MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Portals in Godot Engine

Master's Thesis

BC. VOJTĚCH STRUHÁR

Brno, Spring 2025

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Portals in Godot Engine

Master's Thesis

BC. VOJTĚCH STRUHÁR

Advisor: Mgr. Jiří Chmelík, PhD.

Department of Visual Computing

Brno, Spring 2025

FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

While writing the text of this thesis, I have used Grammarly AI for grammar correction and ChatGPT to help me with LaTeX formatting. I declare that I used these tools in accordance with the principles of academic integrity. I checked the content and take full responsibility for it.

Bc. Vojtěch Struhár

Advisor: Mgr. Jiří Chmelík, PhD.

Acknowledgements

Once again, I want to express my gratitude to my mentor, Mgr. Jiří Chmelík, Ph.D., for his advice and support. His dedication to the game development field is truly inspiring.

I'm also extending my thanks to my colleague Ing. Marek Zouhar. Our brainstorming sessions during the portal implementation phase have been very valuable.

Next up, I want to thank my family for their continued support and for setting me up for success in life.

Last but not least, I want to thank my fiancée for always believing in me.

Abstract

In this thesis, I research and develop a 3D portal implementation for the Godot Engine, which is available as a plugin. The plugin enables simple creation of seamless portals right within the engine editor. I also discuss various engine limitations I encountered while implementing the portal functionality.

Keywords

Portals, Godot Engine, Plugin, Game development, Video games

Contents

Introduction **1**

1 Portal Expectations **2**
1.1 Visuals . . . 3
1.2 Teleportation . . . 4
1.2.1 Partial Teleportation . . . 4
1.2.2 Non-vertical portals . . . 5
1.2.3 Moving Portals . . . 5
1.2.4 Different Size Portals . . . 6

2 Related Work **7**
2.1 Existing Portal Plugins . . . 7
2.2 Different Portal Types . . . 9
2.2.1 Non-visual Portals . . . 9
2.2.2 Seamless Portals . . . 9
2.3 Common Problems . . . 12
2.3.1 Ray Cast Forwarding . . . 12
2.3.2 Recursive Portal Rendering . . . 13
2.3.3 Smooth Teleportation . . . 15
2.3.4 Portals on Walls . . . 16
2.3.5 Lighting and Shadows . . . 17
2.3.6 Portal Plane Clipping . . . 18

3 Plugin Design **21**
3.1 Choice of Game Engine . . . 21
3.2 Scripting Language . . . 22
3.3 Extending the Godot Editor . . . 22
3.4 Design Philosophy . . . 23

4 Plugin Implementation **24**
4.1 General Principles . . . 24
4.1.1 Exit Portals . . . 25
4.2 Rendering . . . 26
4.2.1 Portal Mesh . . . 28
4.2.2 Tone Mapping Adjustment . . . 30
4.2.3 Camera Clipping . . . 31

4.3 Teleportation 34
4.3.1 Teleport Root 34
4.3.2 Smooth Teleportation 35
4.4 Public API 37
4.4.1 Signals 37
4.4.2 Methods 38
4.4.3 Callbacks 39
4.5 Configuration Options 39
4.5.1 General 40
4.5.2 Rendering 42
4.5.3 Teleport 44
4.5.4 Advanced 46
4.6 Editor Integrations 47
4.6.1 Inspector 47
4.6.2 Configuration Warnings 48
4.6.3 Documentation 49
4.6.4 Gizmos 49
4.6.5 Project Settings 50

5 Conclusion **51**
5.1 Future Work 51

Bibliography **53**

A Installation Instructions **55**

B Example Workflow **57**

C Screenshots **61**

List of Figures

1.1 Portals in the Avengers™ 3
1.2 Velocity Transformation 5

2.1 Portal Types in Video Games 10
2.2 Recursive portal rendering in Portal™ 2. 14
2.3 Multiple Portal Renders 16
2.4 Convenient Lighting in Portal Games 19

3.1 Godot releases on Steam 21

4.1 Portal leading into itself 26
4.2 Double Tone Mapping 31
4.3 Near Clipping 33
4.4 Smooth Teleportation Showcase 36
4.5 Portal script inspector 41
4.6 Dynamic Inspector Fields 48
4.7 Portal Gizmos 50

A.1 Plugin Installation From the Godot Editor 56

B.1 Antichamber – Many Paths To Nowhere, Level Setup . . . 58
B.2 Antichamber – Many Paths To Nowhere, Portal Setup . . 59

C.1 Shortcut Corridor 62
C.2 Impossible Cube 63
C.3 Three Room Corner 64
C.4 Column Portal 65

Introduction

Portals are a widely known feature in the video game industry, arguably made famous by the Portal™ game series. They represent both an interesting gameplay mechanic and a powerful level design tool. In this thesis, I explore the ideas and techniques used for implementing seamless portals in existing video games. I also developed a new seamless portal implementation for the Godot Engine, documenting the engine's abilities and limitations in the process.

Thesis Structure

In Chapter 1, I cover how a portal should work. The entire chapter is largely a thought experiment, supported by movies or fantasy books. I purposely don't draw inspiration from video games during this phase.

Chapter 2 introduces related work from other video games. I also discuss various implementation difficulties that need to be addressed when recreating the effects covered in the previous chapter.

Chapter 3 is where I outline the design of my plugin. Major technical choices and the implementation philosophy are described here.

In Chapter 4, I finally describe the portal plugin implementation I developed for this thesis. I also cover how the plugin interacts with the Godot Engine editor.

Chapter 5 serves as a conclusion. In it, I reflect on the project's implementation process and potential future work.

1 Portal Expectations

In this chapter, I will establish theoretical requirements and expectations for how portals should work. These expectations are a collection of ideas from many people about how portals should look and behave. I also look towards movies, books, and other fantasy works for inspiration.

Let's start with a definition of the word "portal" itself. The Oxford Dictionary of English defines it as follows:

> **portal**
> *1.* a doorway, gate, or other entrance, especially a large and imposing one.
> - (in fantasy and science fiction) a means of entering another world or dimension, or of travelling instantaneously from one place or time to another, often imagined as a door or window.

Indeed, portals bend space in impossible ways, allowing essentially instant travel from one place to another. One way to think about them is as if they represented a window into a parallel world. From the traveler's point of view, the two destinations connected by a portal look like they must occupy the same space – but they do not. The worlds beyond each portal do not interact at all; their only intersection is defined as the portal surface.

Each portal that can be traveled through is, in fact, made of two separate portals, each in its respective destination. Portal A leads into portal B, and vice versa. These two portals are linked together, each being the exit portal of the other. The front side of a portal A is the back side of its exit portal B. By looking at a portal from the front, we expect to see what's in front of its exit portal. By walking into a portal's front, we should walk out of the front side of the exit portal. This hints at the transformation needed to implement portals in video games – more on that in Section 4.1.

1.1 Visuals

The most important thing about portal visuals is that they look completely normal on their own. It is only the contrast with the portal surroundings that makes them look supernatural. If we encounter a well-placed portal that blends into its surroundings, we might not even notice it is there. Often, movies or games accentuate their magical properties of portals by giving them a glowing outline – or a similar effect. One such example can be seen in Figure 1.1. This touch is purely the result of the author's artistic expression.

A simple window covered with a carefully polished glass pane serves as a good real-world example of a portal. The view of the space on the other side (indoors or outdoors) adjusts smoothly to our view angle, and light comes through naturally. There should be no way to know if the window frame is filled with glass or if it is actually a portal to a potentially distant place.

Figure 1.1: Doctor Strange is coming through a portal in Avengers™: Infinity War. He steps out of a house in New York right into Central Park.

1.2 Teleportation

As hinted by the dictionary definition of a portal, entering one should be like walking through a door. We expect to be seamlessly transported from one place to another without any perceivable disturbances. When traveling through a portal, momentum is conserved from our point of view, as depicted in Figure 1.2. Therefore, running into a portal should result in running out of the exit portal. Throwing an object into a portal feels completely natural – the object's trajectory still makes sense from our point of view, even after being teleported.

It is interesting to note that if portals were real, objects would get teleported gradually rather than instantly. That implies that they exist in two places at the same time. It is needlessly complicated to model this behavior in video games. Instead, objects get instantly teleported when their origin point crosses the portal surface. The appearance of a smooth teleport is then achieved through clever workarounds, discussed in Section 2.3.3.

1.2.1 Partial Teleportation

Imagine going through a portal with your arm extended sideways over the edge of the portal. I would argue that the correct consequence is that only your body gets teleported, resulting in the portal edge cleanly cutting the extended arm off.[1] This exact effect is shown off in Avengers™: Infinity War, when a monster tries to reach through a portal to grab a wizard on the other side. The wizard quickly closes the portal, which results in the monster's arm being cut off cleanly.

This unpleasant side-effect is not trivial to simulate in video games and does not add much gameplay value. For this reason, most of the portals we see in games have a collider around the portal, similar to a door frame. This portal frame then completely prevents partial teleportation.

[1]Splinching in Harry Potter by J.K. Rowling is another interpretation of the partial teleportation side effect. During teleportation, part of the wizard's body might not travel with him, resulting in injuries.

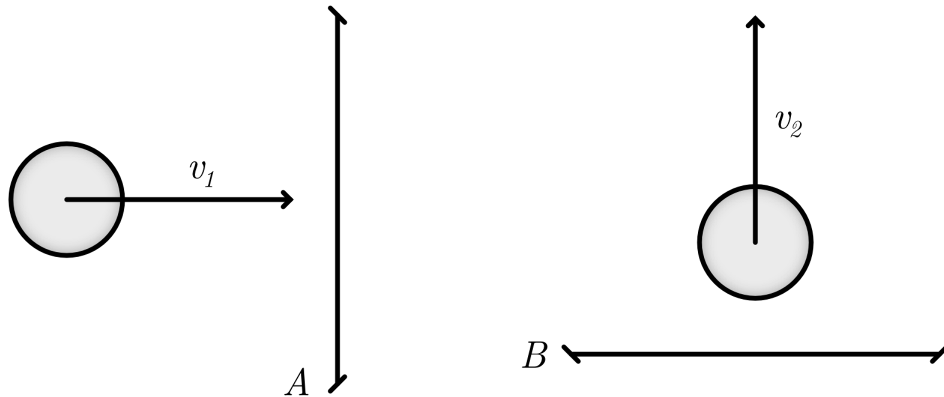


Figure 1.2: When an object enters the portal A, only the direction of its velocity changes when it comes out of the portal B. The magnitudes of v_1 and v_2 are the same.

1.2.2 Non-vertical portals

An upright, perfectly vertical portal pair looks and acts just like a doorway. However, this impression falls apart if one of the portals is rotated around a different axis other than just the up-axis, because suddenly, the room on the other side appears tilted.

The most extreme example is if one portal is upright, like a door, and its exit is on the ceiling. Walking into that portal would result in you falling onto your face. This behavior is obviously undesirable, as people generally like to spend their time mostly upright. Video games prevent this situation by automatically rotating the player into an upright position after he walks through a portal.

1.2.3 Moving Portals

Moving portals is a huge source of confusing situations and paradoxes. Various issues and questions arise when trying to figure out how these should work intuitively. For example:

- When a portal falls onto an object that isn't moving, is it the same as the object jumping into the portal?

- If a moving portal stops moving halfway through an object, should the part of the object that has already traveled through keep some momentum?
- Presumably, portals can pass through another portal. What would teleportation through the intersecting portals look like?

Unfortunately, there are no real-life portals that we could observe and learn from. Most portal implementations in video games simply choose to avoid this problematic area. Portals are completely static and placed by a level designer[2], or they do not move once placed by a player.[3]

1.2.4 Different Size Portals

As I have stated previously, each portal has its exit portal. It is unclear, however, how teleportation should work if these two portals are not the same size.

A realistic approach could be to interpret the size difference as a **teleport scale**. Going through a portal repeatedly would lead to the traveler shrinking to an infinitely small size. Similarly, traveling in the opposite direction would make the traveler bigger and bigger to the point where he would no longer fit into the portal frame for another round trip. Changing the player's size in this way is mostly undesirable in games.

Video game objects have an origin point, which can be used to teleport them proportionally to where they entered the portal. For example, when entering a portal at 75% of its width, the object would appear at 75% width of the exit portal. Different heights might still be an issue, especially if the player's origin is at its center of mass. If the exit portal is significantly higher than the entry, a player entering at ground level could appear in mid-air once they travel through.

Most video games keep their entry and exit portals the exact same size, eliminating all of these complexities.

[2] Antichamber
[3] Portal™ 2, Splitgate

2 Related Work

In this chapter, I am going to go over existing portal implementations in Godot, highlighting what they do well and how my project is different. I also look towards similar works in other game engines. I will then review the technical aspects of implementing portals in a video game, using existing games as a reference. Many games employ teleportation[1] in various degrees, but teleportation is only a part of what makes portals interesting. The importance of the portal mechanic to the gameplay generally guides the style of the portal implementation.

A major source of information on portal implementation techniques is the game Portal™ 2 by Valve.[2] Dave Kircher, one of the game's developers, defined portals as follows:

> *A portal is a discontinuity in 3D space where the back face of a 2D rectangle is defined as the front face of another 2D rectangle located elsewhere (but without actually moving any geometry or overlapping space in incomprehensible ways).* [1]

The concept of portals has also found other applications, aside from being an interesting gameplay element. Rendering of separate rooms has been delegated to separate machines, with the rooms being connected by a seamless portal.[2] Portal-based occlusion is a technique for fast visibility determination, however, the level design has to be suitable for this culling approach to work efficiently.[3, 4] Invisible portals have also been utilized for sound propagation in games.[5]

2.1 Existing Portal Plugins

The concept of portals has been explored by numerous developers within the Godot community, although few implementations are designed as reusable, general-purpose plugins. For this thesis, I examined several publicly available projects to gain insight into various techniques and design decisions related to portal systems.

---

[1] Different games may call teleportation differently, such as Warp (popular Minecraft server plugin), Blink (Dota 2, World of Warcraft), Recall (League of Legends) or Fast Travel (Witcher 3).

[2] `https://store.steampowered.com/app/620/Portal_2/`

A search for the term *"Portal"* in the Godot Asset Library yields only a single relevant project.[3] This implementation features an indented quad mesh as the portal surface – a technique I also explored in Section 4.2.1. The project is well documented, which proved helpful for understanding key concepts such as the portal transformations. However, the system is limited to one-sided portals, meaning they are only visible and traversable from the front. My portal plugin does not have this limitation.

Some developers opt to implement portals by modifying Godot's engine source code.[4,5] These modifications primarily address rendering limitations, such as enabling the overriding of the projection matrix.[6] This is a completely valid approach for developing a portal game – after all, Godot being open source is one of its main selling points.[7] However, modifying the engine introduces a significant usability barrier. One of the key goals of this thesis is to create a widely accessible portal plugin. Requiring users to compile a custom engine build would severely limit the potential user base of the plugin. Therefore, I opted for a solution that works with the official Godot Engine release.

Notably, none of the Godot-based implementations I reviewed are intended as standalone, user-friendly portal plugins. Instead, they tend to serve more as open-source portal demonstrations with varying degrees of reusability.

Portal systems developed for the Unity engine also served as valuable references. Namely, tutorials by Sebastian Lague[8] and Brackeys[9] utilize box meshes to represent portal surfaces. This design choice influenced early iterations of my implementation, I discuss it further in Section 4.2.1. Additionally, the GoThrough tool is another Unity-based portal system that has been the subject of multiple technical articles, which provided further practical insights.[6, 7]

[3] `https://github.com/Donitzo/godot-simple-portal-system`

[4] `https://github.com/majikayogames/portal_demo`

[5] `https://youtu.be/zT6Zlx09UQY`

[6] `https://github.com/V-Sekai/godot/tree/override_projection_4.2`

[7] The headline on Godot Engine's official website says *"Your free, open-source game engine."*. `https://www.godotengine.org`

[8] `https://github.com/SebLague/Portals`

[9] `https://github.com/Brackeys/Portal-In-Unity`

2.2 Different Portal Types

In my observation, video game portals tend to fall into several categories. These categories differ in how they approach rendering and teleportation. See Figure 2.1 for the overview of the following portal styles.

2.2.1 Non-visual Portals

Non-visual portals do not try to conceal themselves or provide an immersive experience. The rectangle of the portal does not display the portal's exit view. Instead, an eye-catching texture might be placed over the portal entry to indicate its purpose. What separates this kind of portal from a simple teleportation is that the player character has to walk into the portal to interact with it.

Perhaps the best example of a non-visual portal is how the Nether portal[10] is implemented in Minecraft, shown in Figure 2.1. Another example would be the Town portal from Diablo 2[11]. In both examples, a loading screen follows the teleportation to the next area, which further breaks the immersion.

2.2.2 Seamless Portals

Seamless portals render the view from their exit onto their surface, emulating more faithfully what a real portal might look like. Depending on a particular game design, they may be obvious or hidden from the player.

Obvious portals

Obvious portals appear more commonly in games that rely on portals for their core mechanic. Examples of such games are Portal™ 2 or Splitgate[12], where the portals are surrounded by a glowing outline, clearly indicating their location. If a game allows the player to place

[10] https://minecraft.fandom.com/wiki/Nether_portal
[11] https://diablo.fandom.com/wiki/Town_Portal
[12] https://store.steampowered.com/app/677620/Splitgate/

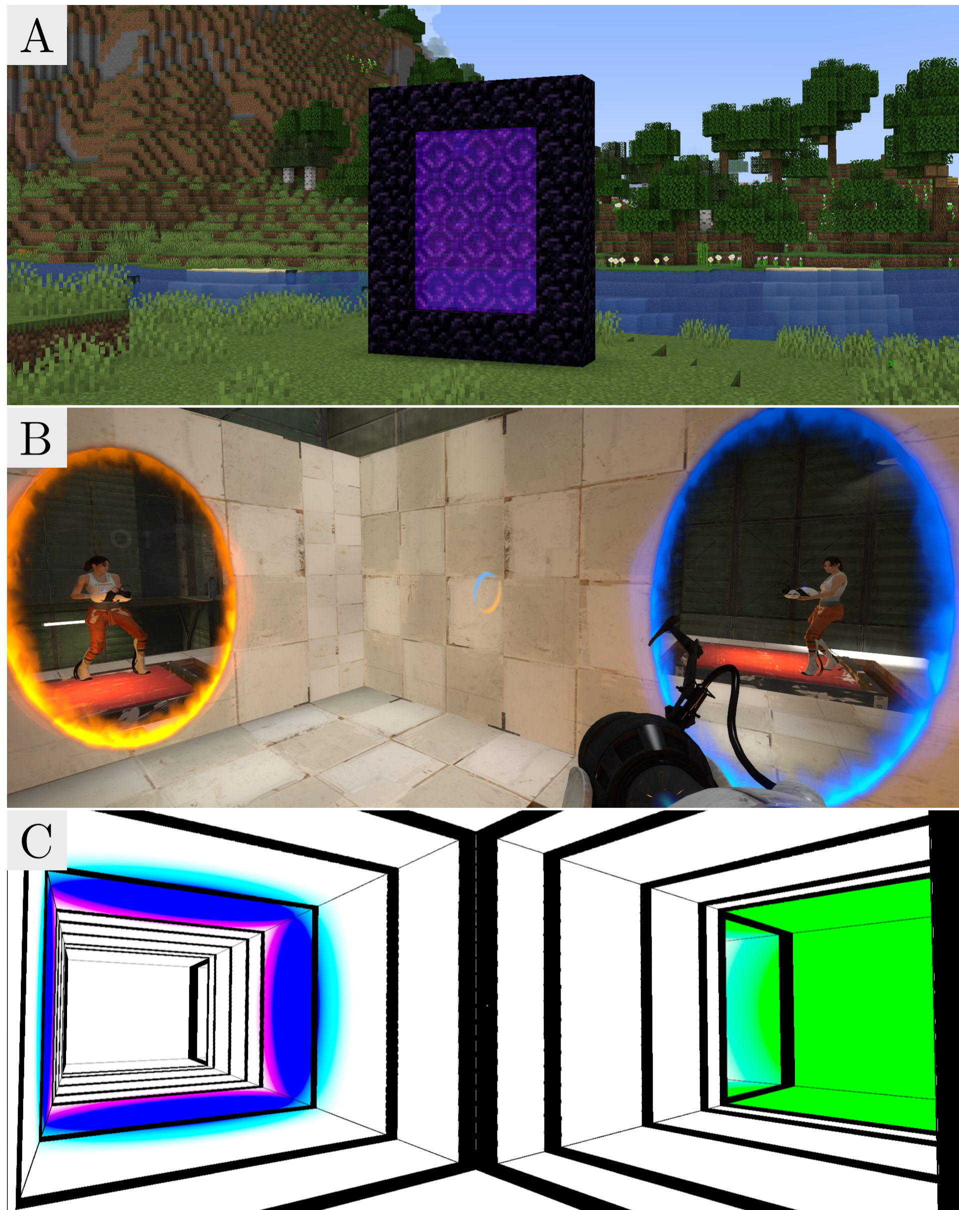


Figure 2.1: Screenshot *A* shows a non-visual Nether portal in Minecraft. Screenshot *B* is from Portal™ 2, featuring seamless and obvious portals. Screenshot *C* shows an impossible maze from Antichamber. Portals are prevalent in the maze but completely hidden from the player.

portals into the world, this is usually the style of portals the game implements. There is no point in hiding the portal from the player.

The unpredictable nature of player-placed portals makes the portal implementation very hard. Difficulties include physics interactions, camera clipping issues, or whether the portal fits into the location where it is supposed to appear. With player-placed portals, games often implement gameplay restrictions to have at least some guarantees about how the portals will be placed. Some examples from Portal™ 2 are:

- Levels are mostly comprised of right angles.
- The game limits where the player can place the portals.
- It is impossible to place portals on top of each other.
- When placed on a tilted surface, the portal orients its top edge to the higher side.

Hidden Portals

Hidden portals bend space in confusing ways and are often used in puzzle games. One example is Antichamber, which presents *"mind-bending challenges"* as its main feature.[13] It is never explicitly revealed that portals are at play in Antichamber, but the game relies on them heavily. A more subtle example is The Stanley Parable[14], which uses portals to guide the player's progression through the game, transporting them seamlessly when they take the wrong turn. Portals are not mentioned in The Stanley Parable marketing materials at all; they only serve as a narrative control tool. Hidden seamless portals have also found some application in VR, where they can be used to fold large in-game areas into a limited physical space.[8]

In an ideal scenario, the player would never notice the fact that the game uses hidden portals for its gameplay. With a sufficiently good portal implementation, there is no way to be sure that these games do not just use a simple teleportation between the same-looking parts of the game world.

[13] `https://store.steampowered.com/app/219890/Antichamber`
[14] `https://store.steampowered.com/app/221910/The_Stanley_Parable`

The hidden type of portal is commonly placed during level design. This provides the game developers with better control over the portal feature, as they can adjust the surroundings of the portal to make it less noticeable. It is important to note that portals are a major feature, and the level design often has to take steps to accommodate them. An example of this is placing a hidden portal into a narrow corridor to guide the players to enter it from a specific angle, which may help to conceal the portal better.

2.3 Common Problems

Implementing seamless portals for a video game brings its fair share of technical difficulties. Many people encountered and solved these problems in many ways. The flagship portal implementation is Portal™ 2, in my opinion. Their expertise and developer talks are a significant source of information about various issues.

2.3.1 Ray Cast Forwarding

Ray casting is often used in video games to check physics collisions along a straight line. For example, shooting a gun in a first-person shooter game might be implemented with the help of a raycast.

The issue with portals is that hitting the portal with a ray cast is rarely the desired outcome. The player expects the ray to go *through* the portal and only hit what is on the other side. To achieve this effect, the portal hit has to be handled in a special way. After hitting the portal, these operations need to be performed on the cast ray:

1. Rotate the ray direction by the portal transform.
2. Translate the ray origin using the portal transform.
3. Shift the ray origin in the new ray direction, so that the forwarded ray originates from the portal surface.
4. Shorten the ray to account for the shift to the portal surface and the distance it has already traveled before hitting the portal.

Additionally, the new ray cast may want to ignore collisions with the portal it just traveled through. After all these operations, the ray can be evaluated again to check for collisions beyond the portal.

2.3.2 Recursive Portal Rendering

Seeing portals through another portal is a complex feature that is difficult to implement correctly. It can produce a similar effect to two mirrors facing each other in real life – a seemingly infinite hallway, as seen in Figure 2.2. Rendering real-time infinite recursive portals is borderline impossible, which is why it is practical to define a recursion limit for such portals. With the recursion limit set to zero, portals are not visible through other portals.

Portal™ 2 features an interesting optimization technique. Only the first two portal recursions are genuine and rendered as they should be. All portals beyond the second recursion level reuse the previous portal texture. At that point, the area that the deeply nested portals occupy on the screen is small enough for the user not to notice this small discrepancy.[9]

There are generally two main ways to render a portal, each one with its tradeoffs [1]:

1. **Texture** – render each portal view into a separate texture and combine all of them for the final output.
2. **Stencil** – use the stencil buffer to only render into the portal rectangle.

Texture Approach

Rendering each portal view into a separate texture is intuitive, but also resource-intensive. An unshaded 3D rectangle mesh can be used as a face of the portal – let's call this rectangle *portal mesh*. The view of the exit portal is then used as a texture for the portal mesh. The texture is mapped so that the screen-space UV coordinates of both portal meshes line up precisely in the same position in both renders. This results in a smooth and seamless portal appearance.

Figure 2.2: Recursive portal rendering in Portal™ 2.

When rendering the inner portals, we use a process similar to Painter's algorithm[10], where the deepest portal is drawn first. Each subsequent iteration is drawn with the previous render incorporated into the portal mesh texture. At least one separate render texture is required for each portal view. The memory requirements for this method grow very quickly with increased recursion limit and the number of portals in the game scene. Figure 2.3 shows a scenario where more than one render texture is needed for a single portal surface.

Stencil Approach

We can utilize the stencil buffer to render portal views into the game scene. This method has no memory overhead compared to a normal render pass, because everything is rendered directly into the back buffer. The downside of this approach is greater rendering complexity, especially when managing the order in which opaque and translucent objects are rendered.[9]

The rendering order starts with the main player view and then descends into the nested portals. When rendering portal surfaces, the recursion level of the portal is written into the stencil buffer. Then, the

view through the portal is rendered and only written into the areas marked by the stencil buffer as portal pixels. Each subsequent portal render increments its respective area in the stencil buffer, and the inner portal view is only written to the areas marked by the incremented value.

More Than One Portal Pair

The most intuitive case for recursive portals is with a single portal pair. However, this arrangement is a special case in many aspects, not excluding portal recursion. More than two portals may exist within a game scene. In such scenarios, portal textures may have to be rendered multiple times – from the player's point of view and the point of view of all the portals in line of sight. See Figure 2.3 for a diagram of a situation where multiple renders of a single portal are necessary.

Portals are already a computationally expensive feature, and rendering them multiple times does not help. The issue could be somewhat mitigated by checking if a portal is visible through any other portal and only then rendering its view of the world. There are a few options for checking if the player has a line of sight to the player camera, such as frustum culling or occlusion culling. All of them are slightly complicated because we are not checking for visibility directly by the player's camera. Instead, we are checking for a line of sight to the point where the player's camera would be if it was looking through any other portals in the scene.

A plausible workaround could be to perform multiple portal-forwarded ray casts (described in Section 2.3.1) from the portal surface to check for a collision with the player's physics body. However, leveraging the physics system to determine visibility is not ideal.

2.3.3 Smooth Teleportation

Teleportation in a video game portal is triggered when the origin of a teleported object crosses the portal plane. This results in the object suddenly disappearing and appearing on the other side, which looks unrealistic to an outside observer.

A common remedy for this issue is to duplicate the incoming object's mesh when it comes near the portal, creating a *clone mesh* on

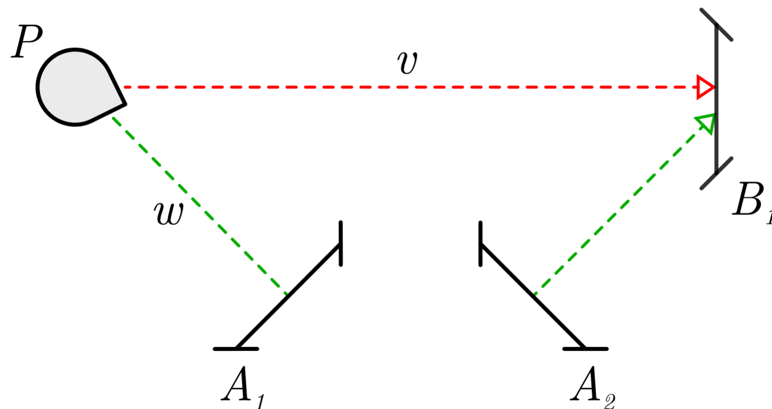


Figure 2.3: Player P can see portal B_1 from two different perspectives. Once along the line v and a second time along the line w, through the portal pair A. Portal B_1 has to be rendered separately for each viewing angle.

the exit side of the portal. Both meshes are clipped by their respective portal planes. As the object comes closer to the portal and parts of its mesh start to poke through the portal mesh, the mesh clone seamlessly covers the clipped parts on the other side. The result is a smooth portal travel when viewed from a third-person perspective.

The effect (or lack thereof) is most noticeable when an object is coming through a portal towards the player. When the object is nearing the portal surface, part of it might get clipped by the portal camera – I discuss this in greater detail in Section 2.3.6. Interestingly, activating this feature for the player character may not be necessary. In many first-person video games, the player's mesh is invisible to the player's camera.

2.3.4 Portals on Walls

The Portal™ games popularized the idea of the player placing a portal directly onto a wall. This, however, carries a significant technical challenge – you have to cut a hole in the wall collider in order for the player

to be able to walk into the portal. Otherwise, the player would just bump into the wall behind the portal and be unable to walk forward through the portal.

Valve solved this problem by first making temporary copies of all colliders in the immediate surroundings of the portal and then cutting a hole in the copied wall collider. Next, when the player comes near the portal, they completely turn off the player's collision with the outside world so that they only collide with the copied colliders. This happens on both sides of the portal at the same time.[9]

2.3.5 Lighting and Shadows

The most popular solution for light correctly shining through a portal, as it would through a window or a door, is ray-traced lighting.[11] With a ray hitting a portal, all you need to do is teleport the ray through it and then continue its propagation. This technique is widely used in non-real-time applications, for example, in Blender's *Cycles* rendering engine.[15] However, seeing real-time ray-traced lighting systems in video games is still somewhat rare today. One of a few examples is a remake of the original Portal™ game with ray-traced lights.[16] This remake has been done by NVIDIA to show off its RTX technology.

Getting light to shine correctly through a portal is extremely difficult with traditional real-time lighting systems. It is possible to make the effect somewhat believable in specific scenarios. After consulting colleagues from the video game industry, I have been unable to formulate a general system using traditional lighting techniques.

Most games that feature portals in a significant capacity employ a particular style of lighting to hide this limitation. Some examples are:

- **Portal™ 2** mainly uses ambient lighting, which fills the levels with soft light. This lighting style avoids sharp light sources and sharp shadows. Examples can be seen in Figures 2.1 and 2.2.
- **Antichamber** has almost no shading at all and no shadows either. Everything is brightly lit. Since both sides of the portal are equally illuminated, shown in Figure 2.4. Point lights are

[15] `https://www.blender.org/features/rendering/#cycles`
[16] https://store.steampowered.com/app/2012840/Portal_with_RTX

used sporadically to guide the game design, not to illuminate surroundings, as seen in Figure 2.1.

- **Splitgate** surrounds its portals with a brightly colored emissive ring, which distracts from the fact that no light is coming through the portal. Additionally, the surfaces on which the players can place their portals also glow. This overpowers any light that should realistically come through the open portal. See Figure 2.4 to see this effect in action.

2.3.6 Portal Plane Clipping

A camera renders each portal surface with the same attributes as the main player camera. If there is an object close enough to the back side of the exit portal for the portal camera to see it, it will show up on the portal surface, which is incorrect. Everything behind the portal's exit needs to be hidden from the portal camera. The following sections describe three possible approaches for portal plane clipping.

General Clip Plane

Using a general clip plane is one way to avoid rendering anything behind the exit portal. One way implement a clip plane is to supply a plane equation as a shader uniform to all objects in the scene and then discard any fragments that are behind that plane. Additionally, some rendering backends have a dedicated mechanisms for arbitrary clipping planes.[17] Aligning the clip plane with the portal plane disables the rendering of the part of the object that is behind the portal. This way, the portal camera only sees what's in front of the portal, which is exactly what it should display. The disadvantage of this approach is that you have to handle the clipping plane in every shader in the game and keep it up to date via shader uniforms.

[17] Clip planes are configured with `gl_ClipDistance` in OpenGL and Vulkan (through SPIR-V). Direct3D exposes the same mechanism via the `SV_ClipDistance` property.

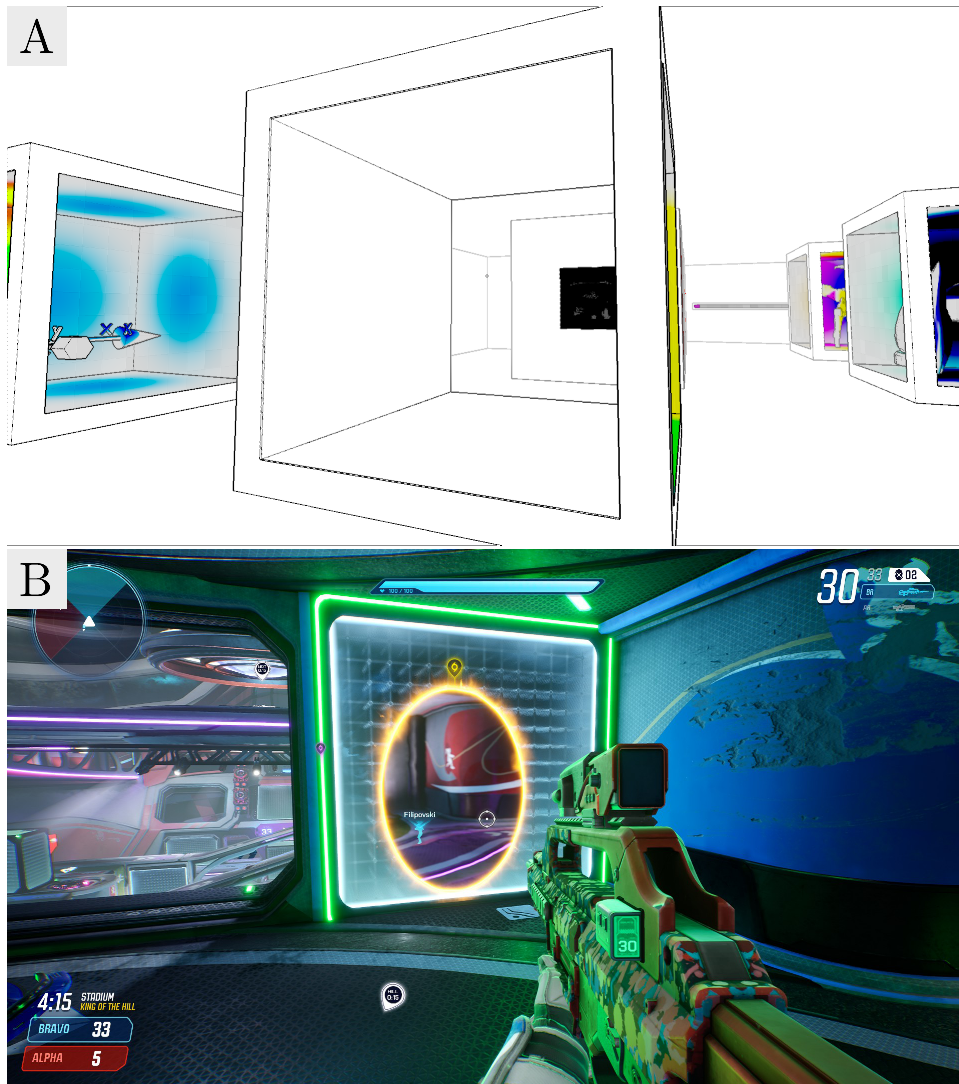


Figure 2.4: Convenient lighting techniques in Antichamber (*A*) and Splitgate (*B*) hide the fact that no light is coming through the portals.

Oblique Projection

Another way to avoid rendering objects behind an exit portal is to modify the projection matrix of the portal camera to use Oblique Projection. This technique moves and skews the near clipping plane of the view frustum to effectively cull everything along an arbitrary plane. Aligning the near clipping plane with the portal frame yields the desired effect. This works regardless The downside of this approach is that the view frustum's far clipping plane is unusable. [12]

Near Distance

The traditional frustum projection matrix culls everything that is too close to the camera. The intended usage of this near-distance parameter is to prevent graphical glitches like seeing the inside of the player character's head. However, it can be used to approximate clipping by the portal plane if the other methods are not available. I describe the implementation of this method in Section 4.2.3.

3 Plugin Design

For this thesis, I developed a Godot Engine plugin that enables developers to easily set up portals in their game. In this chapter, I will describe my main design goals, considering the current state of the game engine.

3.1 Choice of Game Engine

The Godot Engine is going through a period of explosive growth, based on the number of Steam games released with it, as seen in Figure 3.1. It is a popular open-source game engine with an upward popularity trajectory. My goal in this thesis is to explore the engine from an academic point of view.

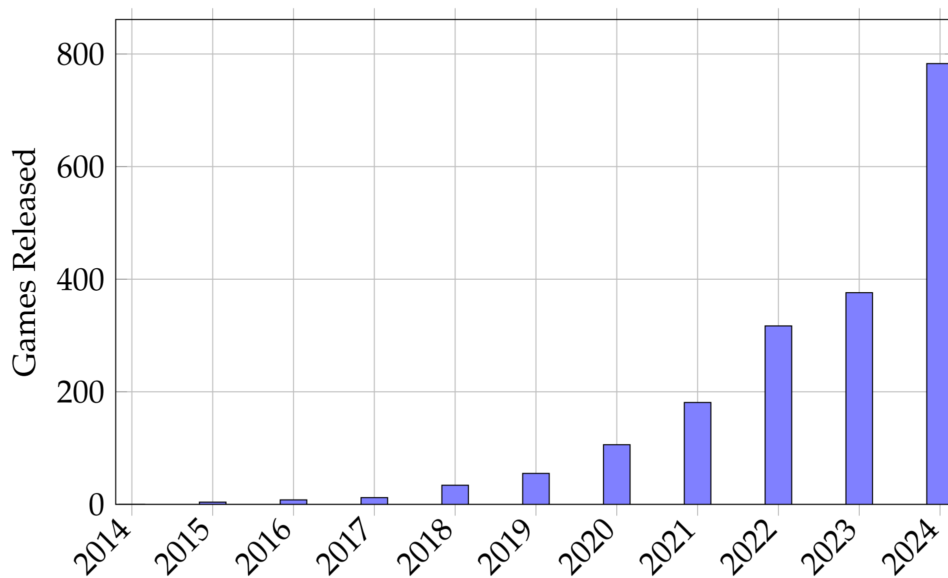


Figure 3.1: Number of Godot games released on Steam per year since its open-source release in 2014.[1]

[1]Steam releases data: `https://steamdb.info/stats/releases/?tech=Engine.Godot`

3.2 Scripting Language

Godot Engine supports GDScript and C# as its first-class scripting languages. On top of that, the community maintains many other language bindings. In order to use C# for scripting in Godot, a separate build of the engine is required, whereas GDScript is built into all Godot Engine distributions.

I implemented the plugin in GDScript rather than C# so that it can reach the largest possible audience. According to a Godot community poll in 2024, GDScript is the scripting language of choice for more than 75% of the respondents.[2]

3.3 Extending the Godot Editor

The Godot Engine editor's functionality can be extended with a plugin mechanism, similar to other major engines. Plugin development is very approachable for anyone with knowledge of the Godot Engine, because the editor itself is a game written in Godot. Here, I will explain some terminology related to the engine's plugin system.

Tool scripts are gameplay scripts that are also executed in the engine editor. To make any script executable within the editor, a `@tool` annotation is required on the first line of the script file. Any node with a tool script attached is then processed within the editor as if it were present in a game.[3] My plugin makes heavy use of this feature. The main portal script is a tool script – it runs part of its logic within the editor and other parts during gameplay.

A plugin in its minimal form is comprised of a metadata file and a single tool script that inherits from the `EditorPlugin` class. This script serves as an entry point for the plugin's lifetime. Within it, other functionality can be inserted into specific areas of the editor.[4] The

[2] `https://godotengine.org/article/godot-community-poll-2024`

[3] `https://docs.godotengine.org/en/stable/tutorials/plugins/running_code_in_the_editor.html`

[4] `https://docs.godotengine.org/en/stable/tutorials/plugins/editor/making_plugins.html`

portal plugin uses this mechanism to add custom portal gizmos – more on that in Section 4.6.4. Plugins are distributed through a first-party platform called Godot Asset Library, integrated into the Godot Engine editor.[5] My plugin is also going to be available through this service.[6]

3.4 Design Philosophy

The central part of the plugin is the portal implementation itself. I aim to offer comprehensive options for accommodating a wide range of use cases. On top of that, the portal script will implement a public API to allow for smooth integration into games. This API will be implemented in a fashion similar to Godot's built-in nodes.

Godot's editor offers many features for integrating player scripts and plugins. The plugin I am developing should make full use of them. That includes, but is not limited to, exposing variables to the editor, centralized plugin settings, type hints wherever possible, and utilizing the built-in documentation mechanism. These integrations are described in Section 4.6. Using the portals plugin should feel native to the editor and familiar to Godot developers.

The portal script should set up part of its node structure within the editor. It is a concept similar to Construction Scripts in Unreal Engine.[7] It helps to visualize where the portal is located in the game world. It also offloads some of the portal initialization to the game engine, instead of doing a complete setup in a gameplay script.

Finally, the plugin should not get in the developer's way. I aim to make as few assumptions about the game being created as possible. A plugin should not dictate how the game is structured; instead, it should offer multiple ways of interacting with its systems, where possible.

[5] Godot Asset Library: `https://godotengine.org/asset-library/asset`

[6] `https://godotengine.org/asset-library/asset/4022`

[7] `https://dev.epicgames.com/documentation/en-us/unreal-engine/construction-script-in-unreal-engine`

4 Plugin Implementation

For this thesis, I implemented seamless 3D portals for the Godot Engine, packaged as a plugin and available to other users. In this chapter, I will describe the general principles of my portal implementation. I will also review the engine's capabilities and the limitations I encountered during the plugin development.

The plugin was developed in Godot Engine version 4.4.1, released on 26th March 2025.

4.1 General Principles

All portal functionality is implemented in a `Portal3D` class, derived from `Node3D`. When `Portal3D` is placed into the game scene in the editor, it creates two child nodes – a mesh instance that represents the portal surface and an area collider that detects teleportable objects. These child nodes are linked to their parent and are managed mostly through the portal node .

Every portal A has to have an exit portal B, which is assigned via the engine inspector. These portals then form a portal pair (A, B), which represents a mapping of positions on the surface of A to the surface of B. This transformation is defined by the following 4×4 matrix formula: [6]

$$M_B^A = M_B \times R_y(\pi) \times M_A^{-1} \tag{1}$$

where

M : Portal transform composed of translation and rotation

M^{-1} : Inverse of M

R_y : Rotation around the Y axis in radians

This transformation is used both for positioning portal cameras and for transporting objects through the paired portals.

4.1.1 Exit Portals

The portal's *exit portal* is a crucial piece of configuration. Two instances of `Portal3D` can be linked together by setting them as the exit portal of each other. This arrangement creates a single bidirectional portal we can see and travel through, as I've hinted in Chapter 1. Exit portals can be configured on each portal instance within the editor inspector – more on that in Section 4.5.1.

The portal scenario we are most familiar with features two portals leading into each other. This, however, does not have to be the case, as Ernest Adams points out in his book:

> *Teleporters can further complicate matters by not always working the same way, teleporting the player to one place the first time they are used, but to somewhere else the second time, and so on. They can also be one-way or two-way, teleporting players somewhere with no way to get back, or allowing them to teleport again.*[13]

While the plugin encourages linking portals together symmetrically, nothing prevents the user from setting any portal as the exit portal. Creating a scenario where portal A leads into portal B, which leads into portal C, is completely possible. It's important to note that walking backwards through an asymmetric portal is not seamless. As soon as the player gets teleported, the portal behind them does not display the original destination.

One-way portals can be created by assigning a deactivated portal B as an exit portal to portal A. In this arrangement, B is invisible and provides no way for the player to go back. This setting is explained in detail in Section 4.5.4.

An interesting edge case I found is setting a portal as its own exit portal. My implementation handles this gracefully, the result can be seen in Figure 4.1. This configuration looks like a mirror at first sight, but the view transformation is wrong. Mirrors flip space along the Z (forward) direction, whereas portals rotate it around the up-axis by 180 degrees. Travelling through such a portal works seamlessly – all the teleportation does in this case is turn the player around.

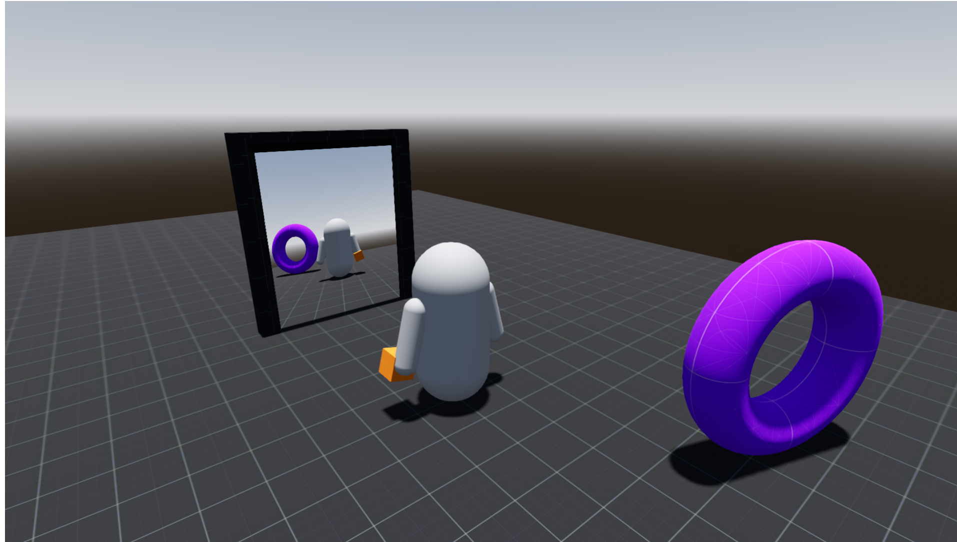


Figure 4.1: Portal leading into itself, being its own exit portal. I used image editing software to place the player's perspective into the portal frame. This configuration does **not** produce a mirror. You can see that the purple torus is on the player's right side both in reality and in the portal.

4.2 Rendering

The **stencil approach** to portal rendering I discussed in Section 2.3.2 is the most desirable, thanks to its minimal memory footprint. However, due to the rendering limitations of Godot Engine, I was not able to implement this system. Godot's render pipeline has historically been rigid, and the engine still does not give developers access to the stencil buffer. Users have been requesting this feature for years now, and work is being done in this area as of the writing of this thesis.[1] However, in Godot version 4.4.1, it is not possible to replicate the portal rendering method used in Portal™ 2.

My next candidate for portal rendering is the **texture approach**, which I also discussed in Section 2.3.2. The main requirement of this approach is the ability to perform multiple customizable opaque ren-

[1]Pull request to give developers access to stencil buffer in Godot: `https://github.com/godotengine/godot/pull/80710`

der passes. This is where I have hit another rendering limitation. Godot version 4.4.1 simply does not support scriptable render passes.

> *"Godot renderer takes objects, renders them, and that's it."*
> – Juan Linietsky, 2023 [2]

Since Godot does not allow for easy rendering of the same scene in multiple ways, I considered a few options for working around this limitation:

- **Render layers** can be used to filter out objects that we do not want the camera to see. This puts a hard upper limit on the number of portals and recursions we can render. Eventually, we run out of Godot's 32 possible render layers. And some of them may be used for gameplay, not just portals.

- **Separate worlds** is a technique in Godot, where the viewport you are rendering to also contains its own `World3D`.[3] This means that the viewport has its own rendering and physics context. Things under this node do not interfere with the objects outside of it. It is a powerful feature, but duplicating the entire scene for each portal recursion would come with a considerable performance penalty.

Ultimately, I deemed both of these workarounds unfeasible for the portal plugin. On top of their various downsides, they both impose significant expectations on how the game should be structured. This is unfit for a plugin to do. In my opinion, a plugin should dictate to its user how to make their game as little as possible.

The solution I developed for rendering portals comes at a significant concession – **recursive portals are not supported**. For recursive portals to be feasible in the Godot Engine, more development needs to happen on making its renderer more flexible.

[2] Proposal for rendering compositor in Godot by Juan Linietsky: `https://github.com/godotengine/godot-proposals/issues/7916`

[3] This behavior is toggled by the Viewport's `own_world` property.

My Solution

Each portal has a *portal mesh* (represented by a `MeshInstance3D` node), a *portal camera* (represented by a `Camera3D` node), and a reference to the player's camera. The portal camera looks through the back of the exit portal, rendering into a separate texture (represented by a `SubViewport` node). This texture is then sent to a custom shader material and used as the albedo color of the portal mesh. The UV mapping is straightforward – each fragment uses its own on-screen UV coordinate to sample the incoming albedo texture. Because the view from the portal camera always has the same aspect ratio as the player camera, the portal surface gets mapped precisely into the space occupied by the exit portal's mesh. The portal mesh is unshaded; its appearance is dictated entirely by what the portal camera sees.

It is essential to position the portal camera correctly. A good intuition for this is to ask, *"What would the player camera see if it were teleported through the portal right now?"*. The solution is to apply the portal transformation (Equation 1) to the player camera and assign the resulting transform to the portal camera. All portals update their cameras in this way every frame.

4.2.1 Portal Mesh

The main task of the portal mesh is to visualize the portal surface in the game. However, another important thing to consider is that the player is expected to walk through this mesh. Visual glitches can occur when traveling through a portal, such as the portal mesh being clipped by the near clipping distance of the player's camera right when they are about to enter the portal. Also, the player might see what is behind the portal for a split second, since they need to physically cross the portal plane to get teleported – see Section 4.3 for a more detailed description.

My solution to these issues is to give the portal mesh a bit of depth so that the player cannot see through it when they come close. This depth comes at a cost – the portal is visible when viewed from the side.

While implementing portal rendering, I tested various meshes to represent the portal surface:

- **Quad mesh** is the obvious first choice, since portals are rectangular flat surfaces. This mesh suffers from all of the issues I described above. Its advantage is that it has zero depth and looks perfect when viewed from the side.

- **Indented quad** is a modified quad mesh with a smaller rectangular section in the middle that's pushed to the back slightly.[4] There are angled margins around the indented region that connect it to the portal's outside rectangular frame. The indent provides the depth necessary to prevent the player from seeing through. The downside of this shape is that it doesn't have a front face, which becomes apparent when the portal is placed in a non-rectangular door frame.

- **Two quad meshes** are two rectangles stacked behind each other to provide the necessary depth. The shortcoming of this approach is that the sides of the portal are not covered. When the player walks sideways into the portal, a slim strip of the surrounding world is visible before they get teleported.

- **Box mesh** covers the shortcomings of the two-quad approach with its side faces. The box mesh needs to disable face culling, in order not to be transparent when viewed from the inside.[5]

- **Inverted box mesh with a flipped front face** is the custom mesh I settled on. It is a modified box mesh with all its faces facing inwards except for the front one, which is facing out. It provides all of the upsides of the box mesh and keeps the default back face culling turned on.

I found modeling portal meshes in Blender confusing, due to its up-axis being Z, and Godot using Y as its up-axis. Portal meshes are very sensitive to the directions their faces are pointing in, so I found it easier to generate the mesh with a script right inside Godot.

[4]This mesh shape is used by an existing portal implementation for Godot: `https://github.com/Donitzo/godot-simple-portal-system`

[5]Box mesh is used by Sebastian Lague in his Unity portal implementation `https://github.com/SebLague/Portals`

With the portal mesh having some depth, the portal script constantly adjusts its position and scale to keep its front face where the portal surface should be. Depending on the direction from which the player is looking, it flips the mesh along the portal's forward direction to face the player.

4.2.2 Tone Mapping Adjustment

Godot uses an `Environment` resource to define many environmental and post-processing effects, such as ambient lighting, skybox, or ambient occlusion. This environment is then used in a game scene by wrapping it in a `WorldEnvironment` node. All cameras that share a viewport with a `WorldEnvironment` node take on its environment settings. It is a convenient way to store environment settings for many cameras in one place. The environment settings can also be overridden on each camera by giving it its own environment resource.

One of the effects defined in a scene environment is tone mapping. This tone mapping is applied to a portal camera, which displays its output on the surface of a portal mesh. And then it is applied a second time by the player camera, seeing the portal. The resulting portals look out of place, as seen in Figure 4.2. This is even more noticeable when the player travels through the portal, and everything snaps back to the normal tone-mapped colors, spoiling the seamless effect that portals are trying to provide.

The solution to double tone mapping is to disable it on the portal cameras. At the game's start, each portal duplicates the environment that is affecting its portal camera. Then, it sets the tone mapping mode to linear with exposure set to 1 on the duplicate – this is equivalent to having no tone mapping at all. The adjusted environment object is set as the environment override on the respective portal camera.

I have not noticed any other properties getting applied twice. Isolating the portal camera completely with a zeroed-out environment override is undesirable. We want settings like sky box or ambient occlusion to trickle down from the "main" environment.

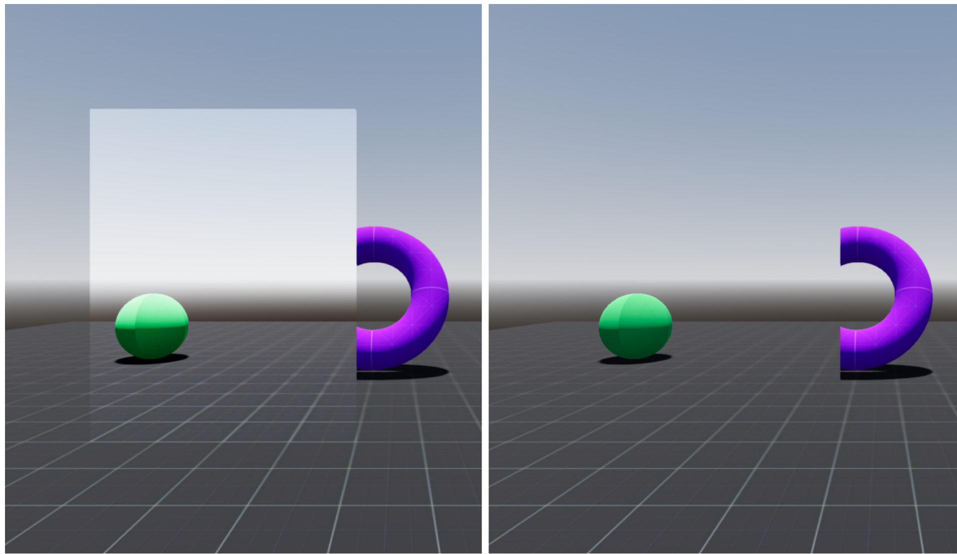


Figure 4.2: In the left screenshot, tone mapping is applied by the portal camera, and then the second time by the player camera. The screenshot on the right contains the fixed version – a completely seamless portal.

4.2.3 Camera Clipping

In Section 2.3.6, I discussed ways to handle camera clipping along the portal plane – shader clip plane and oblique projection. Godot cameras do not allow customizing the projection matrix, which makes oblique projection unusable.[6] Clipping meshes along an arbitrary plane is supported (discussed in more detail in Section 4.3.2). However, without support for multiple render passes, it is not usable in this scenario. Meshes clipped in this way are visible to all cameras in the scene, but we need the clipping to be visible only to the portal cameras.

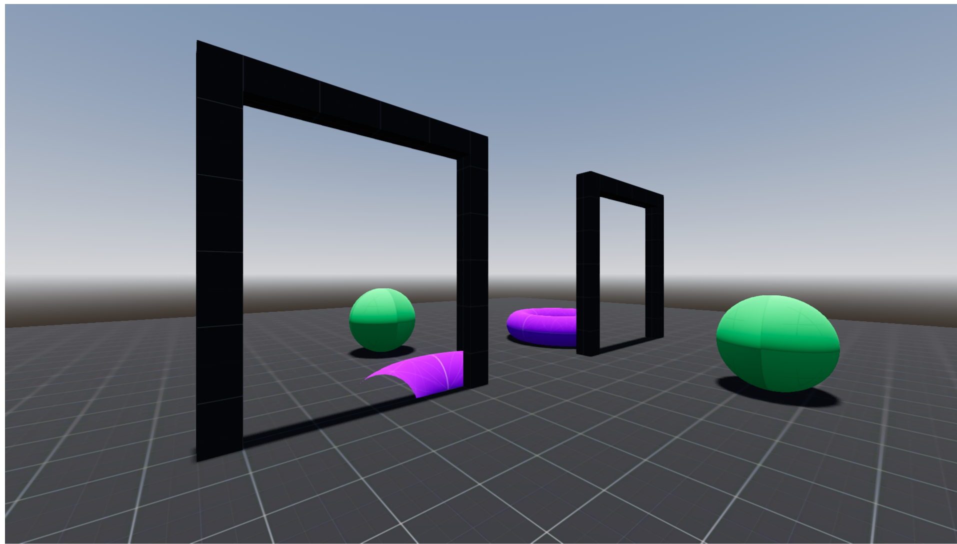
The best solution I found is an imperfect one. I set the near distance of each portal camera as close to the portal surface as possible. Any skewing of the clip plane is out of the question, since the near clip plane's normal vector is aligned with the camera's view direction.

This approach only works perfectly if the player is looking straight at the portal. In other words, when the player camera's view direction

[6]Feature proposal for customizing projection matrix:
`https://github.com/godotengine/godot-proposals/issues/2713`

is perpendicular to the portal surface. But as the viewing angle of the player camera and the portal surface gets smaller, the player is more likely to see what's behind the exit portal (all he should see is what's *in front* of the exit portal). See Figure 4.3 for an example scenario where this clipping method fails.

This limitation can be worked around with level design. Portals inside hallways are convenient because players cannot view the portal from the sides. Similarly, surrounding the portal with a frame mesh (also used in Figure 4.3) limits the most extreme viewing angles, hiding potential glitches. Restricting from which side players enter the portals also helps – if the game only expects the player to enter from the portal front, there is no reason for objects to be present behind the portal.



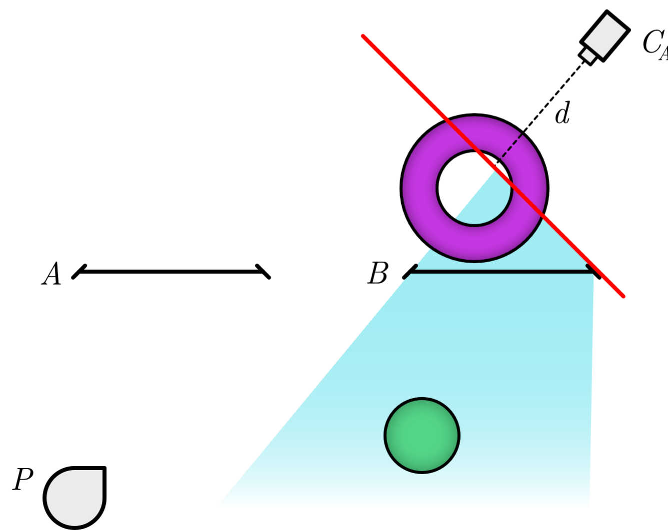


Figure 4.3: The screenshot shows a flaw in the near-plane portal clipping approach, with a corresponding diagram below. Portals A and B are linked together. The camera for portal A sets its near distance d as close to the portal B as possible in an attempt to clip everything behind B. The area visible inside the portal A is highlighted in light blue.

4.3 Teleportation

Each portal features a detection collider (`Area3D` node) that scans for any `CollisionObject3Ds` that enter the portal's proximity. Any object registered by the portal collider is automatically eligible to be teleported. Once a body or an area is detected near the portal, it is placed on a watch list until it leaves or is teleported. Bodies on the watch list are the portal's teleport candidates, and the portal checks every frame to see whether or not they should be teleported. The most important metric for each teleport candidate C is its signed distance along the portal's forward vector:

$$d_f = P_{forward} \cdot (C_{pos} - P_{pos}) \tag{2}$$

This *forward distance* is calculated every frame for all teleport candidates. When the sign of d_f changes, compared to previous frame, it means the object has crossed the portal plane and should be teleported. [14] The portal transformation, described by Equation 1, is immediately applied to the traveling body. It could also happen that the candidate wanders off and leaves the detection collider. In that case, it is removed from the watch list and is no longer monitored by the portal.

The detection collider around the portal acts as a buffer zone, so the portal script can register the incoming objects and monitor them closely in the process loop. Other portal implementations may rely on a very small collider touching the portal surface, which triggers the teleport.[7] I have found this system unreliable, as fast-moving tiny colliders are more likely to pass through the portal in the span of a single physics step than larger ones. Physics simulation runs on a different thread than the render loop, which may introduce inconsistencies and graphical glitches when traveling through portals.

4.3.1 Teleport Root

By default, the object triggering the teleportation mechanism is the node that gets teleported. This aligns with a common pattern in Godot, where all significant nodes of a physics body are its children – for ex-

[7] `https://github.com/Donitzo/godot-simple-portal-system`

ample, its mesh instances. In other words, the physics body is the ideal teleport root node, because it contains all of its important components in a single logical unit. However, my plugin implements a flexible mechanism to override this default behavior.

Each node in Godot Engine can carry *metadata* attached to it. These are arbitrary values stored under string keys.[8] When a teleport candidate is about to get teleported, the portal script checks for the presence of `"teleport_root"` metadata. The value is expected to be a `NodePath`, which points to an ancestor node. If present, the portal script teleports the node specified by this metadata property, rather than the candidate itself.

An alternative approach to using metadata would be to have the teleportable bodies inherit from a plugin-provided class. However, with this feature, I chose to favor composition over inheritance.[15] This approach should provide greater flexibility in the way the game is structured.

Specifying the teleport root is beneficial in scenarios where the entity to be teleported is not a physics body. For example, a magical projectile does not necessarily obey the laws of physics, but you would still expect it to go through the portal. All that's needed is to add an `Area3D` node on it, which, when it gets picked up by the `Portal3D`, causes the entire projectile to teleport.

Furthermore, this feature can also be used to effectively shift the teleport origin point of any object. For instance, placing the teleport trigger (an `Area3D`) onto the player's camera prevents visual glitches when going through portals on floors and ceilings. However, I have found that when dealing with arbitrarily tilted portals, placing the player's camera at the origin point of the player's body yields better results.

4.3.2 Smooth Teleportation

The plugin implements a system for smooth teleportation, as described in Section 2.3.3. This is quite an involved feature that requires significant setup on the user's part. It can also be turned off completely – more on that in Section 4.5.3. The effect is showcased in Figure 4.4.

[8] Implementation detail – Godot actually uses string hashes for faster lookups.

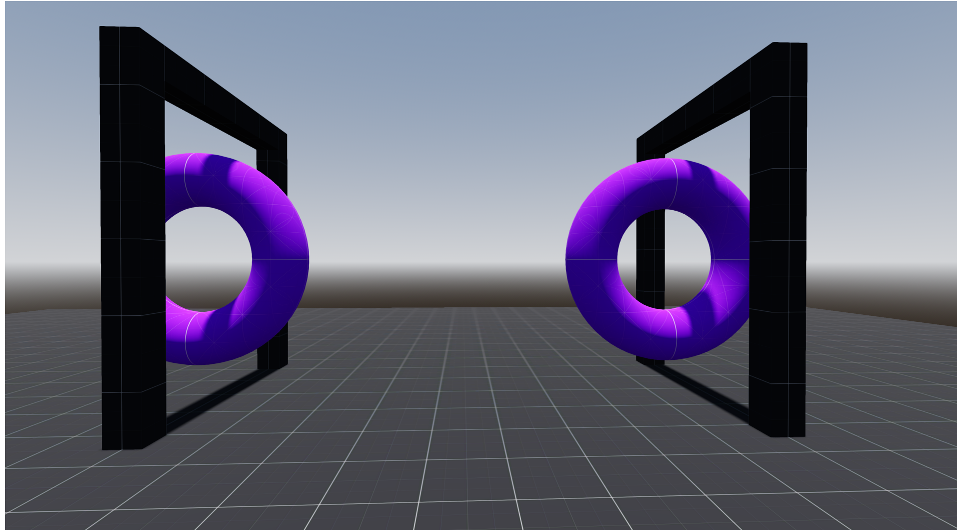


Figure 4.4: A single torus travels infinitely between two portals that are facing each other. The smooth teleportation system ensures none of the meshes are cut off by the portal. The torus on the right is the genuine game object; the one on the left is just a mesh clone.

First of all, the portal must know which meshes to duplicate. To achieve this, the portal script tries to call the `get_teleportable_meshes` method on the incoming object. This method is expected to return an array of `MeshInstance3Ds`. It is up to the plugin user to attach a script to the teleport candidate, which implements this function. If the method isn't found on the teleport candidate, the smooth teleport feature is skipped completely.

An alternative approach would be to query the entire node hierarchy of the teleport candidate for mesh instance nodes. That may be an expensive operation, depending on the size of the node hierarchy of the incoming object. Checking for the presence of a GDScript method is always reasonably fast.

The portal duplicates all meshes retrieved with the callback method, but also keeps track of the originals. Positions of the mesh clones are updated each frame by the portal transformation.

The second part of making this feature work is to clip the traveling meshes along the portal plane. A shader material containing the clipping logic has to be present on all meshes that we want to smoothly

teleport. The portal script then sets up the appropriate shader uniforms, like the portal plane in world space. The shader material then sets the alpha value to zero for each fragment that is on the wrong side of the portal plane, thus achieving the clipping effect.

I rely on Godot's shader pre-processor to make creating the correct shader easier.[9] I prepared a *shader include* file in the plugin files, which the user imports into their own shader. It works just like including a header in C or C++. The included file defines three macros – one for the required uniforms, one for vertex shader operations, and one for fragment shader operations. All the user has to do is include the prepared file in their shader and then place the macros in the correct places. Chances are that the meshes are using resource-based materials, rather than shaders. Luckily, the Godot editor can convert any material to its shader form with a click of a button.

4.4 Public API

The `Portal3D` provides an API to make its integration into games easier. GDScript doesn't have a mechanism for private methods or properties – everything is technically public. However, most of the portal's properties are only meant to be set from the editor inspector (described in Section 4.6.1). There is no guarantee that the portal script will handle unexpected property changes gracefully.

All these methods are documented through the Godot editor's built-in documentation system. See Section 4.6.3 for more details.

4.4.1 Signals

`Portal3D` implements a couple of signals to notify the game that something has been teleported. In both cases, the teleported object is passed to any signal listeners.

- `"on_teleport"` gets emitted when an object goes through a portal and appears on the other side. This event means that the

[9] Godot's shader pre-processor reference:
`https://docs.godotengine.org/en/stable/tutorials/shaders/shader_reference/shader_preprocessor.html`

traveling object is no longer at this portal – it has been teleported to its exit portal.

- `"on_teleport_receive"` gets emitted when an object gets teleported to this portal. The portal emitting the signal has nothing to do with the teleport; it is triggered by some portal that has this portal set as its exit. It makes tracking objects coming out of a particular portal convenient.

4.4.2 Methods

Portal instances can be activated and deactivated via their public methods. Additionally, `Portal3D` features helper methods for raycast forwarding.

- **Deactivate:** The `deactivate` method hides the portal mesh and disables teleportation and processing of the node. A deactivated portal is invisible and does nothing. Optionally, a boolean argument can be passed to the `deactivate` method, indicating whether or not it should also destroy its internal viewport. Destroying the viewport that the portal is rendering to frees up memory. Allocating the render texture again may cause a slight lag, which is why viewports are not destroyed by default. There is also an option to make the portal deactivated by default, described in Section 4.5.4.
- **Activate:** Calling the `activate` method restores the portal to fully active form. Activating a portal that's already active has no effect.

Ray Cast Forwarding

There is nothing special about a raycast hitting a portal collider. When that happens, it's up to the developer if he wants the ray to be forwarded through the portal. If so, the `Portal3D` instance offers helper methods to do just that.

There are two ways 3D ray casts are performed in Godot.

1. `RayCast3D` node can be set up in the scene editor. It is generally the more convenient of the two, providing visualization of the

ray and a graphical interface for its settings. Passing this node directly to the portal's `forward_raycast` method returns the forwarded ray cast results.

2. Querying Godot's physics space directly in code. This is what the `RayCast3D` uses under the hood. The query parameters for the original ray can be passed to the `forward_raycast_query` method. The portal will create a query for the forwarded ray cast, perform it, and return the results.

Both ray cast forwarding methods query Godot's physics space directly. It is up to the plugin user to interpret these results accordingly. The process of ray cast forwarding is described in Section 2.3.1.

4.4.3 Callbacks

The portal script sometimes calls into the objects it's interacting with. For this, it expects the objects to implement methods of particular names and signatures – a feature also known as duck typing. All of these callbacks are optional and can be turned off through an inspector field, described in Section 4.5.3.

- `"on_teleport"` method gets called on an object when it is teleported. This makes it easy to track when a particular object goes through a portal. The same effect could be achieved by subscribing to the signal of the same name on all portals in the scene and then checking if the teleported node was me. The only argument of this method is the teleporting portal, which also references the teleport destination.
- `"get_teleportable_meshes"` is used to get mesh instance nodes for smooth teleportation. This functionality is described in greater detail in Section 4.3.2. This method has no arguments and is expected to return an array of `MeshInstance3D` nodes.

4.5 Configuration Options

The `Portal3D` settings are exposed to the plugin user via the editor inspector. The hard requirement is to connect a portal to its exit portal.

All other fields are populated with sensible defaults, making the portal ready to use. See Figure 4.5 for an overview of default configuration options and values.

4.5.1 General

Some configuration options deserve more attention than others. Therefore, they are exposed in the top section of the `Portal3D` inspector. Other options are hidden away in collapsible sections to prevent visual clutter. The following properties must often be set on each portal instance, even for basic usage. Everything else is pre-filled with sensible defaults, ready to go.

Portal Size

A `Vector2` property indicating the width and height of the portal.[10] This setting affects both the portal mesh and the teleport collider. Portal thickness is determined entirely by the portal script.

The portal size has to be the same as the size of its exit portal. If there is any disparity, the editor displays configuration warnings on the `Portal3D` nodes, described in Section 4.6.2. An inspector button is shown, offering to synchronize the portal sizes. This is useful when pairing up a new portal to a portal that already has its size configured – the user just clicks a button, and the new portal adopts the size of its exit portal. More on this system in Section 4.6.1.

Exit Portal

Another `Portal3D` node that serves as the exit of this portal. Exit portals are described in greater detail in Section 4.1.1.

In most cases, this setting is symmetric. That means that portal A is the exit portal of portal B and vice versa. The inspector tries to make this typical setup easier. When you set A as the exit portal for B while A doesn't have an exit yet, an inspector button shows up and offers to pair the portals together. Clicking it sets portal B as the exit portal of A.

[10] Godot uses the metric system for everything in 3D, with 1 unit being equal to 1 meter.

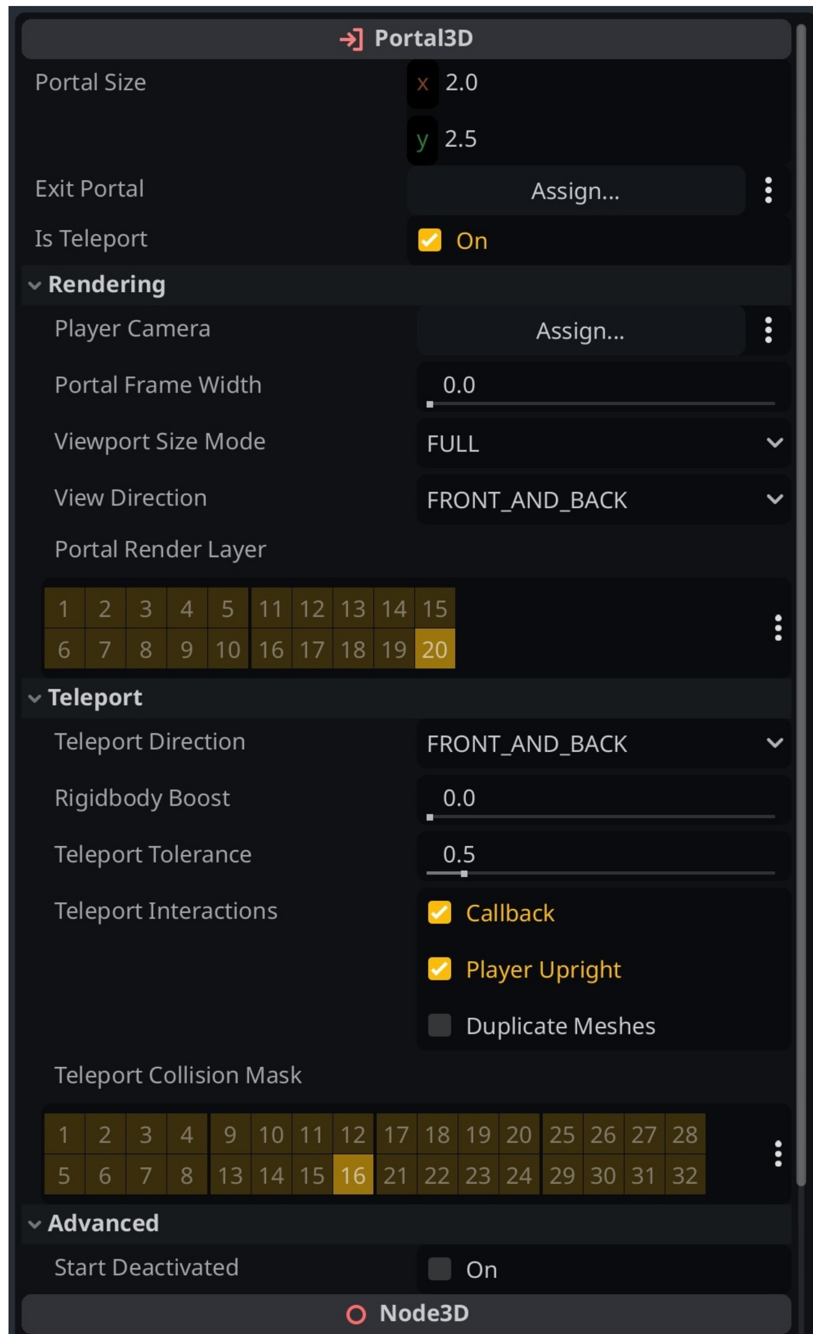


Figure 4.5: Overview of the default configuration options and values. This is the state of a newly created `Portal3D` instance.

The exit portal is a required property; a portal without an exit is pointless. If a `Portal3D`'s exit is unset, the editor warns about this fact with a configuration warning, as described in Section 4.6.2. The portal script also generates errors during gameplay if the exit portal property is not set.

Teleport Toggle

A boolean flag indicating whether or not the portal serves also as a teleport. True by default, but can be turned off to make visual-only portals.

Toggling this property sets up and removes the portal's teleport collider node right within the editor. The *"Teleport"* inspector section is also shown or hidden depending on the value of this property. Hiding an inspector field depending on a checkbox value is a common practice in various areas of the editor, like in material or environment inspectors.[11]

4.5.2 Rendering

Options related to the portal's visual features are grouped under the *"Rendering"* inspector section.

Player Camera

Specifies which camera to use to render the portal's exit view, usually the player camera. If not set, the camera rendering the portal's parent viewport is used. It's useful to specify in games that use multiple cameras or render to multiple viewports.

`Portal3D` uses this system to filter out portal meshes from its exit view (see why recursive portals are not supported in Section 4.2). Portals use the 20th render layer by default, to hopefully stay out of the plugin user's way when developing a game. I don't expect them to interact with this setting often.

[11] New feature is coming in Godot 4.5, which enables a more streamlined way to toggle inspector sections.
`https://github.com/godotengine/godot/pull/105272`

Portal Frame Width

As discussed in Section 4.2.3, my portal implementation uses the camera's near distance in an attempt to clip everything behind a portal. The camera's near distance is set so that the near clipping plane is leaning against the portal as much as possible. The value of this property is subtracted from the portal camera's near distance. One use case for setting this property is not to clip a portal frame mesh surrounding the portal's exit.

Viewport Size Mode

The portal camera renders into an internal viewport, effectively a texture. By default, this viewport is as big as the game window, but for optimization reasons, it might be desirable to make it smaller than that. With smaller portal textures, the portal view may appear blurry, especially up close.

`Portal3D` presents several sizing modes for its internal viewport:

- **Full** mode is the default. The portal renders at the same resolution as the game.
- **Max Width** mode allows the plugin user to specify a maximum width that the portal viewport must not exceed. A number field dynamically appears in the inspector if this mode is selected.
- **Fractional** mode restricts the internal viewport's size to a fraction of the game window resolution. As with the *Max Width* mode, a dynamic number field allows the user to specify the desired size fraction.

View Direction

This property specifies the direction from which the portal is expected to be viewed. As I mentioned in Section 4.2.1, the portal script shifts the portal mesh depending on the direction from which the player is looking at the portal. This behavior might be undesirable if the portal is, for example, supposed to be flush with the wall. By restricting the view direction property to *FRONT* or *BACK*, the portal mesh is locked in one position. The default is both – *FRONT_AND_BACK*.

Portal Render Layer

Godot uses the notion of *render layers* to mask out objects that are not supposed to be rendered. These layers are implemented with a single integer whose individual bits signify if an object should be rendered in a particular layer. Godot uses 32 render layers internally, but the top 12 are reserved by the editor and used for drawing gizmos and overlays.

4.5.3 Teleport

These properties are related to the `Portal3D`'s teleportation features. They are grouped together under the *"Teleport"* inspector section. This entire section is hidden if the portal is visual-only, as shown in Figure 4.6.

Teleport Direction

If an object approaches the portal from the direction specified by this property, it will be teleported; otherwise, it will pass through the portal mesh unaffected. As outlined in Section 4.3, the portal monitors each candidate's forward distance d_f, and triggers teleportation when d_f's sign changes. This corresponds to the *FRONT_AND_BACK* teleport direction. In the alternative modes (*FRONT* or *BACK*), teleportation occurs only when d_f becomes positive or negative, respectively.

The primary use case for restricting the direction of portal travel is when multiple portals are on top of each other. Each portal is a one-way portal, and their teleport directions are set to opposites, in order not to interfere with each other's teleport candidates.

Rigidbody Boost

If the object being teleported is a `RigidBody3D`, the portal takes its current velocity direction and applies a central impulse to the body, multiplied by the boost parameter value. The boost value is zero by default.

This option is inspired by Portal™ 2, where objects thrown into portals get a little velocity boost. This effect generally prevents objects from getting stuck on the portal surface, which might be undesirable.

It is most noticeable when both portals are on the ground and the object rolls into one of them. Instead of stabilizing between the two portals, like a piece of wood would on a water surface, it bobs up and down and is eventually thrown out of the portal. This behavior is physically inaccurate, but then again, we are dealing with portals here.

Teleport Tolerance

If teleportation is triggered and the object is further away from the portal than the teleport tolerance parameter, it is a false positive, and the teleportation is canceled.

This can happen when two portals, A and B, are nearby (or on top of each other). Both portals register the player as a teleport candidate and begin to watch them closely. The player then enters portal *A* and gets teleported. By doing so, they unknowingly cross *B*'s portal plane. Due to the physics system running in a separate thread, it might not be fast enough to notify *B* that the player left its watch zone. Portal *B*, unaware that the player is no longer near, triggers its own teleportation of the player. That is a false positive. Since the player may be very far away from portal *B* at this point, they also get teleported far away from *B*'s exit. This often leads to the player being teleported outside the map, which is a game-breaking bug.

Teleport Interactions

An integer consisting of several bit flags, indicating what should happen when an object is being teleported. The Godot inspector shows each possible flag as its own checkbox. The options are:

- **Callback** – This flag indicates that the portal should attempt to call the `"on_teleport"` method on objects it teleports.
- **Player Upright** – When a player passes through a tilted portal, it is rotated to an upright position with a slight animation. This behavior mirrors the Portal™ 2 game. This behavior could be easily implemented inside the `"on_teleport"` callback, but it is so common that I included it out of the box.

- **Duplicate Meshes** – A flag enabling a callback needed for smooth teleportation. See Section 2.3.3 for the idea of smooth teleportation and Section 4.3.2 for my implementation. This flag is off by default.

Teleport Collision Mask

Specifies a physics collision mask for teleported objects. This setting is directly used by the portal's detection collider. Any object whose physics layer matches this mask is considered a teleport candidate, as described in Section 4.3.

4.5.4 Advanced

This section of the portal inspector is dedicated to settings that do not easily fit into the *"Rendering"* or *"Teleport"* categories. At the same time, they are too niche to warrant a place in the general section.

Start Deactivated

A boolean indicating that the portal should not be active from the game's start. Under the hood, it uses the *deactivate* mechanism, described in Section 4.4.2. The purpose of this option is optimization – a deactivated portal does not allocate its internal viewport texture, which consumes a lot of memory. With portals being inactive by default, they don't consume any texture memory and barely any processing power.

Another use case for inactive portals is for one-way portals or visual-only portals. Inactive portals can still serve as exit portals for others, but function only as reference points for the portal camera and teleportation. Exit portals still have to be set on inactive portals, but they have no effect until the portal is activated.

Portals that are deactivated from the start have to be explicitly activated via script.

4.6 Editor Integrations

Since my portal implementation is packaged as a plugin for the Godot Engine, it utilizes many features that the Godot editor offers. In this section, I will review those features and how I use them in my plugin. None of these are strictly required for implementing portals in Godot Engine, but they elevate the user experience for the plugin user.

4.6.1 Inspector

Godot Engine provides a way to expose script properties to the editor. A simple way is to prefix a script variable with the `@export` annotation. Later, when the engine parses the script, it collects metadata about its properties. Exported variables have their type, default value, and usage noted and used to construct the script inspector. Their overrides are also automatically serialized and stored in the scene file. The downside of this approach is that these exported variables are static and cannot react to the set values in any way.

Since `Portal3D` is a tool script, it can override an editor-only method to specify its exported properties.[12] This method returns an array of dictionaries. The returned dictionaries precisely follow the metadata format that the engine collects about exported variables. We can hook into the editor's inspector-building system and define our custom inspector. The power of this approach lies in the fact that a method is producing this metadata. Therefore, it can react to the current state of the script and produce different results for different variable values.

I use the dynamic inspector abilities to show inspector buttons and variable fields only when necessary, or to hide unnecessary sections altogether. See Figure 4.6 for examples of dynamic inspector fields.

Two more built-in method overrides are required to get the default values working with the customized inspector. The first one returns a boolean indicating whether or not a variable of a particular name has a default value. The second one provides the default value itself. The advantage here is that you can query the default values dynamically from project settings or other script fields.

[12] `https://docs.godotengine.org/en/4.4/classes/class_object.html#class-object-private-method-get-property-list`

Defining inspector fields with a script method also has downsides. I noticed that all manually defined inspector field values get saved to the scene file, whereas only value overrides are stored for variables marked with `@export`. This makes the scene files marginally bigger on disk. But more importantly, changing defaults is more complicated, as everything is always serialized and saved.

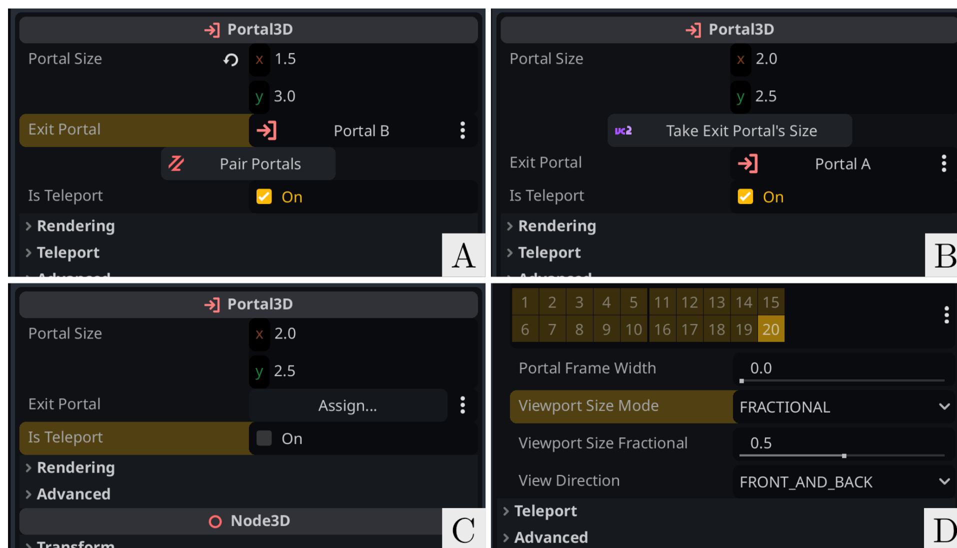


Figure 4.6: Inspector fields change dynamically, depending on the state of the portal. In section *A*, it offers a button to link portals together if the exit portal doesn't yet have an exit. In section *B*, a button makes adopting the exit portal's size straightforward. In section *C*, the *Teleport* group is hidden completely, since the portal has teleporting turned off. In section *D*, a new field is shown when the *Viewport Size Mode* is set to *Fractional*. The default inspector state is shown in Figure 4.5.

4.6.2 Configuration Warnings

Another method that tool scripts can override is used to define configuration warnings. This method returns an array of strings, which are things you want to warn the user about. These warnings appear as a yellow triangle in the scene tree next to the node they affect. Nodes built into the Godot Engine also provide warnings through this sys-

tem, which makes the portal script warnings look native in the Godot editor.

The `Portal3D` script uses this system mainly to warn against different portal sizes. Section 1.2.4 discusses why varying portal sizes are problematic. The simplest cause for the size difference is the user inputting the wrong value into the inspector. This is easy to spot and correct. A less apparent reason is that one of the portal ancestor nodes has scaling applied. This is much more difficult to spot and causes trouble during teleportation, since it uses the global transform of the `Portal3D` node. Scaling the portal itself is also forbidden. Lastly, having an unset exit portal also generates a warning.

4.6.3 Documentation

The Godot editor features a dedicated documentation viewer. In addition to all built-in nodes, user scripts are parsed automatically, and their respective documentation pages are generated. The portal script makes full use of this feature. I took care to provide documentation comments to all important parts of `Portal3D`, including the appropriate formatting for the documentation viewer.

4.6.4 Gizmos

I developed two gizmo plugins to make working with portals easier in the scene editor. Both gizmos are shown in Figure 4.7.

The first one outlines a portal's exit. When a `Portal3D` node is selected in the 3D scene editor, its exit portal gets a colored outline. This visualizes where the portal leads, which makes creating a scene with a lot of portals much easier.

The second gizmo plugin visualizes the portal's front direction. As I've already mentioned, portals are sensitive to orientation, and it's important to keep track of where the portal is facing. This gizmo just adds a short pink line going from the portal center. The line turns into a thicker arrow when the portal is selected.

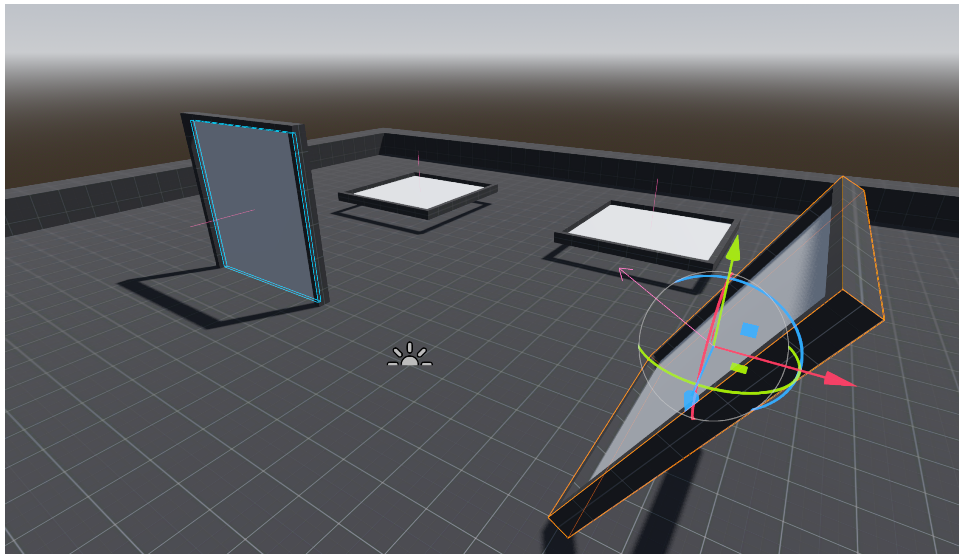


Figure 4.7: The figure shows the tilted portal on the right selected. The portal on the left is highlighted with a light blue outline, clearly marking the selected portal's exit. Additionally, the forward direction of each portal is indicated with a thin pink line.

4.6.5 Project Settings

On startup, the portal plugin registers its own settings into Godot's project settings. These settings mainly control gizmos, described in the previous section. Each gizmo can be deactivated or have its color modified through the built-in project settings window.

Custom project settings are registered with metadata dictionaries similar to those required for custom inspector fields. I was unable to provide documentation along with the custom settings due to a missing feature. The plugin settings are registered on editor startup, but the documentation for the built-in settings is baked into the editor engine build. The Godot editor has no mechanism for documenting the new project settings.

5 Conclusion

For this thesis, I have examined existing video games that use portals as one of their mechanics. I have researched what constitutes a seamless portal implementation, documenting its technical difficulties. I then went on to implement portals inside of Godot Engine, packaged as a plugin for everyone to use in their games.

The plugin enables easy creation of seamless portals right within the Godot Engine editor. Portal functionality facilitates both rendering setup and teleportation, the two main components that make up a seamless portal.

The portal script also features a wide range of configuration options to accommodate various use cases. Its public API should make integrating portals into games easy. I've taken care to implement the plugin's features in a way that is familiar to experienced Godot users, honoring the engine's paradigm and integrating it tightly with its editor.

5.1 Future Work

Creating the illusion of seamless portal travel requires careful coordination of a few tightly integrated systems. My plugin provides a functional and unopinionated portal implementation that can be expanded to accommodate further use cases.

Recursive Portal Rendering

Rendering portals inside of portals is not supported. This is due to engine limitations, as discussed in Section 4.2. Work is being done to make Godot's renderer more accessible to its users. I'm convinced that it is eventually going to be possible to implement recursive portals properly. A proof-of-concept recursive portals can be made today in Godot Engine, but the way to do so is so resource-intensive that I deem it to be unfeasible for a production-grade portal game.

Custom Portal Meshes

Currently, all portals made with my plugin are rectangular – I discussed the different choices for portal mesh in Section 4.2.1. Due to the popularity of Portal™ 2, video-game portals are often depicted as ovals. The portal script could be expanded to be able to use a user-provided portal mesh. Currently, it is tailor-made to work with this custom adaptation of a box mesh.

Physical Clones

The smooth teleportation effect described in Section 4.3.2 is visual only. The portal duplicates meshes of the object being teleported, but they don't interact with physics at all. Giving the cloned meshes a collider that feeds back into the original physics body would enhance the realism of teleported bodies. For example, an object would be able to push objects on the other side of the portal without traveling through it.

Bibliography

1. *CS50's Introduction to Game Development, Lecture 11: Portal Problems* [online]. Cambridge, Massachusetts, USA: Harvard University, 2018 [visited on 2025-04-02]. Available from: `https://cs50.harvard.edu/games/2018/notes/11/`.
2. KOTZIAMPASIS, Ioannis; SIDWELL, Nathan; CHALMERS, Alan. Portals: Aiding Navigation in Virtual Museums. In: *VAST*. 2003, pp. 149–154.
3. LUEBKE, David; GEORGES, Chris. Portals and mirrors: simple, fast evaluation of potentially visible sets. In: *Proceedings of the 1995 Symposium on Interactive 3D Graphics*. Monterey, California, USA: Association for Computing Machinery, 1995, 105–ff. I3D '95. ISBN 0897917367. Available from DOI: `10.1145/199404.199422`.
4. LOWE, Nick; DATTA, Amitava. A fragment culling technique for rendering arbitrary portals. In: *International Conference on Computational Science*. Springer, 2003, pp. 915–924.
5. FOALE, Cameron; VAMPLEW, Peter. Portal-based sound propagation for first-person computer games. In: *Proceedings of the 4th Australasian conference on Interactive entertainment*. 2007, pp. 1–8. Available from DOI: `10.1145/1367956.1367965`.
6. SILVA, Luca; VALENÇA, Lucas; GOMES, Arlindo; FIGUEIREDO, Lucas; TEICHRIEB, Veronica. GoThrough: a Tool for Creating and Visualizing Impossible 3D Worlds Using Portals. In: *2020 19th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*. 2020, pp. 97–106. Available from DOI: `10.1109/SBGames51465.2020.00023`.
7. SILVA, Luca; FIGUEIREDO, Lucas Silva; GOMES, Arlindo; TEICHRIEB, Veronica. GoThrough: A tool to render and interact with arbitrary planar transformative portals. *Entertainment Computing*. 2023, vol. 44, p. 100529.
8. GÜLCÜ, Ali Emre; ATALAY, F. Betül. Infinite Spaces Using Recursive Portals. In: *2022 7th International Conference on Computer Science and Engineering (UBMK)*. 2022, pp. 332–337. Available from DOI: `10.1109/UBMK55850.2022.9919479`.

9. *Valve developers discuss Portal problems* [online]. Cambridge, Massachusetts, USA: Harvard University, 2018 [visited on 2025-04-02]. Available from: `https://cs50.harvard.edu/games/2018/weeks/11/`.

10. *Visible Surface Determination: Painter's Algorithm* [online]. ACM SIGGRAPH Education Committee, the Hypermedia, Visualization Laboratory, Georgia State University, and the National Science Foundation, 2005 [visited on 2025-04-09]. Available from: `https://education.siggraph.org/static/HyperGraph/scanline/visibility/painter.htm`.

11. SUBILEAU, Thomas; MELLADO, Nicolas; VANDERHAEGHE, David; PAULIN, Mathias. RayPortals: a light transport editing framework. *The Visual Computer*. 2017, vol. 33, pp. 129–138.

12. LENGYEL, Eric. Oblique View Frustum Depth Projection and Clipping. *J. Game Dev.* 2005, vol. 1, no. 2, pp. 1–16.

13. ADAMS, Ernest. *Fundamentals of Game Design*. Pearson Education, 2013. ISBN 9780133435719. Available also from: `https://books.google.cz/books?id=Lm1jAgAAQBAJ`.

14. METZLER-GILBERTZ, Reed. *Rendering Portals* [online]. [N.d.]. [visited on 2025-05-18]. Available from: `https://www.cs.rpi.edu/~cutler/classes/advancedgraphics/S21/final_projects/metzlr.pdf`.

15. GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN 0-201-63361-2.

A Installation Instructions

The portals plugin is created in Godot version 4.4.1, released on 26th March 2025. You can download the engine version from the official website.[1]

There are several ways to download the portal plugin I developed for this thesis:

- The plugin is available through the official plugin distribution platform called Asset Library.[2] You can search for it by going to the *AssetLib* editor window and searching for *"Portals 3D"*, as shown in Figure A.1.
- The project is also freely available on GitHub.[3] You can get the plugin by downloading the project as a ZIP archive.
- The entire Godot project in which I developed the plugin is included in the IS MUNI vault, along with this thesis. To use the plugin, download the entire project and place the `addons/portals` directory into your own project.

After downloading the plugin and placing it in your project's `addons` folder, enable it in *Project Settings* → *Plugins*.[4]

[1] `https://godotengine.org`

[2] `https://godotengine.org/asset-library/asset/4022`

[3] `https://github.com/VojtaStruhar/godot-portals-plugin`

[4] `https://docs.godotengine.org/en/stable/tutorials/plugins/editor/installing_plugins.html`

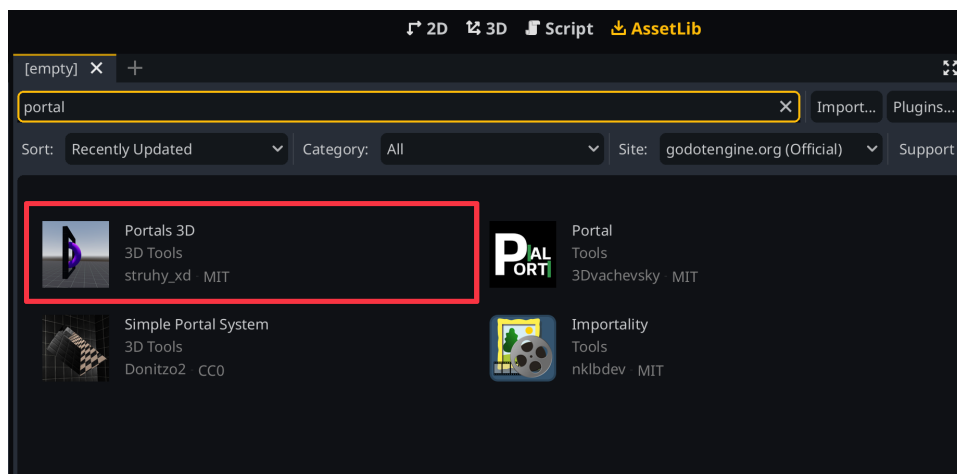


Figure A.1: The *Portals 3D* plugin is available to install right from the Godot editor. Clicking it will prompt you to download the plugin.

B Example Workflow

To give an example of the plugin workflow, I recreated one of the first puzzles of Antichamber – Many Paths To Nowhere.[1]

The puzzle starts out with the player walking through a corridor and encountering two sets of stairs – red and blue. The red staircase leads down, the blue staircase leads up. Both staircases eventually lead the player through a corridor and back in front of the stairs. It seems impossible to progress, since any path the player picks leads him where he started. Even going backwards leads to the last used staircase.

Intended Walkthrough

Once the player tries out both staircases, the level changes. Going back to where he came from now results in him progressing further in the game.

The solution to this puzzle is to do this sequence of actions:

1. Walk through one of the staircases, completing a first lap around the level.
2. Walk through the second staircase, completing a second lap around the level.
3. Turn around and go back to where you came from.

Level Setup

I modeled the level in Blender[2] and imported it into Godot. To mimic the outline aesthetic from Antichamber, I created a separate wireframe mesh for all of the corridors in the level. The alternative would be to create the outlines in a post-processing shader. However, this effect needs to run on all of the portal cameras, making it pretty expensive.

[1] `https://antichamber.fandom.com/wiki/Many_Paths_To_Nowhere`
[2] `https://www.blender.org`

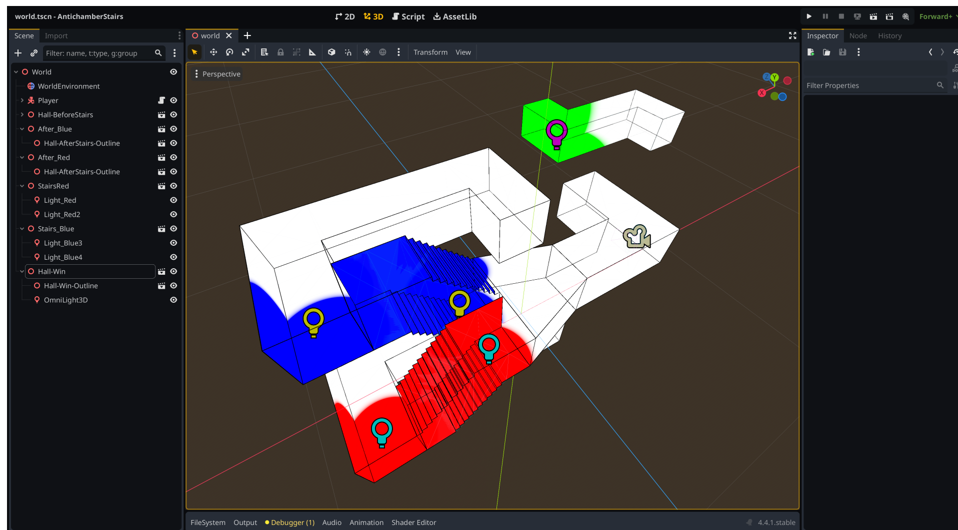


Figure B.1: Level layout in Godot, replicating the *Many Paths To Nowhere* puzzle from Antichamber.

Additionally, the portal meshes would need to be excluded from the outlines, in order not to give away the locations of the portals.

Another staple of Antichamber's aesthetic is its bright environment. This is achieved by setting the ambient light source to a white color inside the world environment settings. It, unfortunately, also means that adding more lights to the scene doesn't have much effect. Point lights[3] are needed to simulate the color bleeding from the stairway corridors. To get this effect, the lights I used are *negative*. This just means that their lighting effect is inverted – darkening its surroundings and casting bright shadows. That is why the light gizmos in Figure B.1 have a cyan color, even though they are located on the red stairway. The walls around the light are white. Subtracting the cyan color results in red.

[3]Point lights are provided by the `OmniLight3D` node in Godot.

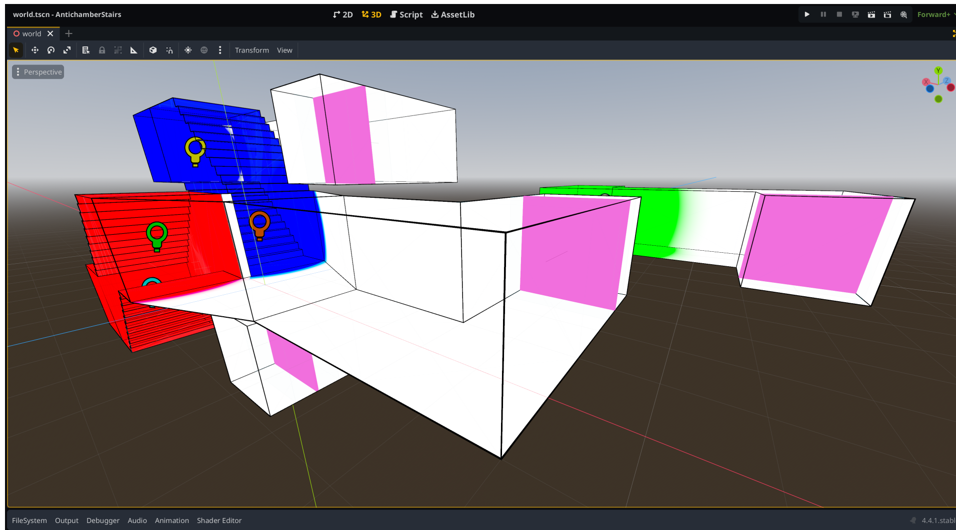


Figure B.2: All 4 portal locations in the scene. Portals only have a bright pink color for visualization purposes.

Portal Setup

The level's geometry is set up, but none of its corridors lead anywhere. To make traveling possible, I placed 4 portals into the scene, as shown in Figure B.2.

- StartPortal is the first portal that the player goes through to enter the level. It also serves as the goal, once the puzzle is completed.
- StairsPortal leads to the two staircases.
- RedPortal is in a corridor following the red staircase.
- BluePortal is in a corridor following the blue staircase.

All portals are exactly 2 meters tall and 2 meters wide, to fill out the entire corridor. The StartPortal and StairsPortal form a portal pair. This is just their initial connection, it will be changing during gameplay. The Red and Blue portals also lead into the StairsPortal to enable running in circles.

Transitions Logic

It's time to write a script that will manage the portal connections. Any portal can be linked to a different one simply by setting its *exit portal* property, which I described in Section 4.5.1. Here are the technical rules of the puzzle I described above:

- When a player passes through the RedPortal, set it as the exit portal of StairsPortal. This enables the player to backtrack in a way that makes sense from their perspective. Also, let's remember that the player already passed through Red once.
- BluePortal works identically to the RedPortal.
- Once the player passes through both Red and Blue portals at least once, the StairsPortal is connected to the level exit. In this case, it is back to the StartPortal.

We can listen to the `"on_teleport"` signals of the Red and Blue portals to know whenever a player passes through them. These signals are described in Section 4.4.1. It then takes only two more boolean variables for us to keep track of which portals the player traveled through. Inside the signal handler functions, we can manage the connections to StairsPortal accordingly.

This entire behavior can be modeled in just a few lines of GDScript.

C Screenshots

In this appendix, I'll show off some of the interesting effects that are possible with my plugin. It is difficult to capture the essence of seamless portals in a still image. Without the context of player movement, they look like nothing out of the ordinary. For that reason, most of the showcase images are made up of multiple screenshots around the same level.

I created many small levels alongside the portal functionality, mainly to test and debug various `Portal3D` features. All of the levels featured in the following figures are a part of the main portal project, submitted together with this thesis. You can see all the levels for yourself and explore how they are made.

The mesh textures you can see in this section's figures are part of the Prototype Textures asset pack made by Kenney, licensed under the CC0 license.[1]

[1] `https://www.kenney.nl/assets/prototype-textures`

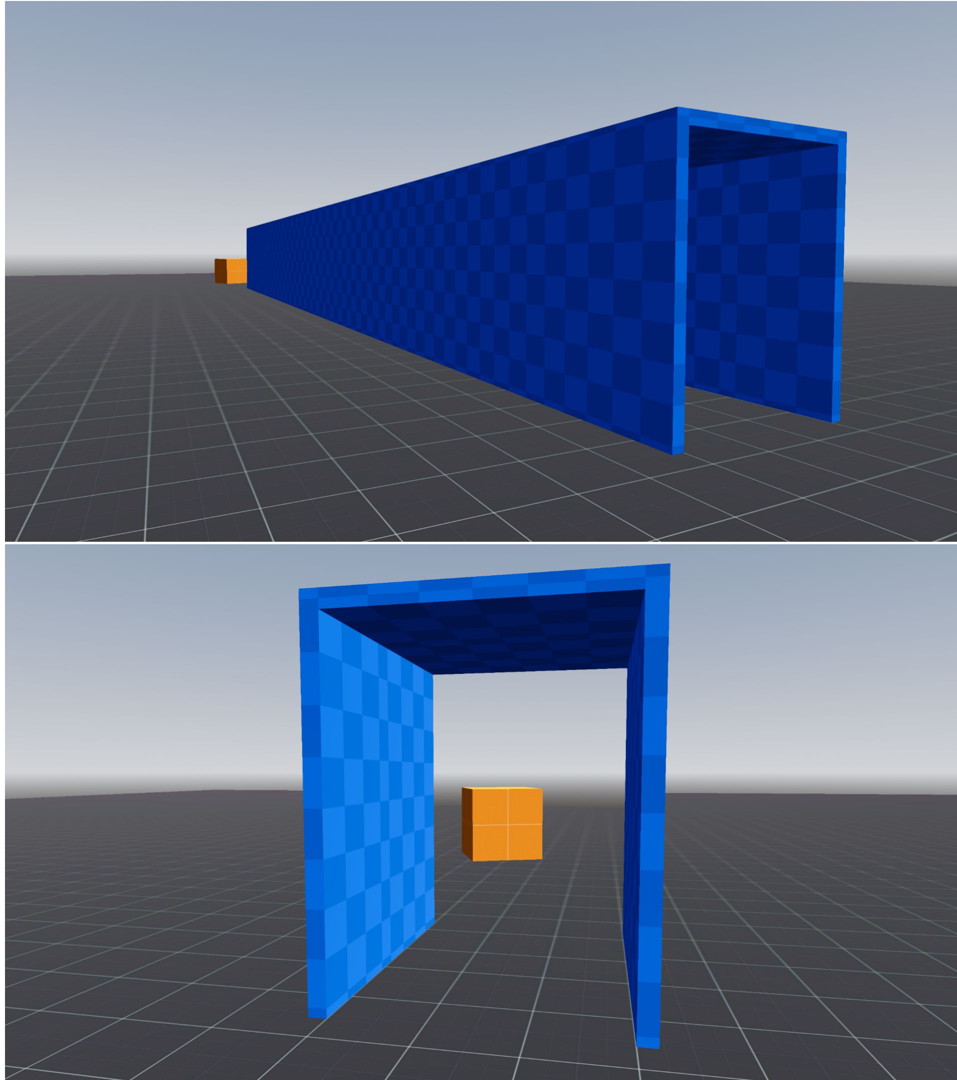


Figure C.1: Corridor that is shorter on the inside than on the outside. This is achieved simply by placing a portal on each end of the hallway.

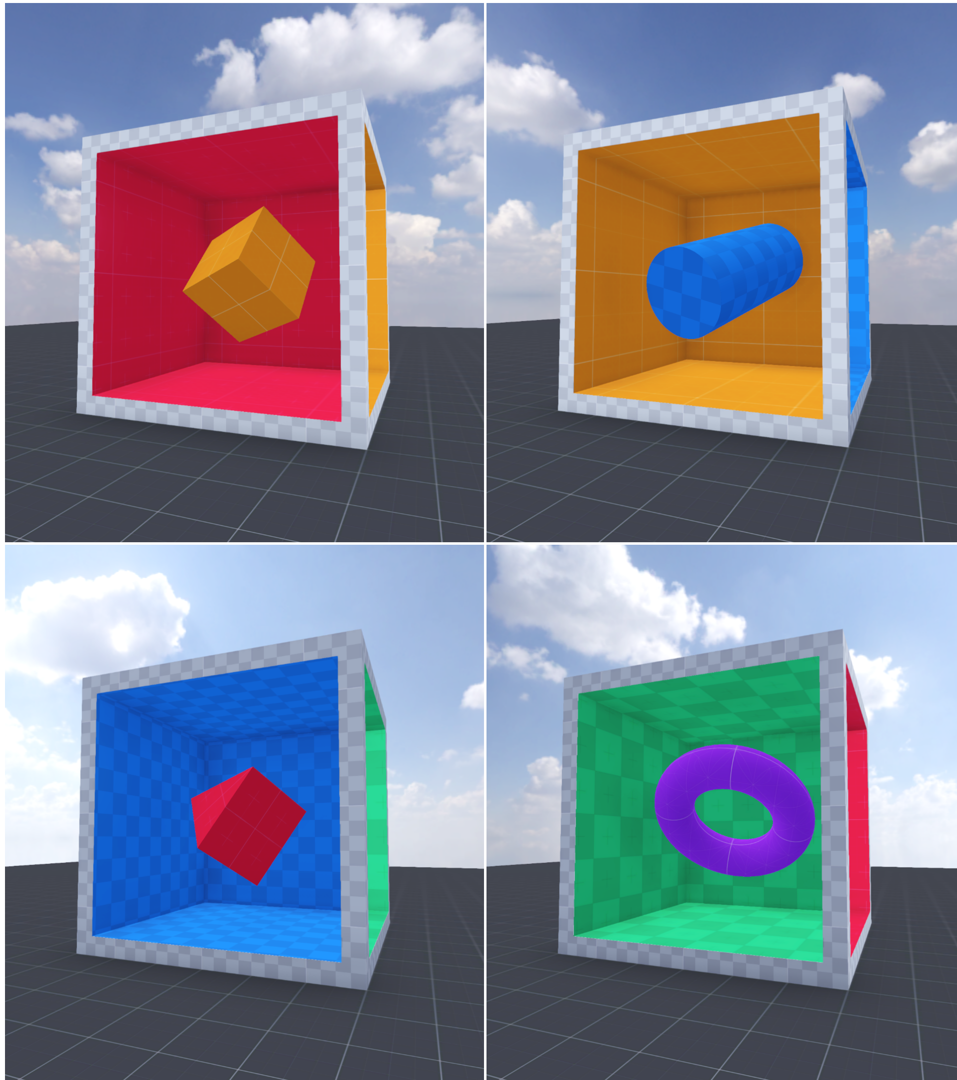


Figure C.2: The figure shows an impossible cube. Each side is covered with a portal, each showing a different inside of the cube. [a]

[a]Skybox (CC0 license): https://polyhaven.com/a/kloofendal_48d_partly_cloudy_puresky

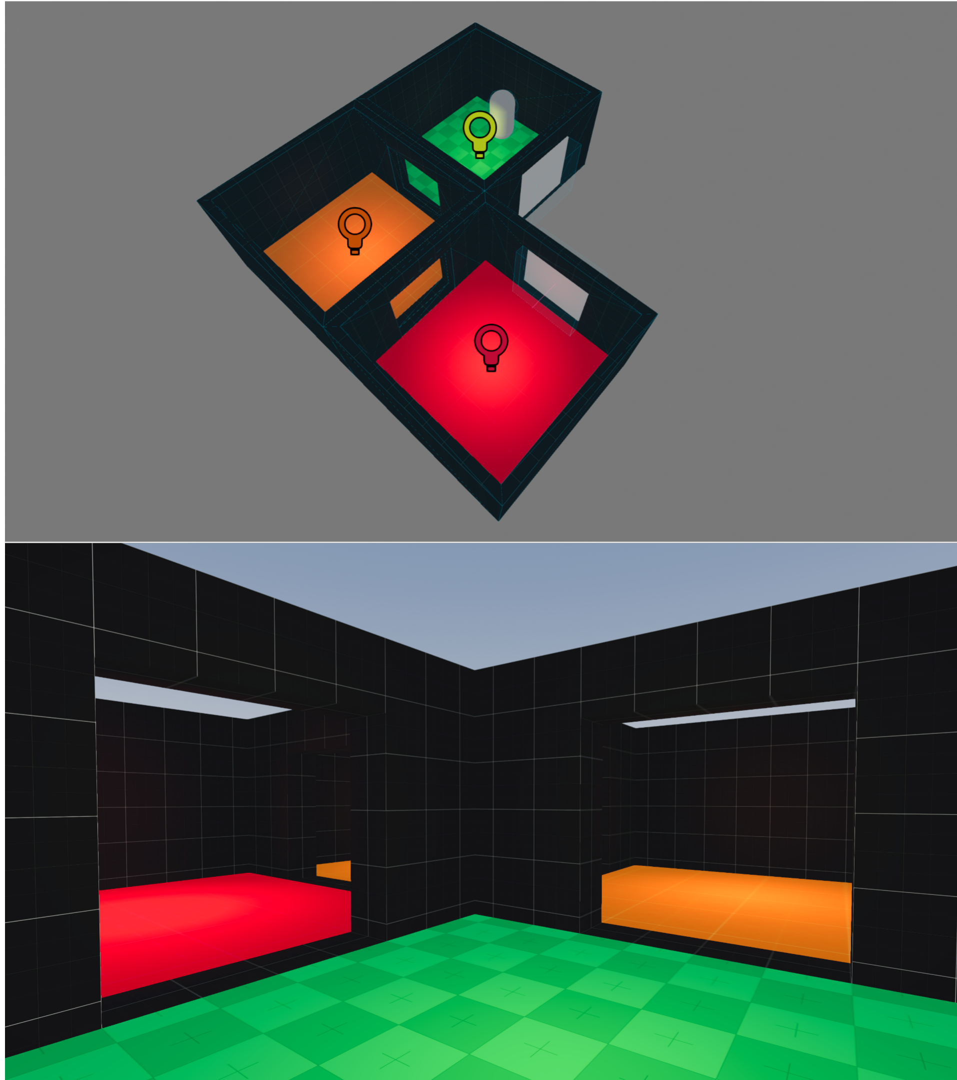


Figure C.3: Three rooms seemingly make up a right-angle corner in the middle. The red and green rooms are linked by a portal; the other connections are natural.

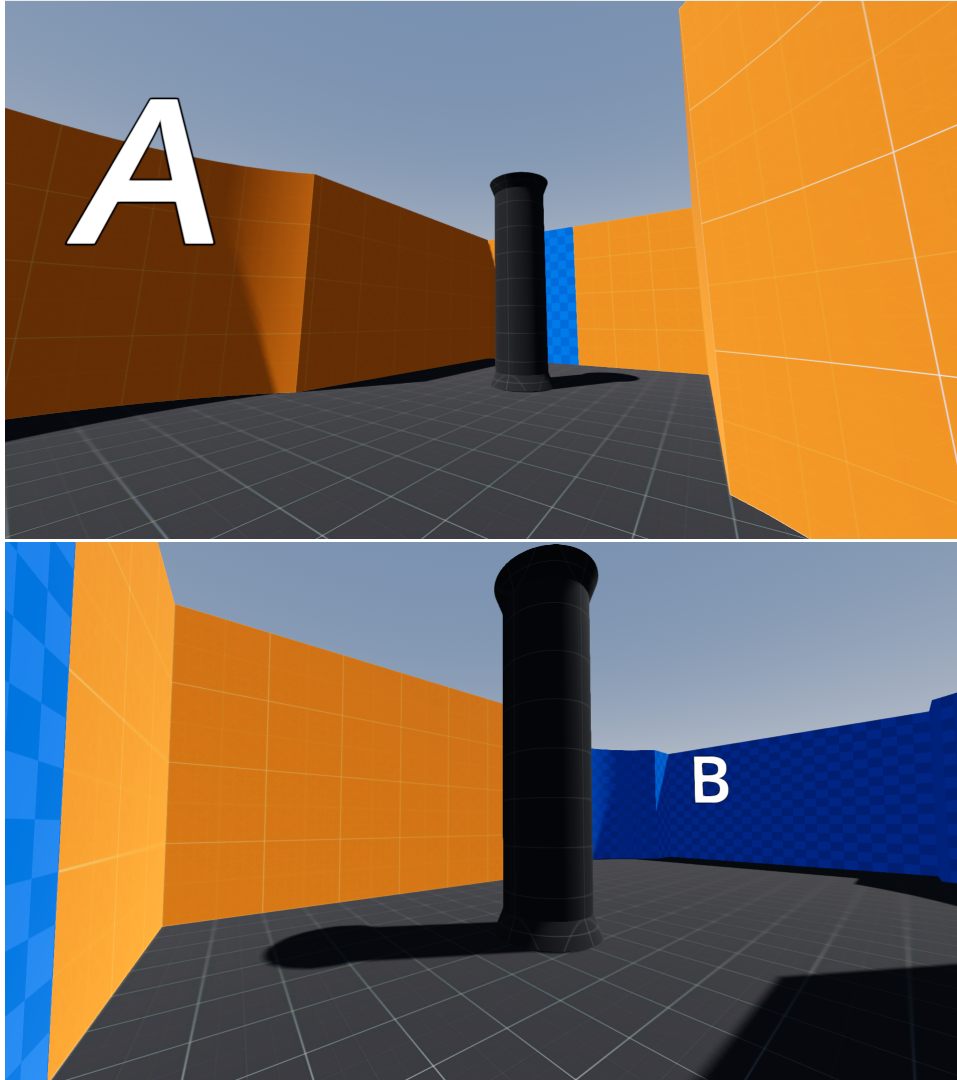


Figure C.4: A portal lies behind the column, hidden in plain sight. By walking around the column, we enter an entirely different space. This goes to show that convenient level design makes the portal truly hard to discover.