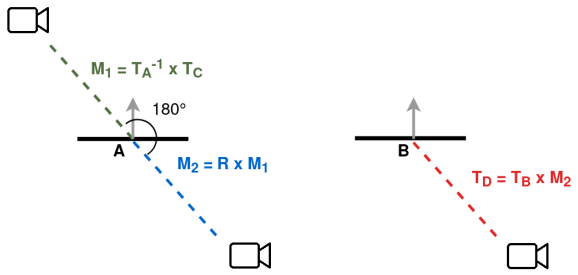
3 Portal Basics

It's important to have an intuition as to how a portal actually works. Imagine you, a camera, are in a scene with two portals, portal A and its destination, portal B. This is represented in Figure 2a, where the main camera (you) and its view frustum are drawn in blue. On the other end of portal B there is a green cylinder. As you look through portal A (which remember leads to portal B) what you see displayed on the surface of portal A can be rendered by another camera view, whose frustum is drawn in red. The world transformation matrix for this other camera, T_D, is calculated as follows:

$$T_D = T_B \cdot R \cdot T_A^{-1} \cdot T_C \tag{1}$$



(a) *Main camera perspective (blue) and portal A's destination camera perspective (red)*



(b) *Steps to convert main camera transform T_C to portal A's destination camera transform T_D*

Figure 2

In this formula, T_C represents the original camera's world transformation matrix. T_A^{-1} represents the inverse world transformation matrix of portal A and T_B represents the world transformation matrix of portal B. Finally, R represents a rotation matrix that applies a 180 degree rotation.

In order to make this formula more intuitive, Figure 2b splits it up into each of its three matrix multiplications. First T_A^{-1} is multiplied by T_C in order to get the transform of the original camera relative to portal A, M_1 (drawn in green). Next, R is multiplied by M_1 to get M_2 (drawn in blue), which rotates the transform of the camera by 180 degrees relative to portal A, placing the camera locally behind portal A (exactly opposite of where it was relative to portal A originally). Finally, we multiply T_B by M_2 to apply the transform of the camera local to portal A to portal B, giving us the final transformed world matrix, T_D (drawn in red relative to portal B).

4 Rendering

The rendering algorithm I have come up with is derived from the method presented by [Lowe and Datta 2005]. One of the key differences is that the algorithm I present does not split the scene up into individual cells, as is common when using portals, since partitioning a scene in such a way can potentially save lots of unnecessary render calls in large scenes with many portals. However, it would be straightforward to extend my method to support individualized cells, as currently it essentially just treats the entire scene as one cell.

The rendering of the scene, including the portals, is handled within the render() function. As input, the function takes the camera world transformation matrix and camera projection matrix that will be used within the function to render the current level of recursion, as well as an integer, $recursionLevel$, that represents the current recursion depth.

4.1 Recursive Portals

Before walking through each part of the function itself, it's important to understand the basics of how recursive portal rendering works. Recursive portal rendering arises when a portal can be seen from the destination of another portal (see Figure 8). Each level of recursion renders the contents that will be displayed within the portal frame of the previous level. So the base recursion level of zero renders the scene from the main camera's point of view, a recursion level of one renders the scene from the destination of each portal visible from that main camera, a recursion level of two renders the scene from the destination of each portal visible from each of the level one destinations, and so on. The challenging part is, how can the data that has been rendered in previous levels be preserved when recursing into a portal's destination and rendering again from that perspective? This is where the stencil buffer comes in. Since each subsequent level of recursion will always be contained within the frame of a portal in the current level, we can increment the stencil buffer within the portal's frame prior to recursing into that portal's destination point of view. Then, rendering from that perspective, we can use the stencil buffer to ensure that we only render within the portal's frame, defined for us by the previous level. Since the stencil buffer starts from zero, we associate the number stored within the stencil buffer with the current level of recursion (see Figure 3),

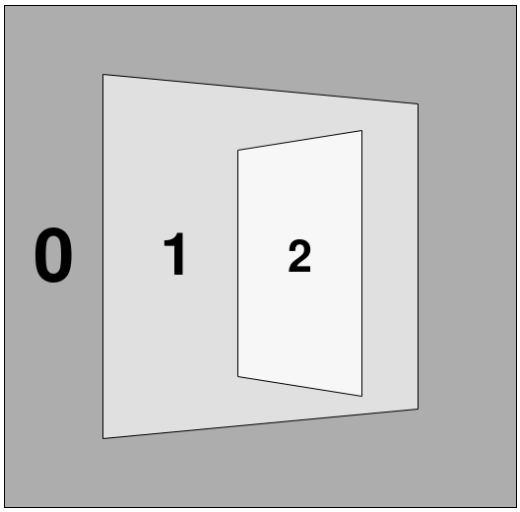


Figure 3: *Values in in stencil buffer at each level of recursion*

Algorithm 1: render()

Input: Camera's world transform matrix, camera's projection matrix and current $recursionLevel$

```
/* Render all of the portals into the depth buffer (only in areas where stencil value ==
   recursionLevel)                                                                      */
```
Enable stencil test and depth test
Disable writing to color and stencil buffers
Enable writing to depth buffer
Set depth function to succeed for values less than the current value
Set stencil function to pass when the stencil value equals $recursionLevel$
Draw portals

```
/* Render rest of scene normally (only in areas where stencil value == recursionLevel)  */
```
Enable writing to color buffer
Draw scene geometry
if $recursionLevel$ == *maxRecursionLevel* **then**
    return
foreach *portal in portals* **do**
    if *!(cameraFrustum contains portal)* **then**
        continue

    ```
    /* Increment stencil buffer values within the visible portal frame                  */
    ```
    Disable writing to color and depth buffers
    Set depth function to only pass for values equal to currently stored depth value
    Enable writing to stencil buffer
    Set stencil function to pass in areas where the stencil value equals $recursionLevel$
    Set stencil operation to increment when both depth and stencil tests pass
    Draw *portal*

    ```
    /* Clear depth buffer within the portal frame (where stencil buffer was just
       incremented)                                                                     */
    ```
    Disable writing to stencil buffer
    Enable writing to depth buffer
    Set stencil function to pass when the stencil value equals $recursionLevel + 1$
    Draw fullscreen quad

    ```
    /* Recurse from portal destination point of view                                    */
    ```
    Call render() with portal destination transform

    ```
    /* Cleanup the modified stencil values by decrementing them within the portal frame */
    ```
    Disable writing to color and depth buffers
    Disable depth test
    Enable stencil test and writing to stencil buffer
    Set stencil function to pass in areas not equal to $recursionLevel + 1$
    Set stencil operation to decrement when stencil test fails

only drawing where the two are equal. This allows for us to recurse up to a maximum depth of 255 (assuming an 8-bit stencil buffer), which is more than enough in our case.

4.2 Render function

Now for the complete rendering function. The first step is to render each of the portals in the scene to the depth buffer, making sure to enable the stencil test and ensure that any pixels where the stencil value does not equal the current recursion level are not drawn to. Next, we render the scene to both the depth and color buffers, still making use of the same stencil test.

Now, we loop over each of the portals. For each portal, we start by checking if the portal's world bounding box intersects with the frustum of the camera perspective we are currently rendering from. If not, the portal is not visible and we can skip it and continue onto the next one. Otherwise, we start by first incrementing the values within the stencil buffer everywhere within the portal's frame. This can be done by drawing the portal with the stencil operation set to only increment the value when both the depth and stencil function pass and the depth function set to only pass for fragments with a depth value equal to the value currently stored (which corresponds to the depth values for the portals we rendered at the start of the function). After that, we need to clear the depth values within the portal frame (which are currently set to the depth of the portal frame itself) in preparation for our recursive render into that portal's destination viewpoint. Although there is no direct OpenGL method for selectively clearing the depth buffer (other than through the use of the scissor test, which only works for rectangular areas) we can accomplish this by drawing a full-screen quad that has a depth value of 1.0, using the stencil test to ensure it only draws to areas within the portal frame. Next, we recurse into the render function again, adding 1 to the recursion level and calculating new camera world and projection matrices that will be used to render from the portal's destination. Once that recursive call has returned, we need to cleanup the stencil values we incremented previously so the next portal in the loop won't overwrite what we just rendered within the current portal's frame. This can be done by setting the stencil function to fail anywhere with a stencil value equal to $recursionLevel + 1$, setting the stencil operation to decrement on failure of the stencil test, and then drawing this portal one last time.

4.3 Run-time and Optimizations

The naive run-time of this rendering method is $O(n^r)$ where n is the number of portals and r is the maximum recursion depth. Therefore, it's obviously extremely important to avoid rendering unnecessary portals. My method implements basic frustum culling to avoid rendering portals that are not within the camera's view frustum. This could be further improved by shrinking the frustum so that its fit around the bounds of the portal frame when rendering from a portal's destination. Even with this improvement, however, frustum culling does not account for the case where portals are within the view frustum but occluded by some other object. [Lowe and Datta 2005] handles this case using the *GL_NV_occlusion_query* extension to check whether or not any pixels were actually drawn for a portal, allowing portals that are occluded to be detected and culled.

5 Oblique View Frustum

One problem that hasn't yet been addressed is the issue where an object is between the destination portal frame and the destination camera perspective. This is illustrated in Figure 4, where a viewer is looking through portal A, seeing what's on the other side of portal B. The purple box behind portal B is visible from the destination camera perspective and therefore would incorrectly show up when we render from that perspective, when in reality anything behind portal B should be cut off.

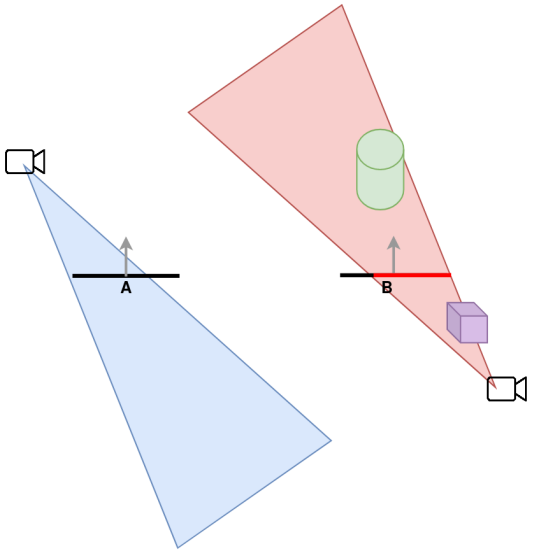


Figure 4: *Example where a purple cube blocks the destination view frustum (drawn in red). The adjusted near plane, aligned to the plane of portal B is represented by the red line. Anything behind the red line should not be drawn.*

There are multiple ways that this problem can be addressed. [Lowe and Datta 2005] made use of dual near and far depth buffers, where any fragments closer than those defined in the near buffer would be culled while the far depth buffer acted as the normal depth buffer. This defined a sort of depth range that could be used to prevent this problem by setting the near buffer to contain the depth of the portal surface. This is also what allows for portals to be of non-planar shapes in their method. However, as OpenGL does not support the use of two depth buffers by default, this requires some clever use of shadow map textures and careful management of the near and far depth values in order to use this approach. As the portals in this project are strictly planar, it seemed unnecessary for me to implement that approach.

For planar portals, this problem can also be solved using an oblique view frustum. With a method outlined in [Lengyel 2005], we can generate an oblique view frustum whose near clip plane is aligned with the plane of the portal. Therefore, everything that is behind the plane of the portal will be clipped. This is illustrated in Figure 4, where the red line aligned with portal B represents the new near plane of the view frustum.

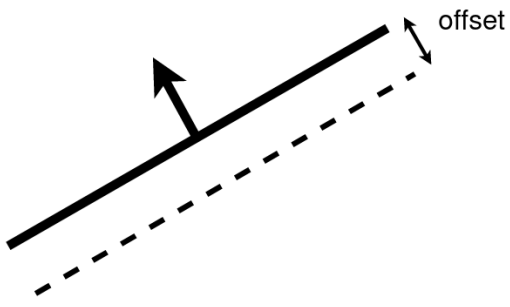


Figure 5: *Near plane offset from portal frame*

There are some problems that arise when using the oblique view frustum method, however. The first issue stems from the fact that the near clip plane of the oblique frustum is exactly aligned with the plane of the portal, resulting in z-fighting with geometry that is placed very close to the portal frame. This is especially an issue since the near clip distance of the camera needs to be set very low for reasons that will be explained in the teleportation section, further reducing the precision of the depth buffer. I worked around this issue by offsetting the near clip plane of the oblique view frustum so that while it's still parallel to the portal plane, it's placed some $offset$ distance behind it (see Figure 5).

The other issue is that when the camera is very close to the aligned near clip plane of the oblique view frustum, depth precision again causes flickering/z-fighting. In order to mitigate this problem, the $offset$ value must be able to dynamically scale down as the distance between the camera and the offset near plane of the oblique frustum gets to be very small. The $offset$ value cannot be reduced indefinitely, however, as it cannot extend past the original portal frame. Therefore, my solution was to switch back to using the normal, non-oblique view frustum whenever the scaled $offset$ value is less than some $cutoff$ value.

The tradeoff for using an offset is that one must be cautious not to put any geometry too close to the portal, as if it is within the offset distance, it won't be clipped by the oblique frustum, and will show up in the portal render.

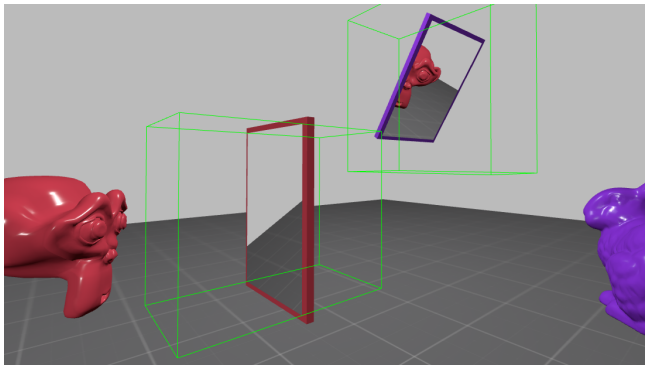


Figure 6: *Portal colliders used for teleportation*

6 Teleportation

The method that I use to implement portal teleportation for the main viewer camera is straightforward and seems to work well as far as I can tell. Each portal is given a box collider that extends slightly in front and slightly behind the surface of the portal (shown in Figure 6). To detect a valid portal teleport, we perform two checks for each portal each frame. First, we compute the dot product of the portal's normal vector and the main camera's position vector relative to the portal. We also check to see if the camera is within the portal's collider. Then we compare the computed dot product to the dot product from the previous frame. If the sign of the dot product has changed, and if the camera was within the portal's box collider either this frame or the previous frame, we know that the camera has crossed the surface of the portal and should be teleported. It's important to note that if the camera has a very high velocity and the box colliders of the portal are not sufficiently extended, it's possible that the camera could pass through the portal surface without coming in contact with the collider, resulting in no valid teleportation being detected.

Once a valid teleport has been detected, performing it is as simple as replacing the current world matrix of the main camera with the portal destination world matrix (the same one used when rendering from the destination's point of view). In order to prevent the possibility that a teleport could be registered immediately again after the initial teleport is completed (leading to the camera being continuously teleported between the two portals) the stored dot product sign and collider data for the destination portal are cleared, meaning that an additional frame will need to pass to restore them before a valid teleport from that portal will be able to occur.

One issue that can diminish the believability of the teleportation effect is the camera's near plane clipping the plane of the portal. As my implementation represents portal geometry using planes, when the camera gets very close to the portal, the near plane can clip a portion of the portal's frame. What this means is that sometimes when walking through a portal, this will cause a brief flicker where the viewer is able to see the geometry behind the portal. Reducing the near clip distance of the camera view frustum can greatly reduce how often this occurs, but reducing the distance too much results in issues with depth buffer precision, and the clipping issue cannot be completely resolved with this method alone. One potential solution might be to render the portal as a very thin box instead of a plane. However, this requires additional adjustments to be made to the teleportation system, and therefore was not something I was able to try and implement in the time frame of this project.

7 Non-Euclidean Geometry

Portals can also be used to create some interesting illusions of non-euclidean geometry. Figure 9 shows two images of the same tunnel, one from the side and one from the front. The tunnel appears to be short on the outside, yet is very long on the inside. Other examples could be a house that has fewer rooms than it appears to, a room that actually contains many rooms, a staircase that loops in an impossible way, etc. The list goes on.

The importance of lighting really stood out to me while playing around with making some of these illusions. As they usually involve sneakily stitching together two different spaces through the use of portals, the way the lighting within the scene affects those two connected spaces is often inconsistent, diminishing the believability of the illusion. I would imagine that a style of rendering that uses simpler shading would make this much easier to manage.

8 Results

Overall, the resulting portals in my final implementation are rendered accurately, supporting recursion up to a user-specified depth. The portal teleportation system works very well, despite the rare flicker as a result of the near plane clipping the plane of the portal frame.

Although the solutions I presented to the issues surrounding depth buffer precision do a good job of preventing most artifacts and z-fighting, these issues become apparent again in portals that are far away from the camera. For small scenes this is not much of an issue, however this would certainly effect the quality of portals in larger, more open scenes.

The performance is good, and runs at an acceptable frame rate in a fairly demanding scene, however, there is certainly plenty of potential for more optimization, as discussed in the rendering section.

9 Future Work

There is still many areas where the rendering method could be further improved. As mentioned in the rendering section, frustum culling would be more accurate and effective if frustums were constrained around a destination portal when performing a render from that perspective. Furthermore, culling portals that are within the frustum, but blocked from view by another object would also help

to reduce unnecessary renders.

Despite the fact that using an oblique view frustum works for planar portals, it still may be advantageous to implement the dual depth buffer used by [Lowe and Datta 2005], as that method may avoid some of the issues that arise as a result of the portal-aligned near plane.

Finally, it would be worth looking into ways to avoid the occasional flickering that occurs when the camera's near plane clips with the plane of the portal, potentially through the use of thin boxes instead of planes as portal geometry.

References

ALIAGA, D., AND LASTRA, A. 1997. Architectural walkthroughs using portal textures. 355–362.

KIRCHER, D., AND KOHLI, T. Portal Problems - Lecture 11 - CS50's Introduction to Game Development 2018. `https://www.youtube.com/watch?v=ivyseNMVt-4`.

LENGYEL, E. 2005. Oblique View Frustum Depth Projection and Clipping.

LOWE, N., AND DATTA, A. 2005. A New Technique for Rendering Complex Portals. *Computer Graphics*.

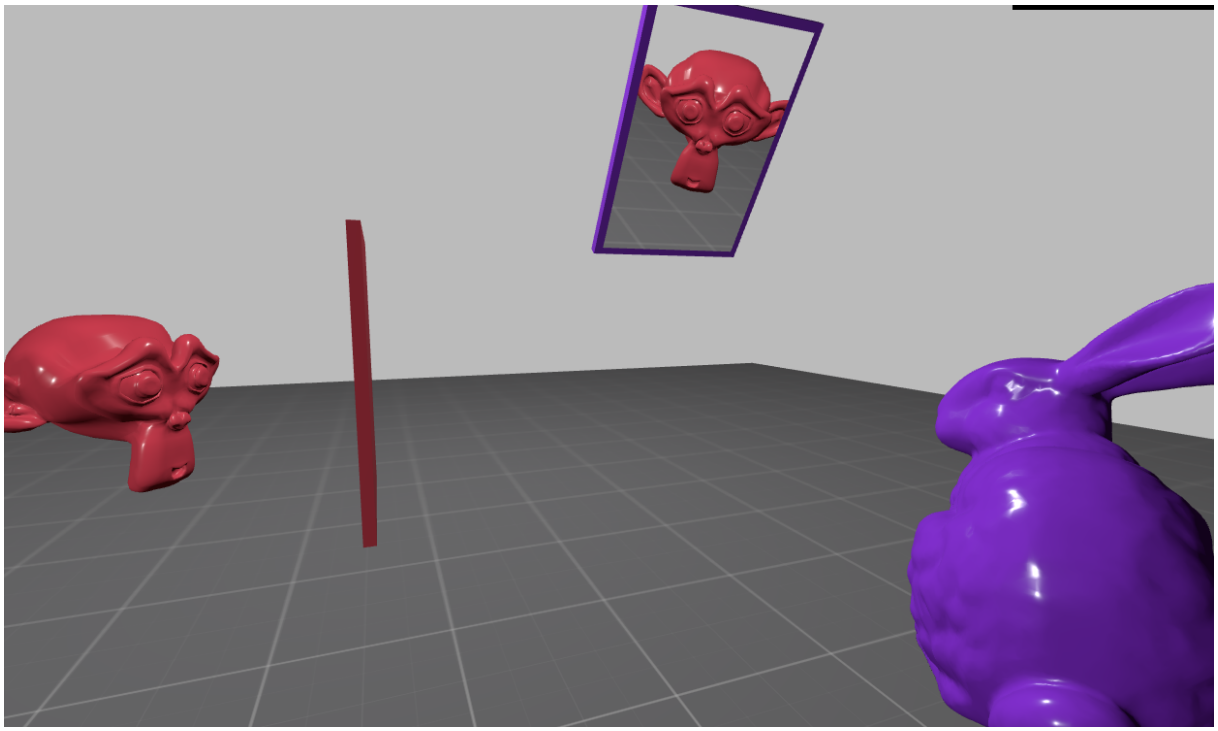


Figure 7: *A scene containing one pair of portals*

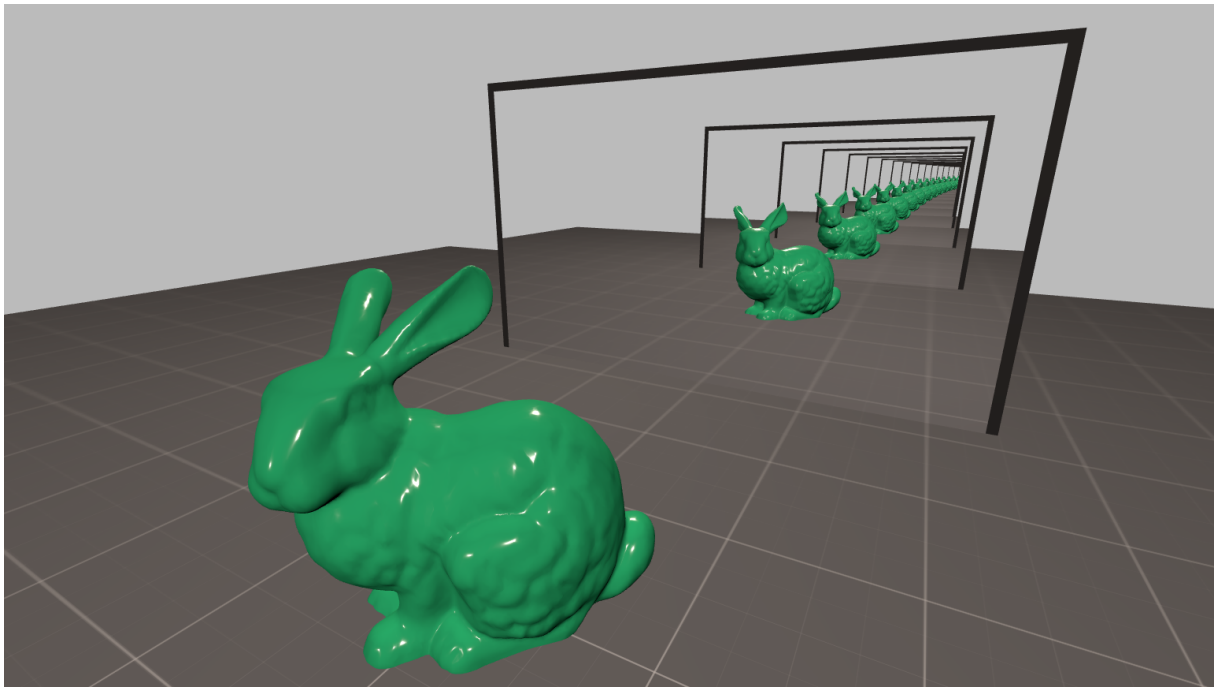
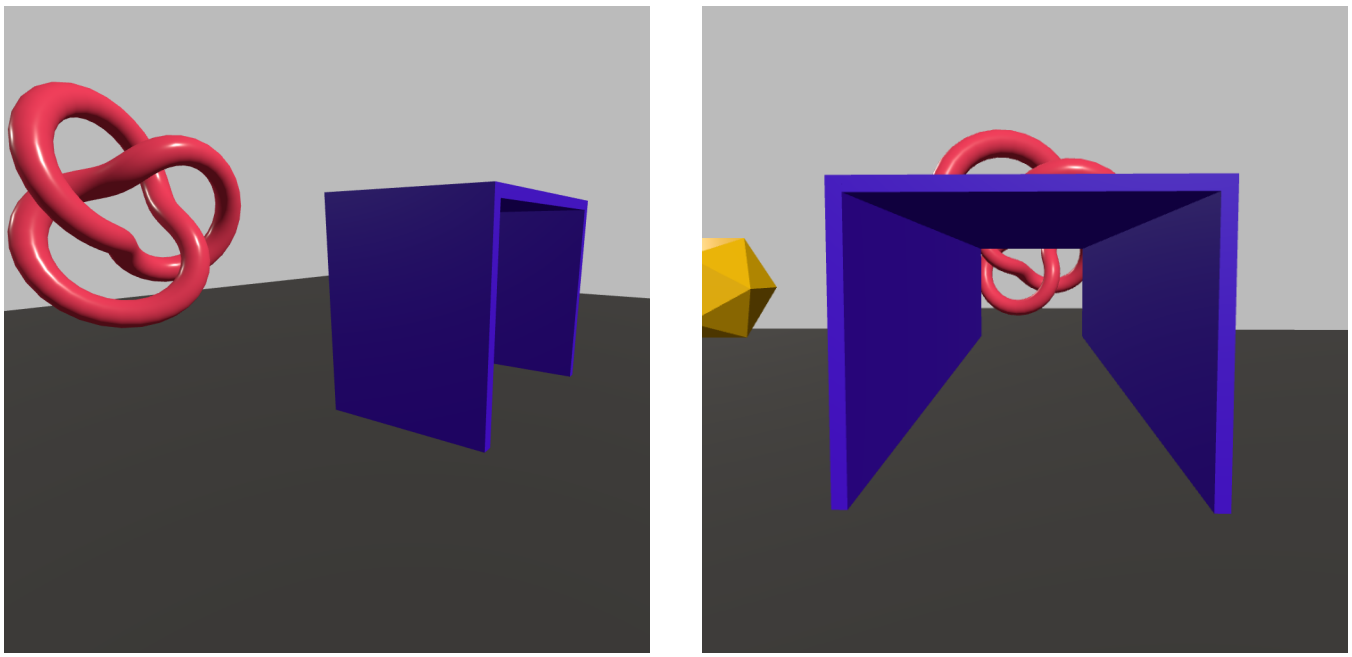


Figure 8: *Portal recursion with a recursion depth of 20*



(a) *Side view of tunnel*

(b) *View from the front of the tunnel*

Figure 9: *A tunnel that's longer on the inside*