Graphics Programming Final Assignment
Portals

By Mikkel Kaj Andersen

December 2020

IT University of Copenhagen
Course Code: KGGRPRG1KU

Contents

1 Introduction

For this project, I have decided to work with the portal mechanic from the game "Portal" (2007) and its sequel "Portal 2" (2011) by Valve Corporation[1]. In the game, the player can place portals, which can be used to teleport in space from one point to another. *This project's* primary purpose is to render what the user sees through the portal before the teleportation occurs, and therefore collision detection or teleportation will not be the primary goal, albeit I do implement a naïve approach for this. Portal views is an interesting problem, as it is a simple idea that turns out to be quite tricky. To render a perfect view, portals have to handle portals in view of portals, "banana errors," and object handling. In this report, the focus is on how the portals should be implemented *theoretically*, using two different approaches, followed by a discussion of their implementation. The first approach uses a texture as the view of a portal, and the other does a real-time rendering of the scene in question by segmenting the screen into different views. I will then discuss their implementation in `OpenGL` using an example scene and compare them by visual quality and performance. The following list of goals was made to keep track of the deadline and compare the two approaches' implementation.

1.1 Goals

1. Have an interactive application, where a player camera moves around freely. ✓
2. The player can place portals by left-clicking. They should be perpendicular and in front of the player camera. ✓
3. Using an FBO as a texture for the portals:
   - The portals should display the scene from the player's camera's point of view. ✓
   - The portals display the scene as if the player camera looks through the *other* portal. ✓
   - Render portals recursively (this should happen automatically when using the FBO approach). (×)
   - If the portals are not perpendicular to the camera, they still show a correct view. ×
4. Stencilbuffer approach
   - The portals should display the scene from the player's camera's point of view. ✓
   - The portals display the scene as if the player camera looks through the *other* portal. ✓
   - Recursive rendering. ×
   - If the portals are not perpendicular to the camera, they still show a correct view. ✓
5. Teleportation from one portal to the other! ✓

2 Theory

For this project, I will define portals to be 2D rectangles in 3D space. To avoid confusion, define the portal rendered in the scene as the *origin* portal, and the portal whose perspective

[1] https://www.valvesoftware.com/en/

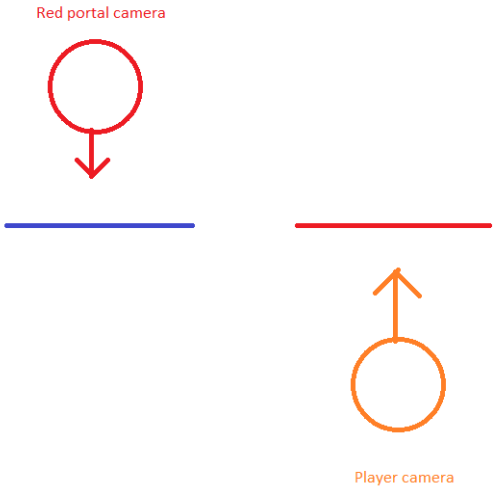


Figure 1: The virtual camera for the red portal, looking through the blue portal, based on the main camera

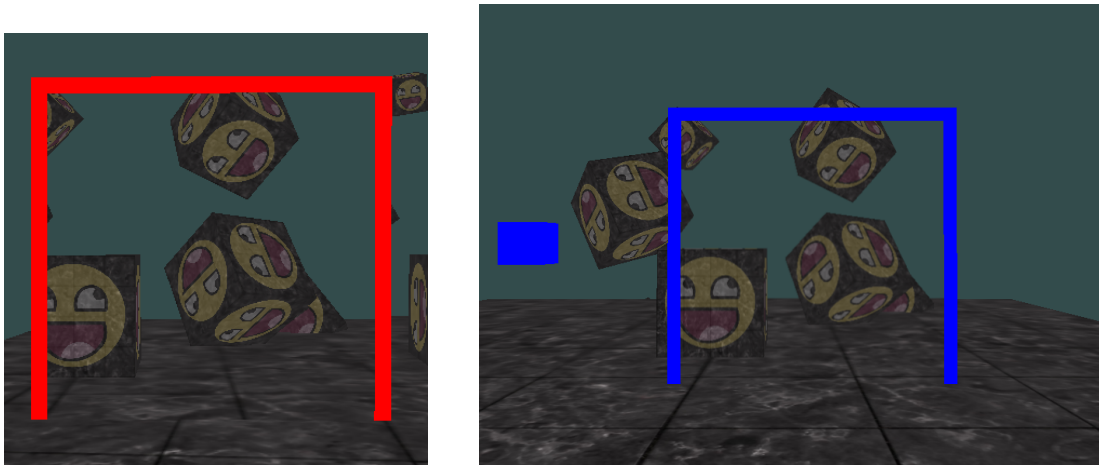
will be rendered from as the *target* portal. A portal works similar to a window, such that a camera looking at the front face of the source portal essentially travels through it and leaves through the back face of the target portal. This allows us to display the scene from wherever it is located (See Figure 1 for illustration). The camera gets flipped to the other side because this is how it was originally implemented in the games. However, this has no impact on the rest of the theory. To achieve this, I will discuss two different approaches: stencil buffers and framebuffers. The next chapter explains virtual cameras, as this is central to both approaches.

2.1 Virtual Camera

At the core of rendering a second view is what is known as a "virtual camera." The virtual camera is a way to set multiple viewpoints in a scene and render from different points of view. To construct the view, I take the main camera's relation to the first portal and apply the inverse to the red portal. Some implementations rotate the camera during this process (I do), but this is completely application-specific. To compute the virtual camera view from the source portal to the target portal, I do a series of transformations. Remember that the view receives coordinates in world space and turns them into coordinates in view space (viewpoint of the camera); I start by calculating the coordinates in relation to the target portal, using the inverse of its model matrix. I then rotate the coordinates 180 degrees around the y axis to "reverse the view." I then see where this lands us in relation to the *origin* portal. This lands us in world coordinates, so I multiply with the view matrix of the main camera to move into *view* coordinates. I do not have to generate a new view matrix, as I have essentially moved all of the coordinates that were viewable from the *target* portal to be in front of the main camera. This is implemented in `Portal::calculateView`. I also use the virtual camera for our teleportation feature, where I cut out the translation coordinates of the inverse virtual camera view matrix and apply these to our main camera.

2.2 FBO Approach

Once two portals have been placed in the game world, this approach uses a virtual camera to generate a texture for the respective portals. This is achieved by using framebuffer objects. This allows us to render the scene to a buffer instead of the default main window. To



(a) Final portal

(b) Virtual camera for the red portal, looks through the blue portal.

Figure 2: Portal pipeline

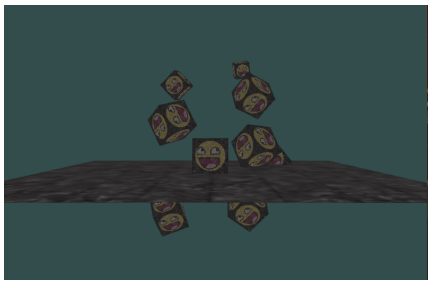
generate the texture, use the `calculateView` function to define the portal view. Then, generate a framebuffer object, and bind a texture object to it. Additionally, I also need the framebuffer's depth values, which is solved by attaching a render buffer. Finally, bind the rendering of the scene to the newly created framebuffer and render the scene. Then, use this texture for the portal, and render the scene normally. To see an example of this, consider Figure 2. Recall that the red portal's view is computed by calculating the view through the blue portal. As can be seen, the part visible in the blue portal is exactly the red portal's texture.

2.2.1 Sampling texture coordinates

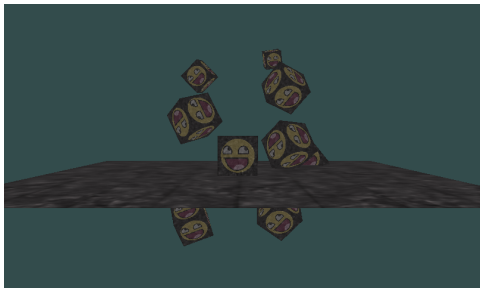
If we took the texture from Figure 2b. We would obtain the entire scene from a different viewpoint, while in fact, we were interested in only obtaining the scene visible through the blue portal. To find the correct coordinates, recall that computing the screen space coordinates projects the 3D objects into a 2D scene, and simply finding the screen coordinates (in the 0-1 range) of the portal will allow us to sample the correct texture coordinates from Figure 2b. To do this, I multiply the projection matrix onto our coordinates and perform perspective division with w to get normalized device coordinates in the range of $[-1, 1]$. Finally, I map the coordinates to the range of $[0, 1]$ and use them as texture coordinates. This allows us to sample the virtual camera's texture, where only the portion visible through the opposite portal is used. It is implemented in `portal.vert::main()`.

2.2.2 Recursion

To have recursive portals, one has to use the painter's algorithm, meaning drawing the most in-depth portals first, as subsequent portals' textures will depend on them. This will require a framebuffer for every depth. If drawing the portals with a depth of one first, then the portals viewable from the source portal will not have a texture that can be applied. It is tempting to think that using the framebuffer approach allows recursive drawings for free by applying the current texture to the recursive portal, but this will not give the correct view. If the recursive portal is drawn in the same framebuffer as the source portal, then the texture cannot be applied (because this is currently being generated).



(a) Scene rendered using framebuffer



(b) Scene rendered using stencilbuffer

Figure 3: Visual comparison, looking through a portal (Portal not visible on picture)

2.2.3 Pros & Cons

The drawback of using framebuffers for every texture is that it quickly becomes hard to simultaneously manage many framebuffers. It is my *assumption* that this process may also be more expensive to compute, due to a large amount of context switching, compared to the stencil buffer approach. Especially for drawing recursive portals, I found that creating framebuffers for every frame was impossible for the hardware to handle. Therefore, they should always be precomputed. Finally, since the player will often be very close to the portal to teleport through it, the texture approach might display worse visual quality. This was not an observation that I did in my application, but a consideration that Valve had[8]. See Figure 3 for a comparison made in my application - they seem to have the same level of detail.

Finally, Valve points out that drawing recursive portals for the FBO method quickly explodes in function calls[8], however, it remains unclear why the recursion should get bigger than for the stencilbuffer approach, except, there is clearly more memory to be allocated for increasing amounts of textures.

2.3 Stencilbuffer approach

To get an understanding of how the stencil buffer works, consider the example displayed in Figure 4. Similar to a depth buffer, a stencil buffer contains a value for every fragment which is usually 8-bits. This buffer can then discard or allow fragment rendering by comparing the currently stored value in the buffer with a reference value. This is done by defining the `glStencilFunc` which takes an comparison operator, and a reference value, for comparing the value in the stencil buffer. The `glStencilOp` then defines what action is taken based on pass/fail.

As in the example, a default stencil buffer has 0 for every fragment, but this could be changed by executing an operation based on if the reference value is equal to the currently stored value, and if not, increase from 0 to 1. To obtain the figure's result, you would disable writing to the color buffer and render a square with a hole in it. For every fragment passed, you would increase its stencil buffer value by one and finally render the scene only where the stencil buffer is equal to one.

The approach I use to draw portals is not much different from the idea sketched in the figure. To render portals using the stencil buffer, I first disable the depth and color buffer and enable writing to the stencil buffer. I then render the portals in the scene and give every fragment a unique stencil identification value to refer to these later. I call this way of partitioning the screen *scene fragmentation*. For all intents and purposes, I give the first portal a stencil value of 1 and the other a stencil value of 2. I then render the portal scenes by computing the `calculateView` function and rendering only where there is 1 and 2 in

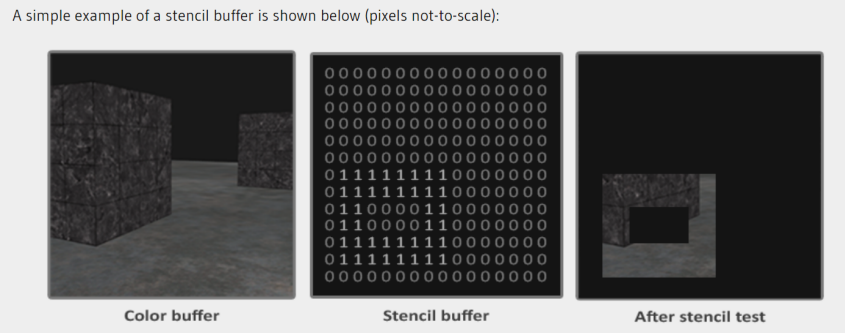


Figure 4: Stencilbuffer example [3]

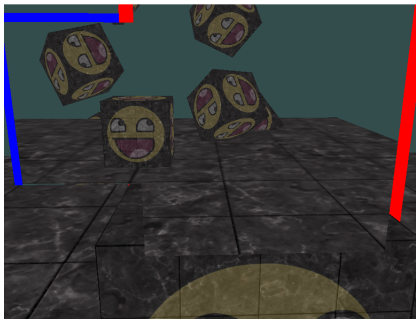


Figure 5: Wrong depth values for portal

the stencil buffer, respectively. Once this is implemented, however, the view will not be entirely correct, as objects viewable from the main camera in the scene will not occlude the portal view (see Figure 5). To fix this problem, I have to clear the depth buffer and write the portals depth values in the scene to the depth buffer, which I accomplish by disabling writes to the color buffer and disabling the stencil buffer test in general and then drawing the portal. Once this is done, the portals will be occluded by other objects.

2.3.1 Recursion

To perform recursion in this setting, one does not have to use the painter's algorithm. However, some do this[7]. This also seems like the easiest way to handle the many stencil values. Recursion starts by segmenting the scene for every recursion depth i and portals $i+1$ and $i+2$, and then renders according to which portal the fragment belongs to along with the current depth. This is easier said than done, as we have to distinguish which portal we are in, or we will encounter an occluding bug, described in Chapter 3.3.2. One way to solve this is by using the stencil buffer as a bitfield that defines which "parent" portal a fragment belongs to [6]. Another clever way [7] is to build up the recursion by first doing a complete segmentation of the scene and start rendering in a depth-first manner. The brilliant part is to clean-up the recursion as the scene is rendered, such that the stencil buffer ends up with all 0's once finished, allowing for easy rendering of the next portal. I played around with this idea, but once the stencil buffer is all 0's, then the scene is rendered everywhere, and thus I have not gotten a recursive feature working.

2.3.2 Pros & Cons

The stencil buffer approach seems like an ideal way to draw portals' view. It does not need to handle texture sampling, and the visual quality is homogeneous, compared to the texture approach, which might show worse quality based on the size of the texture. Additionally,

there is no need to hold a large amount of texture in memory, as the scene is rendered through clever use of the stencil buffer. The biggest drawback is that it is much harder to implement correctly.

3 Application

3.1 Features and credits

The application can create portals in a typical scene based on the tutorials from `www.learnopengl.com`. The ideas for computing views using FBO and Stencil was based on `https://th0mas.nl/2013/05/19/rendering-recursive-portals-with-opengl/`[7], in addition to other sources [2][8][1][6][5]. Even though teleportation was not the focus of this project, the portals also allow the player to teleport once debugging is enabled. The teleportation occurs a bit rough due to a poor collision detection mechanism.

3.2 How to set up the application and use it

To set up the application, everything that is necessary is to move the entire project folder into the assignment folder, as given in the exercise repository from class. There should be no additional setup required.

3.2.1 Documentation

The application can do the following by using these keybindings:

- (Mouse left-click): Places a portal. In order to start rendering, two portals must be placed. Continuously left-clicking will update the existing portal positions.
- (1): Runs the portal rendering using the FBO method.

  Debugging mode in this setting will allow recursive portal outline renderings, and draw the portals perpendicular to the camera.
- (2): Runs the portal rendering using the stencil buffer approach.

  Debugging mode in this setting displays the virtual cameras.
- (3) `(Must use FBO Rendering)`: Render the scene from the view of the blue portal, which is subsequently applied as a texture for the red portal.
- (4): Render the scene from the player camera.
- (P): Enable debugging mode. **This mode** is required for teleportation to happen. It also displays the location of the virtual cameras of the respective portals. When disabling the debugging mode, the cameras will stay, such that one can move to them and see their perspective.
- (O): Disable debugging mode.

3.3 Improvements

In this section, I discuss some of the current problems and work for the future, divided into the FBO and stencil implementations.

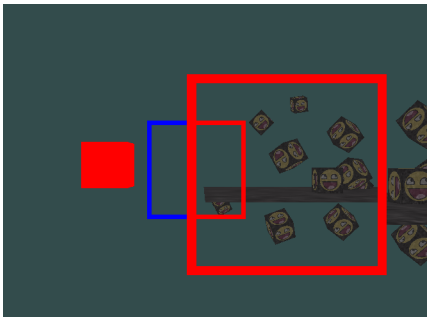


Figure 6: Red portal changes color in the blue portal

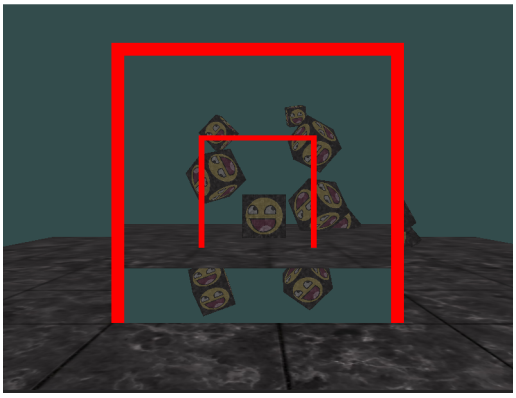


Figure 7: No texture applied to the portal we are looking through

3.3.1 FBO Approach

Rendering portals using the FBO method is not completely fully functioning for some parts of the system. *Note that the photos taken in this section were made before the decision to rotate the view* 180°*, so in the figures, the portal views point in the same direction as the camera.*

1. **Recursive portals (`generateTextureForPortals`):** As can be seen in Figure 6, portals change color when they are being rendered recursively; in fact, they do not render the correct view at all. They render the view of the other portal. This occurs because of how we implemented rendering for the view of the *red portal*. To render the view for the red portal, we move to the red camera (as the red square seen in Figure 6), render the scene, and snip only the part visible through the blue portal. There is a problem; however, we are rendering recursively, so the blue portal already has a texture applied to it, so when we "snip" the scene from it, we are snipping what is visible for the blue portal through the red portal. To work around this issue, we disable applying textures for the portal we are looking through, as done in Figure 7. The thought was that we would be able to apply the texture in the process of being made to the recursive portal; however, this texture is not ready to be applied at the time of rendering, so the portal becomes transparent. For testing purposes, if we use the other portal's texture, the recursive portal, in fact, does render the texture. For this to work, I would assume one should be *very* careful with handling the generated textures. This is additionally complicated due to the overhead of handling different framebuffer objects. It was attempted to generate new framebuffers on the fly and use them as the in-screen portal's textures. However, this causes way too much overhead for OpenGL to handle. Additionally, simply using the same texture will not give a correct view.

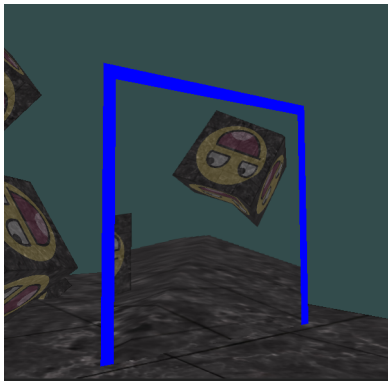


Figure 8: Warping effect when viewing portals at an angle

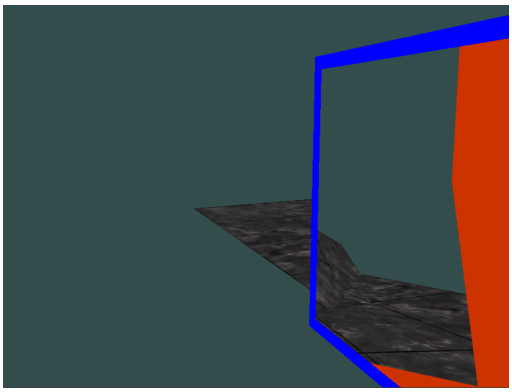


Figure 9: UV coordinates being sampled outside of texture

2. **Sampling texture coordinate (`portal.vert`):** Our approach to texture sampling assumes that the portal and the camera is perpendicular, which is a consequence of discarding the depth values for the screen coordinates. This means that when we sample the coordinates, they will be skewed and not entirely suitable for applying to a square (see example in Figure 8). This effect occurs as we cut out a trapezoid-like object from the texture and attempt to force the trapezoid shape onto a square object (the portal), which does not make sense. To confirm that this is the case, I implemented a `DrawPerpendicular` version of the portals that always places them perfectly aligned with the camera, and the views now appear perfectly. I believe Figure 8 is related to the next error (see Figure 9): In this figure, we see that some part of the portal has a red color, which means that the u, v coordinates are sampled outside of the range $[0, 1]$. This happens once you get close enough to the portal that a section of it is outside of view. This is yet another consequence of our flawed texture sampling, however, it is unclear to me exactly what causes it. I believe it may be because the projection matrix makes the u, v coordinates go out of alignment.

3.3.2 Stencil Buffer Approach

With no further adjustments, the stencil buffer approach does not incur the problems described earlier using the FBO approach. I.e., the largest bug (texture sampling, seen in Figure 9) was solved by a simple change of implementation and using the same logic for computing views, etc. I expected it would be pretty easy to start drawing recursive portals, but this indeed also turned out to be quite frustrating, as one has to be constantly aware of the depth and stencil state. Because of time restrictions, I decided this would not be in

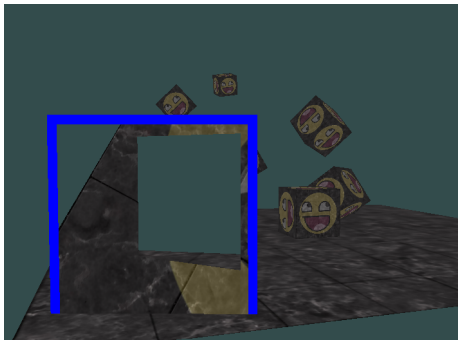
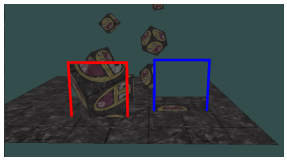
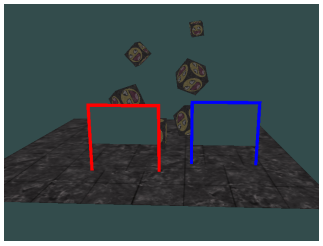


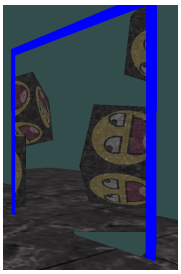
Figure 10: Stencil value bug



(a) Banana juice occuring



(b) Banana juice fixed



(c) Error in view

Figure 11: Banana Juice comparison

scope.

1. **Portal overlapping** (`recursiveStencil`): Because our stencil buffer approach simply writes a unique identification value and renders to these in the same pass, there is no distinction on if the portals are overlapping. This causes the fragment seen in Figure 10. One workaround would be to define a stencil operation that passes or fails based on the stencil value and the depth value, such that occluding portal views will not overlap.

3.3.3 Banana Juice

Fellow for both implementations, they suffer from banana juice, which is a name coined by Valve, for when the virtual cameras are intersected in their view of the portal. To fix it, one has to define the near clipping plane of the projection matrix as the distance to the portal plane, such that anything behind the portal is clipped. In other words, the view frustum of the camera sees only things in front of the portal. This was attempted to be implemented using "Oblique Near-Plane Clipping"[4] as suggested by [7], but no satisfying results were obtained. Additionally, it may be the case that using this projection technique would allow our texture coordinate approach to work without further changes.

A simple fix was to define the near distance solely based on the distance between the camera and the portal. This works pretty good and can be seen in Figure 11 for comparison. However, this is a crude approximation of the near plane that does not account for the angle. Therefore, certain artefacts are occurring, as seen in the figure.

3.4 Discussion of stencil buffer vs FBO

Based on the current work with the stencil buffer and the frame buffer implementations, I would needlessly say that the stencil buffer is a much better way to implement the view of

portals. We saw that, in Figure 4, the visual quality was homogeneous for both approaches, but it is a common concern [8][7] that the texture quality will be worse, and use more memory. It may be easier to implement recursion for the framebuffer approach, but even if done so, it will probably feel incomplete, compared to a crisp stencil buffer rendering. Additionally, many of the drawbacks that occured for the frame buffer implementation was not suffered for the stencilbuffer, and this makes it a lot easier to use and implement. There was only a single error occured for the stencil buffer implementation, and I believe it is relatively straight forward to fix.

3.5 Conclusion

This project sought to implement the rendering of portal views using two different approaches. In conclusion, the project was successful on many parameters and fell short on some, like texture sampling and correct scene segmentation for the stencil buffer. However, this report included an extensive overview of some of the pitfalls that occur in portal rendering. The camera fails when viewing frame buffer portals at an angle, and clearly, this is a significant flaw, but it would also be tough to correct and would be getting out of the course's scope. In conclusion, the results are very satisfying, and it has been great to compare the two approaches, for which I concluded that a stencil buffer approach is superior. The knowledge gained on framebuffers and stencil buffers is beneficial, and as a student, I find that this project has the potential to be used as an exercise for future course iterations.

References

[1] *Coding Adventure: Portals.* `https://www.youtube.com/watch?v=cWpFZbjtSQg`. Accessed: 2020-12-15.

[2] *How were the portals in Portal created? — Bitwise.* `https://www.youtube.com/watch?v=_SmPR5mvH7w`. Accessed: 2020-12-15.

[3] *LearnOpenGL stencilbuffer tutorial.* `https://learnopengl.com/Advanced-OpenGL/Stencil-testing`, note = Accessed: 2020-12-15.

[4] *Modifying the Projection Matrix to Perform Oblique Near-Plane Clipping.* `http://www.terathon.com/code/oblique.html`. Accessed: 2020-12-15.

[5] *OpenGL Programming/Mini-Portal.* `https://en.wikibooks.org/wiki/OpenGL_Programming/Mini-Portal`, note = Accessed: 2020-12-15.

[6] *Rendering "Portal".* `https://torinmr.github.io/cs148/`. Accessed: 2020-12-15.

[7] *Rendering recursive portals with OpenGL.* `https://th0mas.nl/2013/05/19/rendering-recursive-portals-with-opengl/`. Accessed: 2020-12-15.

[8] *Valve developers discuss Portal problems - CS50's Intro to Game Development.* `https://www.youtube.com/watch?v=riijspB9DIQ`. Accessed: 2020-12-15.