Visualization and Physical Interaction of Non-Euclidean Spaces in Euclidean Game Engines

Edin Suta

Bachelor’s Thesis in Computer Science
at Stockholm University, Sweden, 2021

Stockholm University

Visualization and Physical Interaction of Non-Euclidean Spaces in Euclidean Game Engines

Edin Suta

Bachelor’s Thesis in Computer Science (15 ECTS credits)
Bachelor’s Programme in Computer Science
Stockholm University year 2021
Supervisor at the Department of Mathematics was Lars Arvestad
Examiner was Woosok Moon

Department of Mathematics
Stockholm University
SE-106 91 Stockholm, Sweden

Abstract

This report finds and implements a method for portraying non-euclidean spaces in a modern, euclidean game engine that also allows the player to interact with these non-euclidean spaces in a believable way, in real-time. The method does not allow the use of ray-tracing or modification of game engine source code. The chosen method is a portal system. The portals created by this portal system are completely seamless and thus, when placed in certain configurations, they can make a euclidean space appear non-euclidean. Since the system essentially creates an illusion of a non-euclidean space, it does not break any of the underlying logic in the euclidean game engine or its scripting API. A performance analysis is performed on the end-result (before any optimization) which concludes that the method has a linear time complexity (portal rendering cost grows linearly with the complexity of the 3D scene). This is deemed to not be ideal for large, complex game environments, but suitable for enclosed, less complex game environments. If the system is optimized and some of its other flaws get fixed or improved, it has the potential to be a universal solution for games needing to portray non-euclidean concepts.

Visualisering och fysisk interaktion av icke-euklidiska rum i euklidiska spelmotorer

Sammanfattning

Denna uppsats hittar och implementerar en metod för att framställa icke-euklidiska utrymmen i en modern, euklidisk spelmotor som också gör det möjligt för spelaren att interagera med dessa icke-euklidiska utrymmen på ett trovärdigt sätt, i realtid. Metoden tillåter inte användning av ray-tracing eller modifiering av källkod i spelmotorer. Den valda metoden är ett portalsystem. Portalerna som skapats av detta portalsystem är helt sömlösa och kan därför, när de placeras i vissa konfigurationer, få ett euklidiskt utrymme att verka icke-euklidiskt. Eftersom systemet i princip skapar en illusion av ett icke-euklidiskt utrymme så förstör det inte den underliggande logiken i den euklidiska spelmotorn eller dess skript-API. En prestanda-analys utförs på slutresultatet (innan någon sorts optimering) som drar slutsatsen att metoden har en linjär tidskomplexitet (kostnaden av att rita portalerna växer linjärt med 3D-scenens komplexitet). Detta anses inte vara idealiskt för stora, komplexa spelmiljöer, men räcker för slutna, mindre komplexa spelmiljöer. Om systemet är optimerat och några av dess andra brister fixas eller förbättras, har det potential att vara en universell lösning för spel som behöver framställa icke-euklidiska koncept.

Acknowledgements

A special thanks goes out to my family for the direct and indirect support throughout this work.

Contents

Introduction 1

1 Related work **4**

1.1 Literature 4

1.2 Games 6

1.2.1 Antichamber 6

1.2.2 Portal 7

1.3 Analysis of the literature and game study 10

2 Methods **13**

2.1 Ray-tracing 13

2.1.1 What is ray-tracing? 13

2.1.2 Ray manipulation 14

2.2 Stencil masking 15

2.3 Portal system 22

3 Implementation **24**

3.1 Portal rendering 24

3.1.1 Initial portal setup 24

3.1.2 Dynamic portal view rendering 28

3.1.3 Remedying portal view distortion 38

3.1.4 Two-frame portal view delay 39

3.1.5 Near-clip plane adjustment 40

3.2 Seamless portal teleportation transition 42

3.2.1 The "PortalManager" class 42

3.2.2 Collision filtering 43

3.2.3 Teleportation logic 45

3.2.3.1 Player teleportation 46

3.2.3.2 Non-player teleportation 49

3.2.3.3 Cloning portal overlapping meshes 49

3.2.3.4 Resolving portal surface clipping issues 53

3.3 Seamless portal interaction 64

3.3.1 Carving holes in surfaces 64

3.3.2 Adding edge collision to the portals 67

3.3.3 Synchronizing physical interaction of base and clone objects 69

4 Results **72**

4.1 Performance analysis 72

4.1.1 Testbed and specifications 73

4.1.2 Identifying if CPU or GPU bound 74

4.1.3 GPU profiling 76

5 Discussion **78**

5.1 Drawbacks and limitations 78

5.2 Error sources 80

5.3 Further use-cases 80

6 Conclusion **82**

7 Future work **86**

Appendix **93**

A Project properties and Blueprint code 93

List of Figures

1.1 Antichamber cube demo 7

1.2 Portal mechanics demonstration 8

1.3 Portal recursion demonstration 9

1.4 Portal non-euclidean demonstration 10

2.1 Depth buffer illustration 16

2.2 Shader pre-computed shadows example 17

2.3 Stencil index illustration 18

2.4 Stencil desaturation example 19

2.5 Stencil “looking glass” 20

2.6 Stencil Antichamber cube example 21

2.7 “Portal” doorway demonstration 23

3.1 UV-mapping example 27

3.2 Basic portal structure 27

3.3 World space vs Local space illustration 29

3.4 Portal space illustration of a location conversion 30

3.5 Portal space illustration of a direction conversion 33

3.6 Portal space illustration of a rotation conversion 34

3.7 Converting location to portal space 35

3.8 Converting direction to portal space 35

3.9 Converting rotation to portal space 36

3.10 Converting transform to portal space 36

3.11 Portal camera movement logic 37

3.12 Portal view synchronization 37

3.13 Portal view distortion illustration 38

3.14 Portal object tick group setting 40

3.15 Camera view frustum illustration 41

3.16 "Near-clip plane fix" visual comparison 42

3.17 Collision preset example 45

3.18 Player camera collision shape issue 47

3.19 Demonstration of mesh overlapping teleportation problem (in-engine) 50

3.20 Demonstration of mesh overlapping teleportation problem (top-down graph) 50

3.21 Demonstration of mesh overlapping portal after cloning fix is applied 53

3.22 Portal surface clipping illustration 54

3.23 Portal “depth box” mesh illustration 55

3.24 “Portal Viewer” mesh attached to player camera 56

3.25 Portal depth box viewed from the front (in editor window) . . 57

3.26 Portal depth box viewed from the side (in editor window) . . . 58

3.27 Illustration of portal interior box mesh occlusion 60

3.28 New portal structure with depth box mesh 61

3.29 Initialization of portal scene capture component (portal camera) 63

3.30 Portal viewer dynamic material instantiation 63

3.31 Physics constraint example 1 65

3.32 Physics constraint example 2 66

3.33 Portal edge collision example 69

3.34 Cloned collision environment illustration 71

4.1 Unreal Engine 4 testbed scene 73

6.1 Example of non-euclidean space using portals 84

A.1 Portal surface shader 93

A.2 Portal surface shader properties 94

A.3 Run-time render target resolution setup 95

A.4 Near-clip plane project settings 95

A.5 Near-clip plane game logic fix 96

A.6 Portal camera near-clip plane properties 96

A.7 Portal surface collision settings 97

A.8 Player teleportation script 98

A.9 Non-player teleportation blueprint logic 98

A.10 Full “TeleportActor” function blueprint 98

A.11 Clone initialization script 99

A.12 Clone transform update script 99

A.13 Clone removal script 99

A.14 “M_PortalDepth” shader code 100

A.15 Portal interior box mesh properties 101

A.16 Portal surface overlap event code 101

A.17 Collision binder properties 102

A.18 Portal Begin Overlap CollisionDisabler code 102

A.19 Collision binder properties 102

A.20 Portal End Overlap CollisionDisabler code 103

A.21 “IgnoreActorWhenMoving” function utilization (begin overlap) 103

A.22 “IgnoreActorWhenMoving” function utilization (end overlap) . 104

List of Tables

3.1 Collision filtering 44

4.1 PC specifications 74

4.2 Thread frametimes (portals enabled) 75

4.3 Thread frametimes (portals disabled) 75

4.4 GPU frametimes 76

Introduction

The concept of euclidean geometry is an ancient concept that has shaped the science behind our world as we know it [5, 14]. It is a mathematical system that has laid the foundation and defines most of modern geometry. This mathematical system was first described by Greek mathematician Euclid (born around 300 B.C) who wrote a mathematical treatise which consisted of 13 volumes called "The Elements". "The Elements" is a compilation of mathematical definitions, propositions, postulates and proofs of earlier knowledge of geometry (of that time). "The Elements" is essentially a rigorously structured system of proofs of many mathematical concepts built up by earlier definitions and Euclid's own set of "common notions" (Euclid's own wording for "axioms") and postulates. He proposed five main postulates which would define a euclidean geometry (the geometry that accurately describes our reality). These are known as "Euclid's postulates" and are defined as follows [5, 14].

1. A straight line segment can be drawn joining any two points.
2. Any straight line segment can be extended indefinitely in a straight line.
3. Given any straight line segment, a circle can be drawn having the segment as radius and one endpoint as center.
4. All right angles are congruent.
5. If two lines are drawn which intersect a third in such a way that the sum of the inner angles on one side is less than two right angles, then the two lines inevitably must intersect each other on that side if extended far enough. This postulate is equivalent to what is known as the parallel postulate.

The first four postulates were proven by Euclid through logical deduction but he was unable to prove the fifth one. It was later proven by Italian mathematician Eugenio Beltrami in 1868 that the fifth postulate was not provable by the first four [2]. In 1846 Scottish mathematician John Playfair reduced Euclid's fifth postulate to a simpler form that instead said "In a plane, given a line and a point not on it, at most one line parallel to the given line can be drawn through the point."[1].

Any geometry that does not adhere to Euclid's fifth postulate (also known as the "Parallel postulate") or any other of his postulates is considered to be a non-euclidean geometry. The most well known non-euclidean geometries are "Elliptic geometry" and "Hyperbolic geometry". In three dimensions there exist a total of eight known non-euclidean geometries. The first two are Elliptic geometry and Hyperbolic geometry. Five of them are mixed geometries (part elliptic, part hyperbolic) and the last one is completely anisotropic (changes properties based on the direction/angle it being viewed from) [3]. This thesis will mainly focus on three-dimensional non-euclidean geometries.

The aim of the thesis is to see whether or not it is possible to efficiently visualise and replicate physical interaction of non-euclidean geometry and spaces (in 3D-space) in a modern, large-scale (euclidean) game engine, without modifying the game engine's source code. If it is possible, different methods of achieving it will be investigated. One of these methods will be picked and implemented as a dynamic game system in the game engine. The effectiveness of this method will be measured and its strength and weaknesses will be identified. The logic and algorithms used for the method should be able to (in theory) be transferred and implemented into any other game engine (that supports it). The resulting game system should aim to be efficient enough to use in (current) modern games at a reasonable framerate. It should also be as general as possible, being able to be applied to as many use-cases as possible (excluding virtual reality and ray tracing applications).

Methods utilizing "Real-time ray tracing" technology (such as NVIDIA's "RTX" technology) [7] will not be used due to performance inefficiency and because the hardware resources required for them are expensive. Only rasterization techniques will be used. Only existing, publicly available game engines will be used. This means that writing a game engine from the ground up will not be part of this thesis. Only existing, unmodified tools and frameworks will be used. The method used must strictly only use the scripting API provided by the game engine/game framework. No custom code allowed. The final resulting game system will be more akin to a prototype than a fully polished product. Minor issues are to be expected. The method used is primarily only meant to work for non-VR (virtual reality) applications (but virtual reality may be a discussed topic in the thesis). The resulting game system should not break any game logic (if possible).

The restriction of not being able to modify any source code and having to use an existing, publicly available game engine is intentional. The reasons are:

- Allowing unrestricted source code modification allows one to change a euclidean game engine into a non-euclidean one since the amount of source code one can edit is theoretically limitless. This compromises the goal and intention of the thesis.
- The implementation of the (potential) method should be as simple as possible and should be possible to implement into as many games as possible (in theory), regardless of the game framework/engine. Potential game developers that read this thesis should not have to dive into, or modify any of the source code in their game engine of choice. Some publicly available game engines do not even allow users access to the source code (such as "Unity"). The effort game developers should need to put into implementing the resulting game system should be minimal.

Since the game engine used is euclidean, its toolset assumes that the game world is euclidean and since the editing of source code is not allowed, the method used will not need to actually create real non-euclidean spaces that adhere to the rules of non-euclidean geometry. It is enough that the method creates spaces/geometry that appears non-euclidean from the players perspective (including physical interaction with these spaces). In other words, it is enough for the method used to create an illusion of non-euclidean space.

The game engine of choice for this thesis will be "Unreal Engine 4" [22] (specifically, version 4.22.3), as it is a widely available, free to use and easily accessible modern (euclidean) game engine.

1

Related work

1.1 Literature

The research and literature on this specific topic seems to be scarce. Some research papers that do however exist regarding this topic are “Immersive Visualization of the Classical Non-Euclidean Spaces using Real-Time Ray Tracing in VR” by Luiz Velho, Vinicius da Silva, and Tiago Novello [10] and “Ray Tracing in Non-Euclidean Spaces” by Joao Rodrigo, André Silva and Alves Silva [8].

The work of Velho (et al.) focuses on creating a framework which has the ability to visualize non-euclidean spaces for virtual reality applications using NVIDIA’s “RTX” real-time ray tracing technology. The intention is for their framework to be able to be used for art, games and virtual tours and even for multiplayer scenarios (the multiplayer application use is, however, never discussed further in the thesis). Their visualization framework is built upon NVIDIA’s existing “Falcor” real-time rendering framework [6]. Falcor utilizes real-time ray tracing and Microsoft’s DirectX 12 API [4]. Falcor is mainly used for research prototyping. A performance analysis is conducted and it concludes that the performance impact is small enough to not cause motion sickness in real-time virtual reality applications but is not optimal (the total frames per second are below the recommended refresh rate of the “HTC Vive” virtual reality headset that they are using for their research). The conclusion of the paper is that they have successfully reached their goal of creating a framework that can visualize various non-euclidean spaces and that the framework could be used to visualize these mathematical concepts in real-time. They also add that their research opens up the possibility of being used for video games and art.

The work of Rodrigo (et al.) focuses on creating a feasible general-use solution for visualizing generic three dimensional non-euclidean spaces, using real-time ray tracing. Some rasterization (non ray-tracing) rendering techniques are also discussed, but it is concluded that these techniques are only applicable for some types of non-euclidean spaces (hyperbolic non-euclidean spaces and spaces that are "mostly euclidean") and that there exists no general-use solution for using rasterization. The end result is them creating a ray-tracer and "tetrahedral modeler" that can visualize any non-euclidean space (flat and curved) where the ray tracer shoots out geodesic rays from the camera view point instead of "normal" rays, thus achieving their end goal of creating a general-use solution for visualizing non-euclidean spaces (using real-time ray tracing). It is also concluded that no general-use rasterization solution for visualizing non-euclidean spaces exists and thus, this is why they must rely on ray-tracing.

Both Velho's (et al.) and Rodrigo's (et al.) work focuses on rendering non-euclidean spaces using real-time ray tracing. This thesis forbids any method using ray tracing as it is very GPU (graphics card) intensive and instead looks for a more practical rasterization method that is less GPU intensive and can run games using it at a reasonable frame rate.

However, Rodrigo's (et al.) work mentions some rasterization techniques for rendering non-euclidean spaces but ultimately concludes that the existing rasterization techniques are scarce and each of them only apply for specific non-euclidean spaces. In other words, no general purpose technique was found. Thus, this thesis will focus instead on either finding a rasterization solution that works for some specific non-euclidean spaces or a solution that creates an illusion of non-euclidean space in a euclidean game world.

1.2 Games

Although the literature in the previous section (section 1.1) has a gloomy outlook on the possibilities of visualising non-euclidean spaces using normal rasterization rendering techniques, there do already exist a few games which explore some non-euclidean concepts that might be worth studying for some ideas or possible insight.

1.2.1 Antichamber

“Antichamber” (released in 2013) [11] is a mind-bending puzzle game set in an Escher-like non-euclidean world where the sole purpose of the game is to confuse the player and present them with puzzles that are solved in very unconventional ways. It demonstrates several non-euclidean concepts such as stretching/compressing spaces, infinite hallways, impossible shapes etc (see Figure 1.1).

The game was built using a modified version of Unreal Engine 3. In an interview with Epic Games, Antichamber’s sole developer Alexander Bruce said “It may not look like it’s utilizing the UDK to its full extent, but there’s more going on behind the scenes than people realize. Everything from how the world constantly changes and wraps around upon itself seamlessly, to the subtractive aesthetic and linework rendering has involved picking apart the engine and bending it in ways that weren’t intended. Unreal wasn’t specifically built for this kind of game, but no engine was.” (UDK is an abbreviation of “Unreal Development Kit” which is, a free standalone version of Unreal Engine 3) [12].

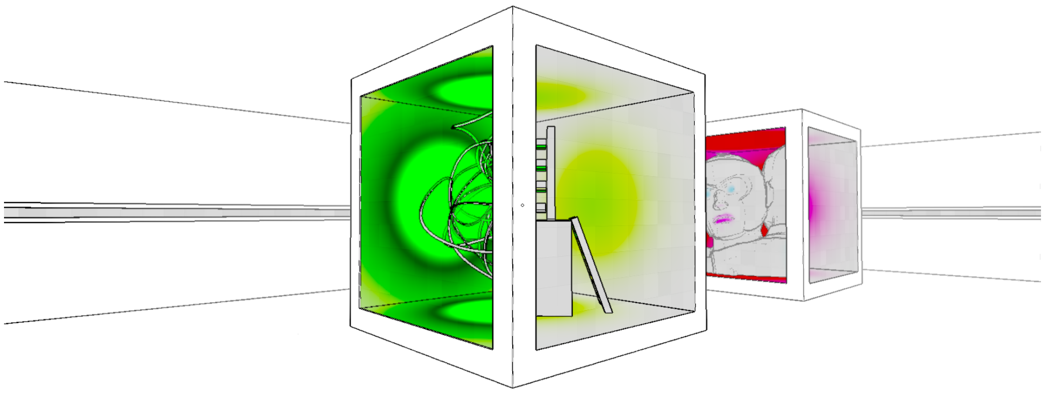


Figure 1.1: *An image from Antichamber. Notice how different spaces (and objects) exist in the same spot simultaneously in the hollow glass box. Each face of the cube shows a different space.*

1.2.2 Portal

“Portal” [20] is a puzzle game released in 2007 by the game developer Valve. The game’s main mechanic revolves around solving puzzles using a “portal gun”; a device which allows the player to place portals on any white surface in the game (almost every surface in the game is marked by either a black or white color). The surfaces of these portals render what’s on the other side in real-time (Portal A renders what Portal B “sees” and vice versa). Any object in the game (including the player) can seamlessly pass through any portal. Material objects are not the only things that can seamlessly pass from one portal to another, particles can too, for example, lasers (see Figure 1.2). The player (and any other object) can also stand in the middle of a portal opening and thus the object/player can exist in two places at the same time (see Figure 1.4). Every object that passes through a portal maintains its velocity magnitude but the object’s velocity direction might be different when the object comes out of the exit portal depending on how the exit portal is oriented. The portals can also recursively render each others views if one portal is seen through the view of another portal (see Figure 1.3).

Portal is built by Valve's own in-house game engine, the "Source" engine. The engine is not publicly available but you can contact Valve and strike a licensing deal with them to make a game using their engine (for an expensive upfront fee). Valve only allows using their engine for free if you use it to modify their own games and only if the content you make with it is non-commercial. The engine that is offered for free is however a heavily stripped down version of their full game engine, called the "Source SDK" [13, 32]. For these reasons, this thesis will not consider "Source" to be a publicly available game engine.

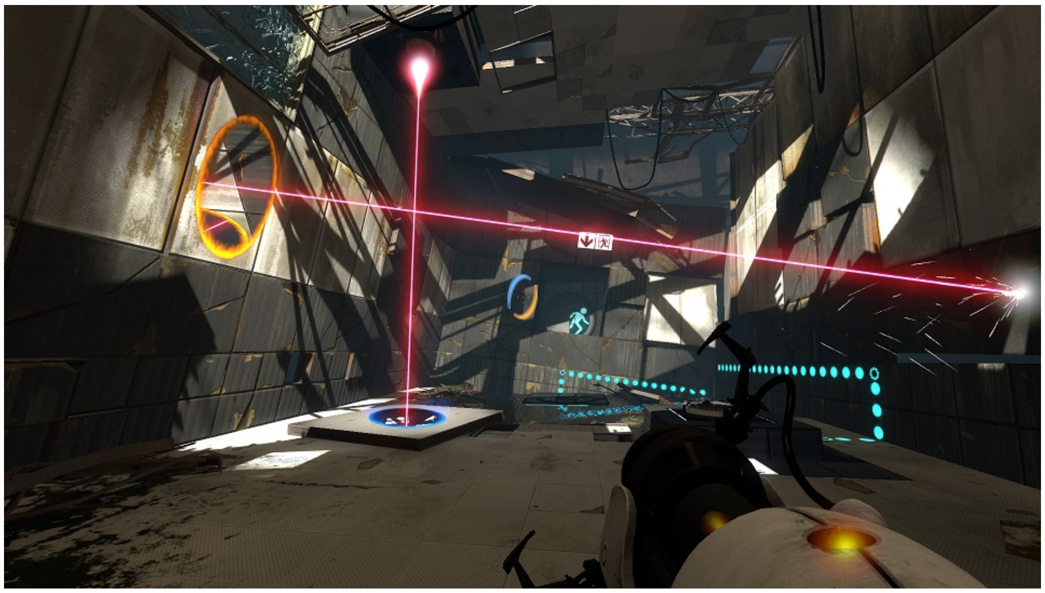


Figure 1.2: *An image from "Portal 2" (the sequel to Portal) demonstrating the portal laser mechanics. The same laser that goes into the blue portal comes out at an appropriate angle from the orange portal.*

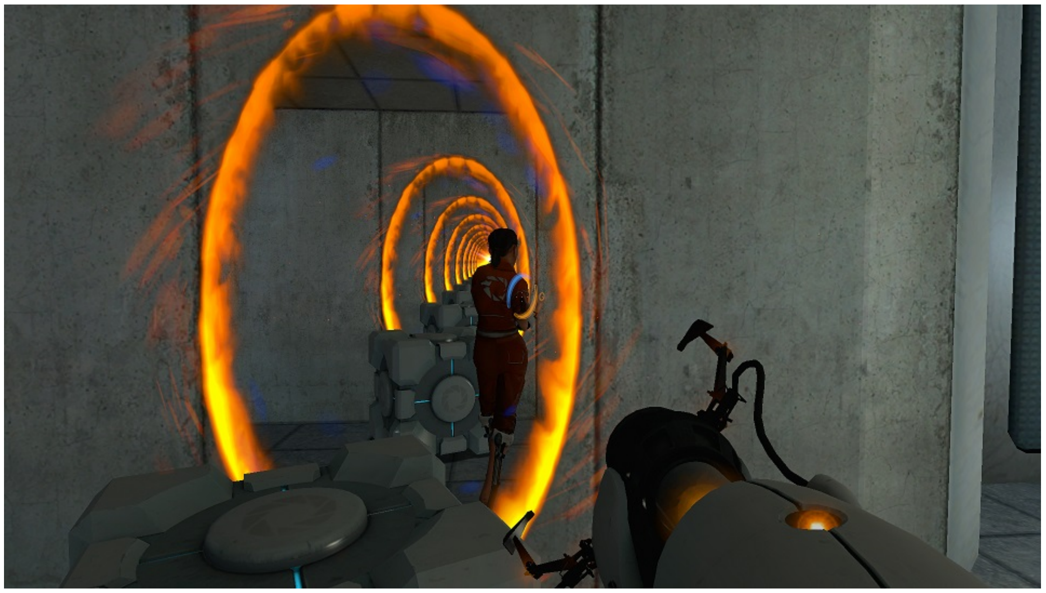


Figure 1.3: *An image from Portal demonstrating recursive portal rendering. The player character is looking at her own back (there is a blue portal placed behind the player, on another wall) as the portals recursively render each others views (because both of the portal's surfaces are facing each other).*

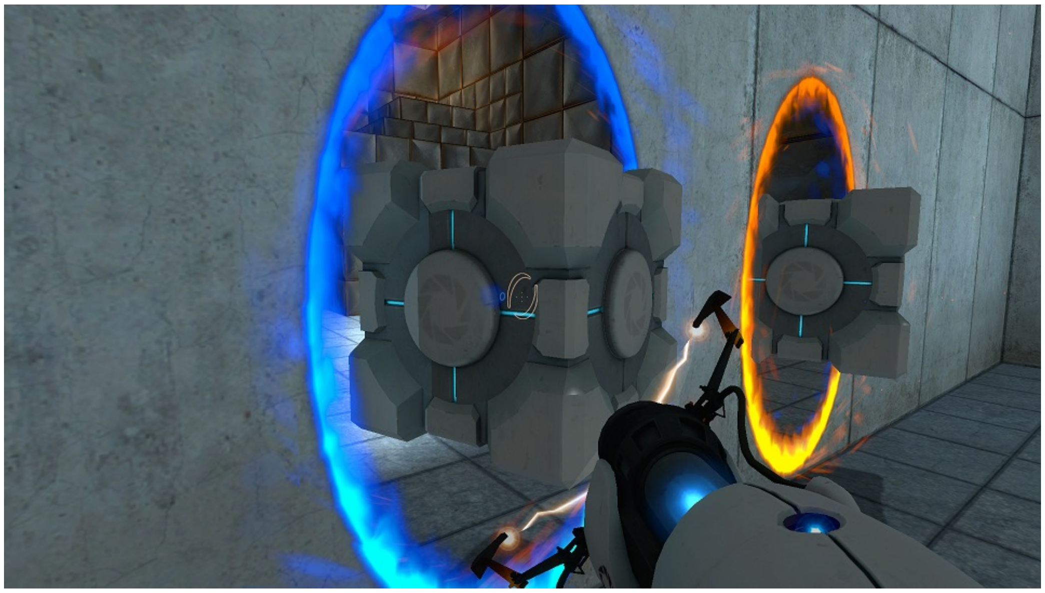


Figure 1.4: *An image from Portal demonstrating the non-euclidean nature of the portals. Half of the cube went into the blue portal, which caused that half to pop out of the orange portal. It looks as if the cube exists in two places at the same time.*

The portal system does, however, have some limitations. You cannot place more than two portals and the portals cannot be placed on moving surfaces (due to velocity ambiguity).

1.3 Analysis of the literature and game study

Velho's (et. al) research deals strictly with virtual reality applications and thus their solution might not be applicable to non-VR applications as virtual reality applications need to do the rendering once for each eye (total of two times). On top of that, depth is perceived differently on a curved virtual reality headset eye lens compared to a flat computer monitor. The thesis claims that the solution is also applicable for games, but the performance analysis of the thesis does not take any more complex 3D scenes into account (scenes that might represent the graphics workload of an average modern game). The performance was done in a fairly simple scene with a repeating mesh ("Mesh" is a technical term for a 3D model or a piece of geometry) in a black void. In other words, a 3D scene that is not particularly heavy on the graphics card (NVIDIA RTX 2080Ti), and despite running the performance analysis on a low complexity scene and having a top-of-line expensive graphics card, their solution still had trouble reaching optimal performance numbers (frames per second). Thus, the claim that their solution is usable for games should maybe be taken with a little bit of skepticism. Maybe it works on some very lightweight games.

Rodrigo's thesis does not really take any games into account since that thesis is more focused on providing a visualization method for scientists and researchers (mathematicians and physicists). The thesis also strictly uses "OpenGL" [16] for their visualization and custom ray tracer which is an outdated rendering API for modern games and thus highly impractical to use for modern games. Most games use "DirectX" or "Vulkan" [17] (successor of OpenGL) API's. The thesis overall leans heavily on theory rather than practice. There is thus no guarantee that the given solution in the thesis would be implementable into a modern game engine framework.

Both "Portal" and "Antichamber" play with non-euclidean ideas and both have seemingly slightly different approaches for doing so. Unfortunately, both developers of "Antichamber" and "Portal" have not released any sort of official documentation or technical paper for how they have achieved the mechanics and non-euclidean effects in their respective games. This is most likely due to them wanting to protect their intellectual property (including technology).

However, by observation, the game system in "Portal" seems like a more robust system for visualizing non-euclidean spaces as it allows for a wider range of physical interaction (most non-euclidean effects in "Antichamber" are simply visual and allow for minimal physical interaction, if any). The game itself is also mentioned as an example in Rodrigo's (et. al) thesis [8]. The most important aspect of the portal system, for non-euclidean space visualization, is that two (or more) portals allow two (or more) non-adjacent spaces to appear adjacent to each other in the game world. This allows, whomever uses or implements the portal system, to create an illusion of non-euclidean space in a euclidean space.

In any case, the observation of both of these games seems like a more promising lead than any of the literature mentioned above since the literature does not use game engines to achieve their goals, and instead use heavily modifiable rendering engines or outdated rendering API's. This does not take physical interaction or game logic into consideration.

On top of that, both of the literature pieces use ray-tracing to achieve their goals which is strictly forbidden by the restriction applied on this thesis.

2

Methods

In this chapter, different (possibly viable) methods of creating or faking (creating an illusion) non-euclidean geometry and spaces will be discussed. Each method has advantages and disadvantages. In the end of the chapter the most suitable method will be chosen for implementation.

2.1 Ray-tracing

While ray-tracing is explicitly not an allowed method to use in this thesis. It is still important to mention as it might be interesting for future readers. A future where the hardware might have gotten powerful enough to fully utilize (non-hybrid) ray-tracing in real-time applications.

2.1.1 What is ray-tracing?

Ray-tracing is a 3D-graphics rendering method that is used for rendering 3D-scenes by projecting the scene unto a 2D screen (as with many other rendering methods). Specifically, ray-tracing works by "shooting" a ray (3D vector), with a distance (could be close to infinite), from your "eye" (the scene camera) unto a 2D grid (in practice, this is your view-port/screen and the grid cells are the pixels of that screen). This ray extends into the 3D-scene until it intersects with an object. At the point of the intersection another ray is shot towards a light source in the scene. From this and the physical properties of the object hit, the light is estimated at the point of intersection which, in the end, estimates the color and luminosity of the pixel (grid cell) that was being shot through. If the ray that is shot towards the light source (from the point of intersection) hits something on its way to the light source (is occluded by something in its path), it is concluded that the hit piece of geometry is covered by a shadow (this is called a "Shadow ray"). A ray can also reflect off unto other objects (if the object hit is sufficiently reflective) or pass through transparent or semi-transparent objects, before traversing towards a light source. This process is repeated for each pixel on the screen.

While ray-tracing produces very accurate rendering results it is however, extremely expensive (performance-wise) even for the best modern hardware. It is not suitable for real-time applications, like video games, and is most often processed offline (non-real-time). Its most common use is for animated movies, CGI and high-quality model renders as these kinds of applications are not time critical, but often require a high level of fidelity.

The graphics card manufacturer NVIDIA has however, in 2018, released graphics cards, using their new "RTX" technology, that are partially capable of real-time ray-tracing. Unfortunately, these cards use something that NVIDIA calls "Hybrid rendering", which is a combination of rasterization and ray-tracing based rendering. In other words, these cards cannot render fully ray-traced scenes in real-time (at least at interactive rates) [9].

2.1.2 Ray manipulation

One can use ray-tracing to visualize non-euclidean spaces by manipulating the rays that are being shot into the scene, that enter a determined "non-euclidean space" in the scene, in various ways (compressing, stretching, changing the rays position etc.) . This will visually distort the geometry that that ray hits, potentially resulting in non-euclidean results [10, 8].

2.2 Stencil masking

Since full ray-tracing is too demanding for real-time video games. One has to rely on rasterization render techniques in order for it to be practical in a real game scenario. One possible technique that can be used to render non-euclidean spaces is the use of stencil masking, which makes use of the game engine's “stencil buffer”. The stencil buffer is a part of the “depth buffer”. The depth buffer is a render buffer (a buffer is just a temporary storage container of data) which stores data that helps the renderer determine the depth of each part of each object in the scene that is projected unto the screen. Parts of objects that have a greater depth than the parts of another object with lesser depth get occluded by the parts of the objects with lesser depth (see Figure 2.1). Specifically, this buffer is made up of a 2D array of integers, where each entry in the 2D array represents a screen pixel. If a pixel is “covered” by an object that is far away the pixel (2D array entry) gets assigned a high “depth value”. If a pixel is “covered” by a object that is close to the screen (camera), that pixel is assigned a low depth value. This “depth value” is commonly referred to as a “z-index”. A certain part of the depth buffer is referred to as the “stencil buffer”. The stencil buffer is used to render parts of certain objects while discarding others. This is similar to the depth buffer expect with the stencil buffer you decide which objects should be rendered and which should be discarded by manually assigning a index (compared to the depth buffer where the z-index is determined by the “depth” of a pixel) to a chosen subset of objects in the game world.

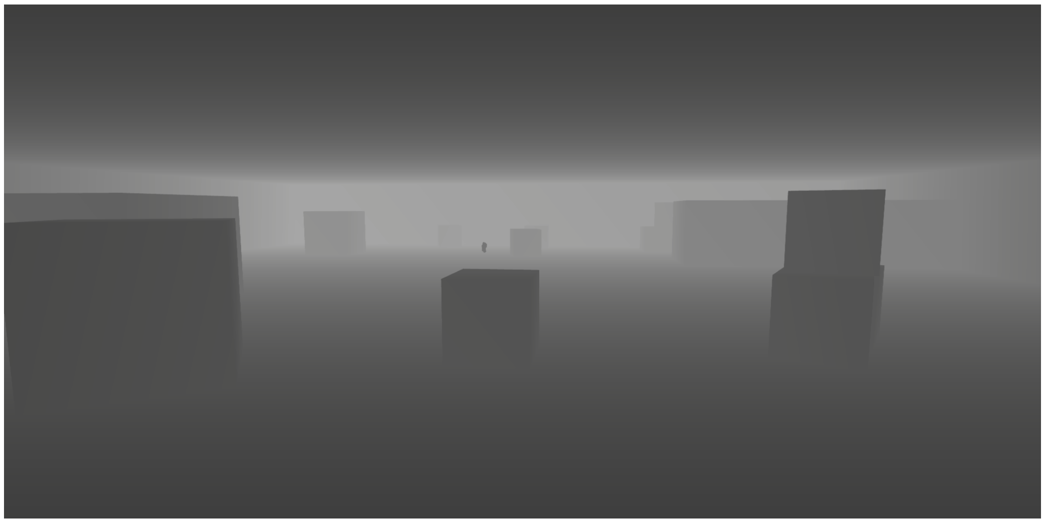
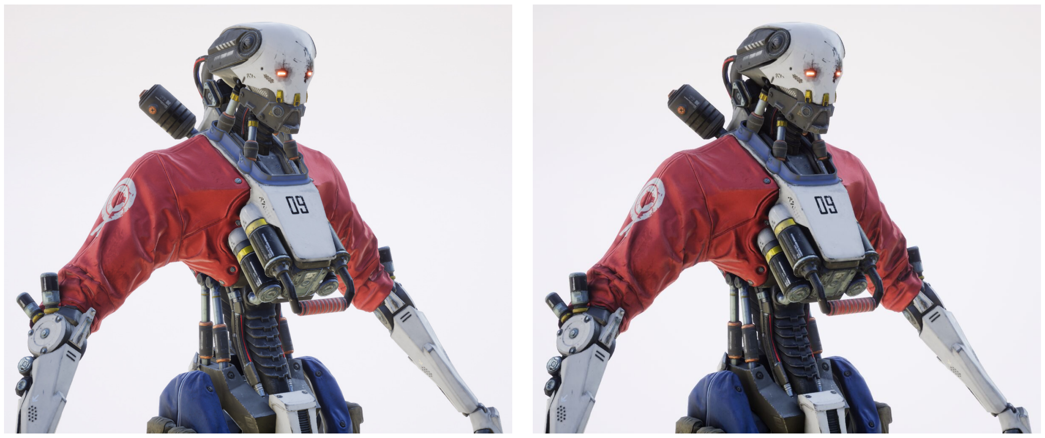


Figure 2.1: *Illustration of how the depth buffer in Unreal Engine 4 looks like when visualized. Notice how all parts of objects with a darker tone (parts of objects which are closer to the camera/screen) occlude parts of other objects which have a lighter tone (parts of objects that are further away from the camera/screen).*

You can use the stencil buffer to mask out a group (or multiple groups) of objects in a scene. Each object in a scene, that makes use of the stencil buffer, has a index or "stencil mask value" attached to it. This value tells a shader what group (a group is a subset of objects in the game world) that specific object belongs to.

A "shader" is a piece of code that determines how the graphics card should draw an object and what the the object should look like. A single shader can be reused and applied to multiple objects. In layman's terms, think of a shader as a layer of paint that coats an object and each paint bucket can be used to paint multiple objects. The paint is not restricted to a solid color, but can have multiple visual patterns and properties (glossy paint, metallic paint, matte paint, iridescent paint etc.). This only describes the very basic functionalities of a shader. More advanced shader code can do more complex things such as, determining how shadows and light interacts with an object, offsetting the position of individual vertices on an object, apply pre-computed shadows to objects (see Figure 2.2), faking 3D volume on a 2D object, cutting holes in geometry (see Figure 2.5) etc. Shaders can even change the look of an object dynamically based on outside influence. For example, the cut hole in a shaded object can expand/shrink based on the current distance between the shaded object and another object in the game world.



(a) *"Bent Normal" disabled* **(b)** *"Bent Normal" enabled*

Figure 2.2: *Example of how shadows can be pre-computed for certain areas of an object using a "Bent Normal" map (Unreal Engine 4 shader feature)*
Source: `https://docs.unrealengine.com/en-US/RenderingAndGraphics/Materials/BentNormalMaps/index.html`

Using this knowledge, a shader programmer can decide what to do with a specific group of objects based on the stencil value this group of objects have, essentially making this group of objects unique, separating them from the rest of the scene [19]. Stencil masking means that you choose what should be drawn within the outline of the objects which have a stencil value/index assigned to them. Each value can represent a different "look" (see Figure 2.3 where each objects stencil value is represented by a different color).

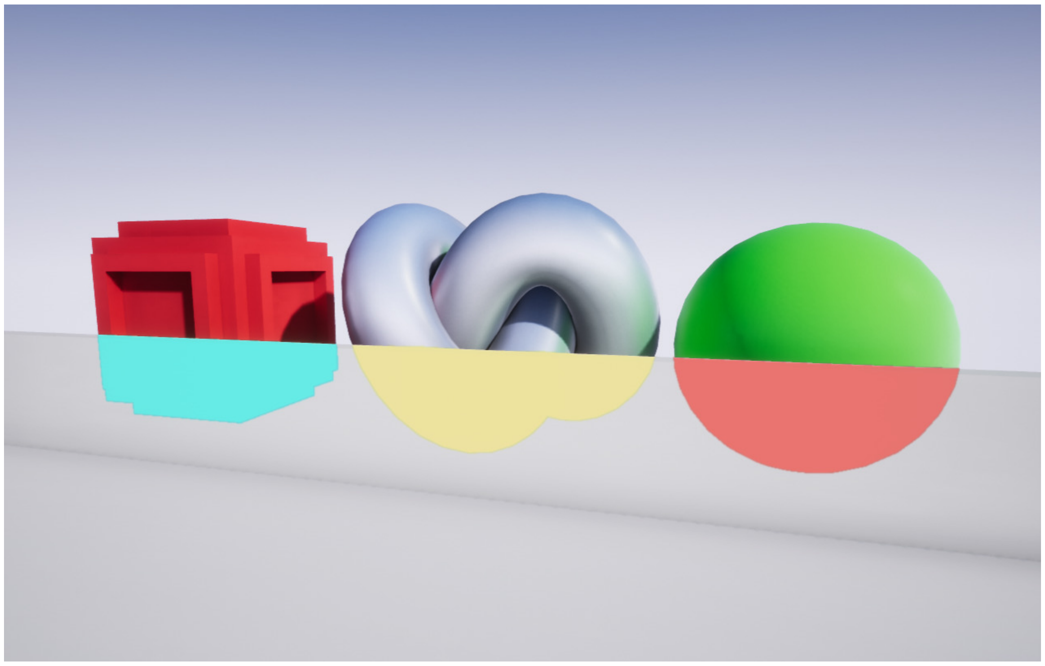


Figure 2.3: *Illustration of the stencil buffer. All three objects are using the stencil buffer. Each have their own separate stencil value (represented by the three different solid colors).*
Source: `https://docs.unrealengine.com/en-US/RenderingAndGraphics/PostProcessEffects/PostProcessMaterials/index.html`

Using this tool, you can apply some shader tricks to your scene. One simple example would be to desaturate your scene completely (black and white) but keep a few objects in your scene saturated (see Figure 2.4).

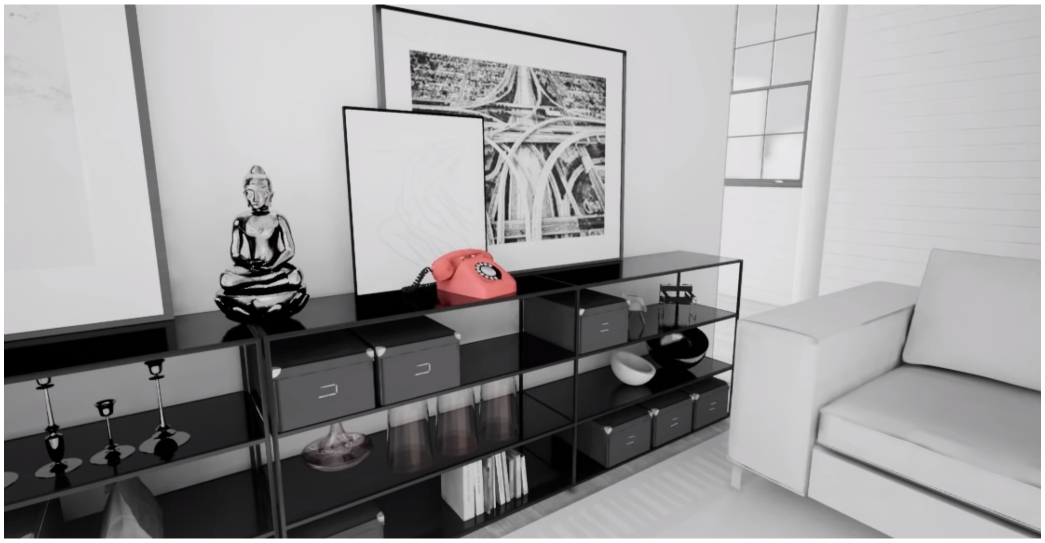
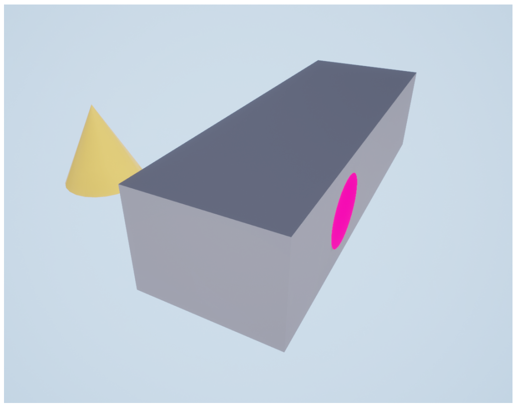
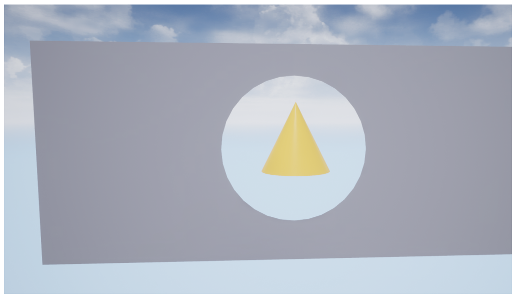


Figure 2.4: *Illustration of a black and white scene using the stencil buffer. Notice how the phone is still in color (saturated). It is because the phone is using the stencil buffer and has a separate stencil value from every other object.*
Source: `https://www.youtube.com/watch?v=PiQ_JLJKiOM`

Another, more effective, way to use this is to toggle the visibility of objects based on how they are being viewed. Using a transparent mesh, or a set of meshes, we can decide which stencil buffer objects can and cannot be seen when looking through these meshes using a stencil mask (regardless of other non-stencil objects that potentially obscure the view), creating this sort-of "looking glass" effect (see Figures 2.5 and 2.6).



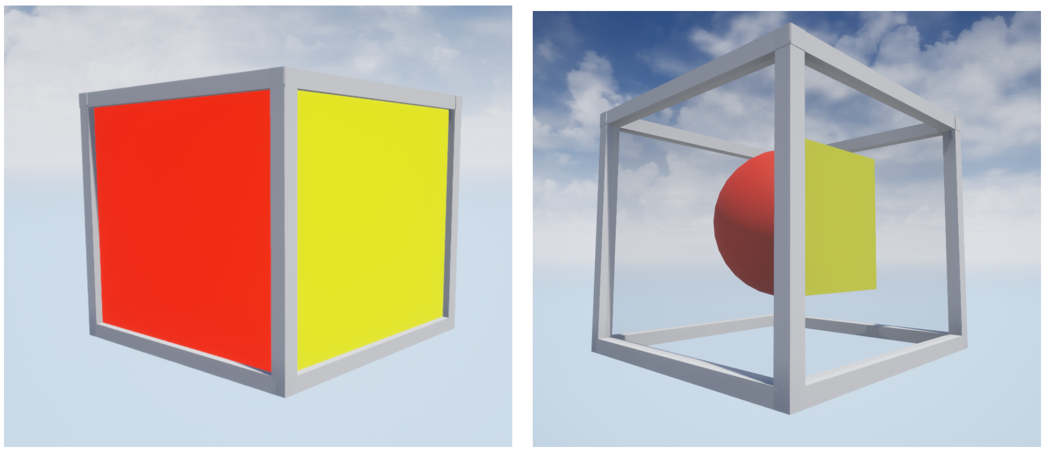
(a) *A pink circle mesh with a yellow cone mesh hiding behind a wall with the stencil buffer disabled.*



(b) *The yellow cone can be viewed through the wall when looking through the pink circle mesh if the stencil buffer is enabled, the pink circle mesh has a stencil value and a shader checking for stencil values is applied to the wall mesh.*

Figure 2.5: *Example of how stencil masking be used to see occluded geometry.*

A trick like this could be used to create the effect previously seen in Figure 1.1 (see Figure 2.6).



(a) *A hollow cube mesh with "looking glass" meshes (red and yellow faces) attached to each side. Each color (red and yellow) represent a separate stencil value.*

(b) *When the "looking glass" meshes in Figure 2.6a are made transparent, each face of the hollow cube can display a different mesh (yellow cube and red sphere) in the same world position within the cube. The stencil value of each mesh within the cube matches the stencil value of the cube face which they can be seen through (the red sphere has same stencil value as the red face and the yellow cube has same stencil value as the yellow face in Figure 2.6a).*

Figure 2.6: *Illustration of how the "Antichamber" cubes in Figure 1.1 could be created.*

The problem with this method is that it can only be used to subtract, add or replace existing geometry. It can only alter the visual appearance of the current space that the player is inhabiting. It cannot be used to change the topology of a space or connect non-adjacent spaces. On top of that it cannot be used to break any physical laws of euclidean geometry (for example, "a straight line is the shortest path between two points").

2.3 Portal system

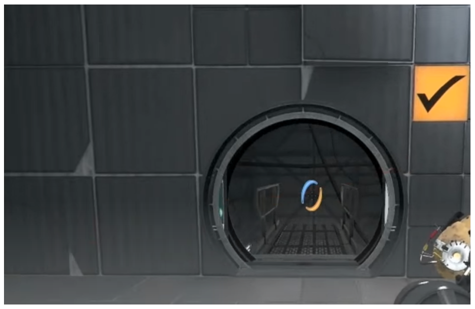
If there existed a method which could seamlessly connect non-adjacent spaces with (potentially) differing topologies, it could be used to create an illusion of non-euclidean space, which from the players perspective is no different than creating actual non-euclidean spaces (in a non-euclidean game engine). This is exactly what the portal system from Valve's Portal game (mentioned in section 1.2.2) is capable of, as long as the physical interactions between the two portals is replicated.

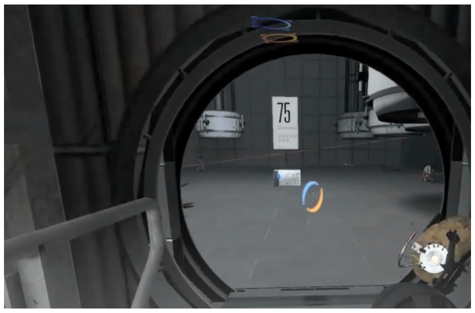
The three main challenges of creating a portal system that can create an illusion of non-euclidean space are

- Seamless rendering: The rendered image that is projected unto a portal's surface should be indistinguishable from the image rendered by the player's camera (the players view). In other words, if a player is staring at a portal surface, he/she should not be aware of the fact that they are staring at a portal, unless explicitly told so by the game.
- Seamless teleportation: If a player (that is unaware of the existence of portals in the game) steps through a portal and exits out the other side, they should not be aware of the fact that they just teleported (moved in space) or be made aware of the existence of the portals.
- Seamless physical interaction: If an object physically interacts with a portal (for example, a ball gets thrown through one) it should seamlessly go through to the other side while retaining all of its current physical attributes and values. Any physical interaction that interacts with an entrance portal with the intention of interacting with something "on the other side" should be replicated on the side of the exit portal.

In other words, the portals should be completely seamless and "pixel-perfect". Unless the portals have a visual indicator indicating their existence (visuals around the border, indicator on a mini-map etc.), the player should not be aware of their existence unless explicitly told so by the game. If any rules set by the above listed challenges are broken, the illusion (of a non-euclidean space) breaks. Maintaining this illusion is the highest priority.

A fully fledged portal system like this is the most versatile method for creating (or faking) non-euclidean spaces as the end result entirely depends on how the two connected (euclidean) spaces are built (topology etc.). This method will therefore be chosen for implementation in this thesis.


(a) *A normal doorway from VALVE's game "Portal 2".*



(b) *Player standing on the other side of the doorway looking into the room they were standing in in the previous image.*

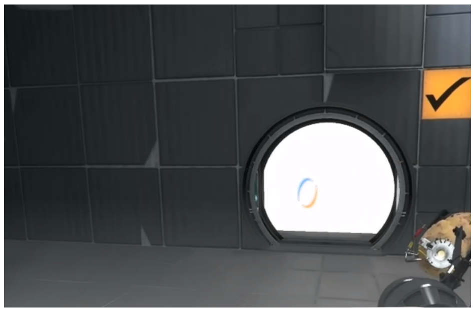

(c) *Tricked you! The doorway was a portal all along! This is what the doorway looks like with the portal rendering turned off.*

Figure 2.7: *Demonstration of the level of seamlessness a proper portal system should have.*

Source: Portal 2, Valve

The images in Figure 2.7 are a demonstration of the level of seamlessness the implementation of this method strives for in this thesis.

3

Implementation

This chapter will cover the implementation of the portal system. The thesis will be restricted to implementing only two portals (that are linked to each other). The game logic will be scripted using Unreal Engine 4's visual scripting solution called "Blueprints". The logic itself however, should be applicable to any sort of game engine.

3.1 Portal rendering

This section will focus on setting up the portals and figuring out a way to properly render the views of each portal so that each portal seamlessly displays what is on front of the other portal.

3.1.1 Initial portal setup

Before starting to implement the portals, one must first decide and understand how these portals are supposed to be represented in a game engine, as an object in physical form. Logically, one would assume that a portal in physical form is represented by some sort of surface, with an image projected unto that surface. How this surface is represented and shaped is mostly up to interpretation (flat, spherical, curved, one-sided, two-sided etc.). In this thesis, a portal surface will be represented by a flat plane mesh that is one-sided, meaning it is only rendered on one side and the other side is culled. "Culled" is a technical term in game development that refers to geometry not being drawn on screen (i.e not rendered).

At the initial stages of the implementation the portal only consists of a single flat plane mesh with no textures and nothing projected unto the plane mesh surface. So, at this stage the portal is, functionally no different than a static wall. In order to add some functionality, the portal must project some sort of rendered image unto its surface (the flat plane mesh). This projected image will represent the portal view (the view that the player sees when looking at a portal's surface). A portal's view is (logically) supposed to show what the other portal "sees" and is therefore showing a different perspective (the other portal's perspective). In order to render a second perspective in a game (in other words, a perspective that is not the player's perspective), one must add a second (non-player) camera.

Therefore, the next step is to add a camera component [25] to the portal class. This camera component should be a child component of the flat plane mesh which in itself is the portal object's root (see Figure 3.2). The root of an object in Unreal Engine 4 is the component which all other components are inherently attached to, as each object is composed from a hierarchy of components (just like the root hierarchy in a tree data structure). The camera component will allow the portal object to capture a new render target that is separate from the player camera's render target (render target in this context refers to the final render target output generated by the different render passes, that are captured by a camera component, in the rendering pipeline of Unreal Engine 4 [30]). This new render target data needs to be stored somewhere (each frame), otherwise it will be captured and discarded. In order to store a newly captured render target from a camera component, one must create a new render target asset (an "asset" refers to a local file in the Unreal Engine 4 project folder that stores data and these assets can be imported or created from scratch using Unreal Engine 4's built-in asset presets). The specific type of camera component used to store render target data into a render target asset (sometimes referred to as a "Render Texture" in other game engines, such as "Unity") is called a "SceneCapture2D" component [31] in Unreal Engine 4 [29].

A render target asset, in Unreal Engine 4, is actually a texture that stores and updates data on itself during run-time. How and what data is stored unto the texture is controlled by the game logic. The texture data can be completely arbitrary and does need need to be render target data captured by an in-game camera component. This texture is then (usually) projected unto a surface in the scene so that the player can interact with it and watch it visually update in real-time (this is the most common use-case but there are others). Render target assets are most commonly used to draw visualizations of game mechanics that need be updated in real-time, such as, drawing with a pen on a whiteboard, tracking the path a player is taking on an in-game map, drawing transparent bullet holes on walls or displaying footprints in snow and deforming that snow by updating the texture tessellation in real-time.

Therefore, a render target asset can be used to update the projection on the surface of the portal, by storing (and overwriting) a captured frame (render target) from a portal object's camera component ("SceneCapture2D" component) in the render target asset (texture), every frame update. Since we have two portals (two portal object instances) in total, let's call them "Portal A" and "Portal B", the image projected unto Portal A's surface (flat plane mesh) has to be captured by Portal B's camera component (SceneCapture2D component) and vice versa. This is logical because Portal A has to display what Portal B "sees" from Portal B's perspective (and vice versa). In addition to that, the shader that is applied to the portal surface must have its UV's mapped to the screen. This is to avoid the UV's wrapping around a mesh, as this can cause the projected texture to be stretched or compressed depending on the size of the mesh that it is applied to. "UV" is a 2D coordinate system with positions (u, v) where $u, v \in \mathbb{R}$, $0 \leq u \leq 1$ and $0 \leq v \leq 1$. It is used to denote coordinates on a 2D texture (image). "UV-mapping" is the process of properly mapping each (u, v) coordinate to a 3D (x, y, z) coordinate of a 3D object. Essentially, correctly "wrapping" the texture (image) around the 3D object (see Figure 3.1). It is important that the UV mapping of the render target projected unto the portal's surface remains the same regardless of the scale of the portal surface. The shader itself must also be unlit as light should not reflect of off the surface of the portal (as the portal should behave as a hole instead of an actual physical surface, such as a wall).

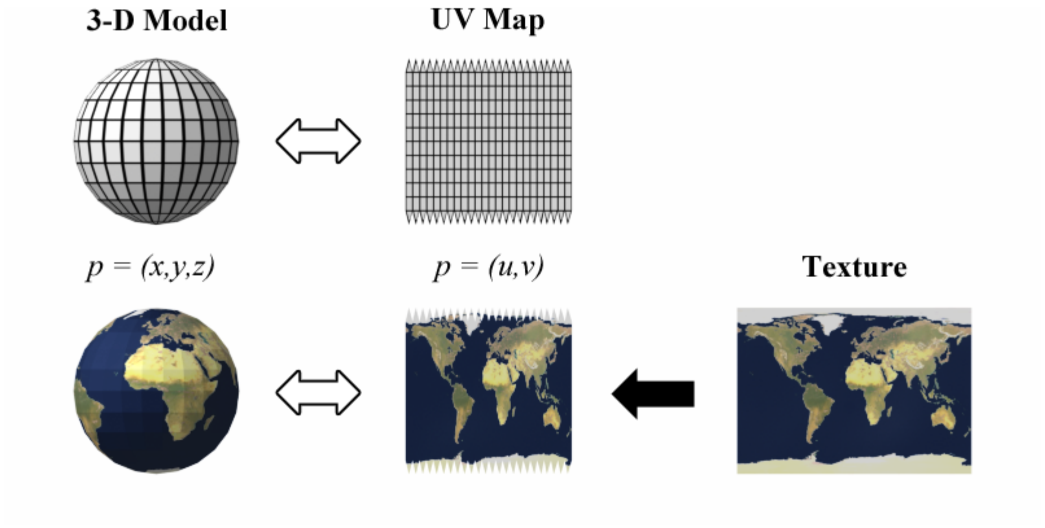
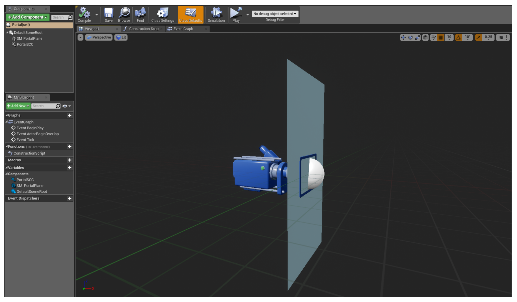
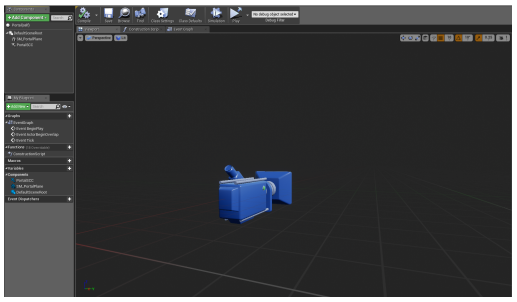


Figure 3.1: *Illustration of how a texture is UV-mapped unto a 3D object.*
Source: `https://commons.wikimedia.org/wiki/File:UVMapping.png`
Attribution: Tschmits, CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0), via Wikimedia Commons



(a) *Illustration of what the visuals of the portal looks like when viewed from the side after the initial setup stages.*



(b) *Illustration of what the visuals of the portal looks like when viewed from the back after the initial setup stages. Notice how the backside of the portal's surface (plane mesh) does not get rendered.*

Figure 3.2: *Illustration of what the basic structure portal class and visuals of the portal look like (right after the basic setup described in section 3.1.1).*

3.1.2 Dynamic portal view rendering

The portal views, as we have defined them, are completely static. The perspective of the portal view does not change depending on where the players "eyes" (camera) are relative to the portal (they currently work similarly to a static security camera monitor). A portal should work similarly to a doorway or a hole in a wall. When a person looks at a doorway, the perspective of what they see on the other side of the doorway changes depending on where they are standing relative to the doorway and the rotation of their head (or more specifically, where their eyes are in relation to the doorway). This sort of effect does not occur with the current portals. To fix this, some logic needs to be added to the portal system that dictates how both portal cameras should move and rotate relative to the player camera (the players perspective).

But before implementing a solution for that, one must understand the concept of "world space" and "local space". These two terms are commonly used in 3D graphics and game development. They refer to two different types of coordinate systems. "World space" refers to a coordinate system where the center of the 3D scene/game world is the origin of the coordinate system. "Local space" refers to a coordinate system where the center of an object in the scene, usually the pivot point of the object (in Unreal Engine 4 the origin point of an object is determined by the placement of its root component), is the origin point of the coordinate system. This includes the axes of the object [33].

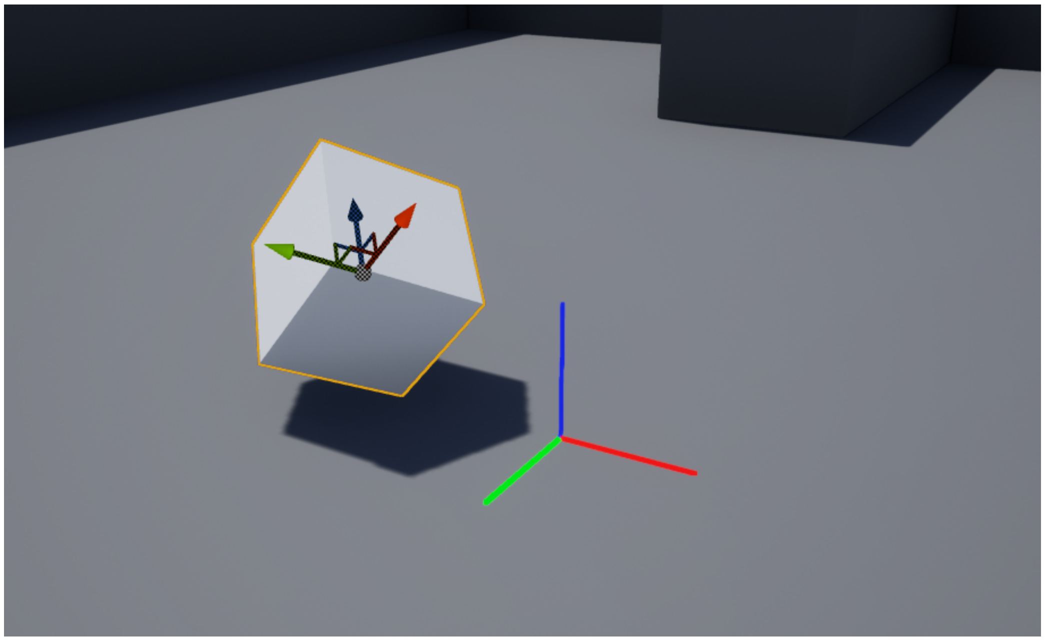


Figure 3.3: *Illustration of the differences between world space and local space, in Unreal Engine 4. The lines on the floor represent the coordinate system in of the game world (world space), while the three checkered vectors in the middle of the rotated cube represent the coordinate system of the cube in local space.*

This thesis will introduce a new term called “Portal space”. “Portal space” is not a coordinate system, but a “conversion system” relating a local transform of an entrance portal to a local transform of an exit portal (a transform is just a struct that holds the location, rotation and scale of an object). Also, keep in mind, from this point on, that the axes in Unreal Engine 4 do not represent the same “directions” as in most other tools and game engines (or in math). The X-axis in Unreal Engine 4 represents forward/backward direction, the Y-axis represents left/right direction and the Z-axis represents the up/down direction (in most other game engines, the Y-axis represents the up/down direction). Any mathematical expressions or code from this point on will assume the axes are defined as in Unreal Engine 4.

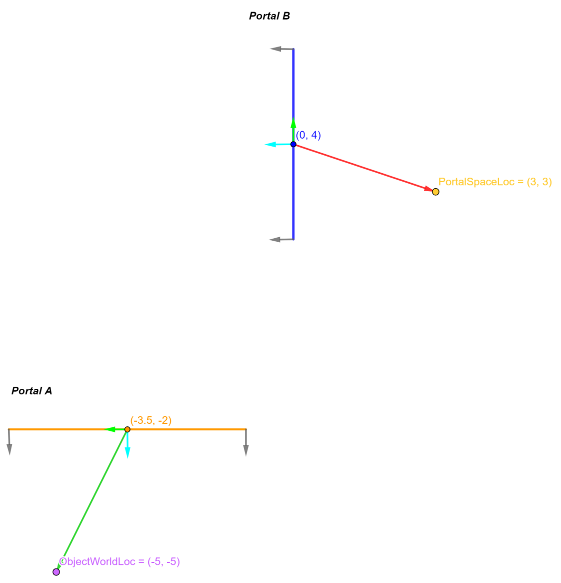


Figure 3.4: *This is an illustration of the location of an object in world space being converted to a new location in portal space. The purple dot is the world location of the object that needs to be converted. The orange dot is the world location of the portal that the object is in front of. The green vector going from portal A's world location to the object world location signifies the location of the object in the local space of portal A. The golden dot is the resulting (converted) portal space position of the object (world space). The red vector signifies the local space position of the resulting portal space location in the local space of portal B. The two small cyan and light green vectors are the local axes of each portal. If you "glued" the backside of portal B to the backside of portal A, the object world location and the converted portal space location should be in the exact same location (because two portals should act as a doorway between two spaces).*

The idea behind the object location to portal space conversion seen in Figure 3.4 is to use the dot product along with the local axes of both Portal A and Portal B and the location of the object, in the local space of Portal A. The object location in local space (of Portal A) is calculated by subtracting Portal A's world location A_P with the world location of the object O_P. This gives a vector, from Portal A to the object, that represents the local space location of the object in relation to Portal A (long green vector in Figure 3.4). Let's call this vector D. This vector now needs to be converted to the local space of Portal B. This can be done by utilizing the local axes of each portal. The local axes of the portals will be represented as a direction (unit vector). Let's call them A_F (local forward direction vector of portal A), A_R (local right direction vector of portal A), A_U (local up direction vector of portal A), B_F (local forward direction vector of portal B), B_R (local right direction vector of portal B) and B_U (local up direction vector of portal B). A_F, A_R, B_F and B_R are represented by the small light blue and green vectors of each portal in Figure 3.4. Let's call the world location of portal B B_P. In order to calculate a location that is behind portal B (such as in Figure 3.4), a vector must be added to B_P. This vector can be represented by the addition of portal B's "axis vectors" B_F, B_R and B_U. If the axis vectors of portal B get added up, the calculation produces a world location that is situated in front of portal B. This is wrong and is more akin to how a mirror would work. The portals need to behave as a doorway where each world location that is in front of portal A gets translated to a world location behind portal B and each world location that is to the right of portal A gets translated to a world location that is to the left of portal B (and vice versa). Thus, B_F (X-axis) and B_R (Y-axis) need to be inverted when performing the addition, like so $B_D = B_P + ((-B_F) + (-B_R) + B_U)$. Now the calculated world location is situated behind portal B. However, at this point in the implementation, the calculated world location behind portal B has no relation to the local space of portal A or the object world location. This is where vector D and dot products become useful. Using D and the dot product, each component of vector B_D can be scaled based on D and the local axes of portal A by calculating the dot product of D and every local axis of portal A. The size of the dot product will thus depend on the distance from portal A's plane to the object world location and the angle between D and portal A's plane (because A_F, A_R and A_U are unit vectors with a magnitude of one). Each dot product of each local axis of portal A can be multiplied by each component of B_D to extend B_D to the proper length and angle of D but in the local space of portal B. The final mathematical expression is as follows

$$D = O_P - A_P \tag{3.1}$$

$$C_P = B_P + (((D \cdot A_F) * (-B_F)) + ((D \cdot A_R) * (-B_R)) + ((D \cdot A_U) * B_U)) \tag{3.2}$$

where C_P is the final converted location (portal space).

Converting a direction (unit vector) in the local space of portal A to a direction in portal space (in the local space of portal B) follows roughly the same logic as the conversion of a object location to portal space. The only difference is that since a direction is being calculated (as opposed to a location), a unit vector is being calculated that does not need to have a start point or end point in world space (in other words, no locations or points in space are a factor in this calculation). Thus, the math expression for converting locations to portal space is modified to not include B_P (no origin point needed) and D is equal to the given direction (the direction that needs to be converted) since we no longer require the length of $D = O_P - A_P$ nor any of the locations (points in space) O_P and A_P (see Figure 3.5). Thus, the mathematical expression for converting a direction to portal space is

$$C_D = ((I_D \cdot A_F) * (-B_F)) + ((I_D \cdot A_R) * (-B_R)) + ((I_D \cdot A_U) * B_U) \quad (3.3)$$

where I_D is the given direction (vector) that needs to be converted to portal space and C_D is the converted direction (portal space).

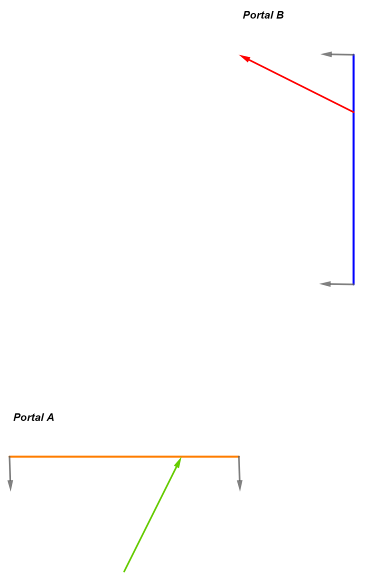


Figure 3.5: *This is an illustration of a direction (unit vector) being converted into a new direction in portal space. The green unit vector represents the direction needs to be converted (before conversion) and the red unit vector represents the converted direction in portal space.*

Converting the world rotation of an object into portal space can be done by utilizing the previous direction portal space conversion. An object's rotation can be represented by its three local axes (local X-axis, local Y-axis and local Z-axis) and these axes can be represented by a unit vector/direction (see Figure 3.6). Thus, to convert an object's rotation into portal space, one must take the three local axes of the object and convert each individual axis (direction) into portal space and then take those three converted axes to construct a new rotation (see Figure 3.9 for blueprint code).

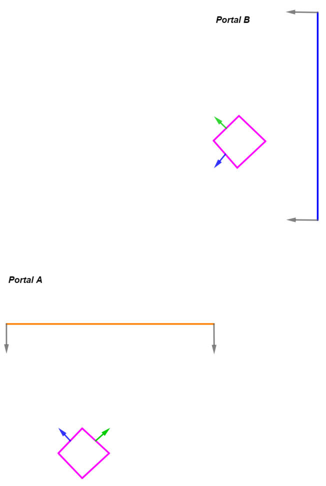


Figure 3.6: *This is an illustration of a rotation being converted into a new rotation in portal space. The green and blue vectors represent are the local axes of the pink box and represent its rotation. The location of where the pink box is placed in the figure does not matter for the conversion in any way (in other words, the boxes are placed near the portals in the figure just for demonstration purposes).*

Figures 3.7, 3.8, 3.9 and 3.10 show the logic (blueprint code) behind the implementation of the conversion system.

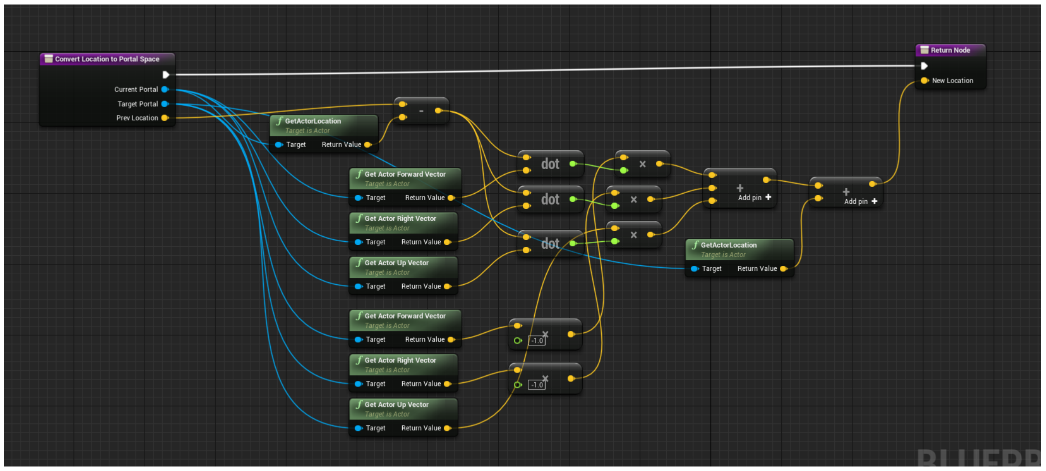


Figure 3.7: *This illustrates the logic for converting a world location to portal space (a visual illustration of this is shown in Figure 3.4). The yellow lines are vectors and the green lines are scalars.*

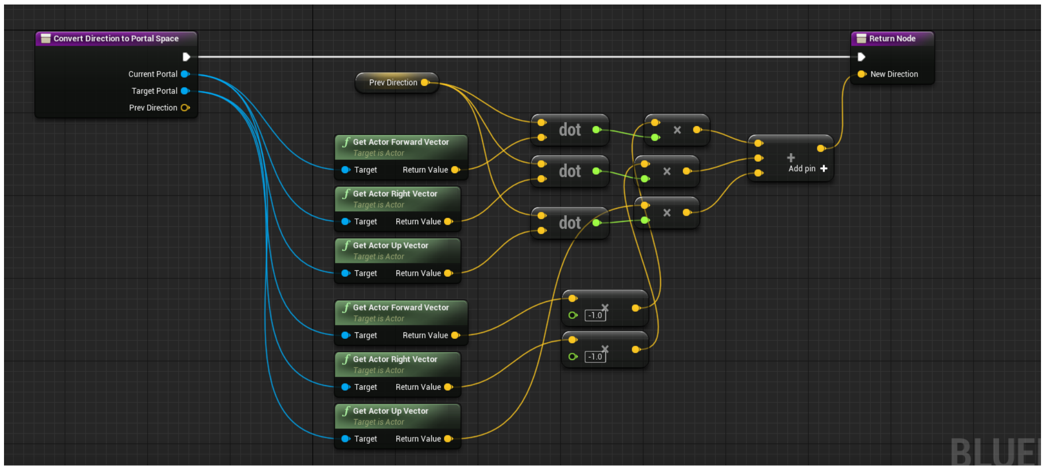


Figure 3.8: *This illustrates the logic for converting a direction to portal space (a visual illustration of this is shown in Figure 3.5). The yellow lines are vectors and the green lines are scalars.*

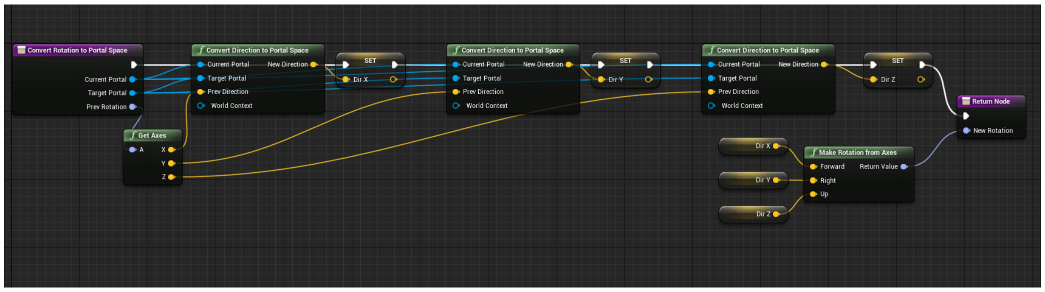


Figure 3.9: *This illustrates the logic for converting a rotation to portal space (a visual illustration of this is shown in Figure 3.6). The yellow lines are vectors, the green lines are scalars and the purple lines are rotations.*

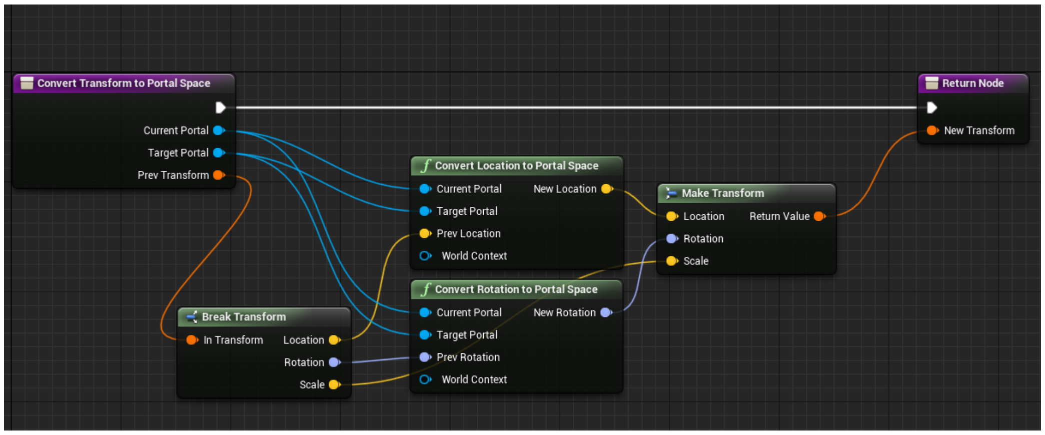


Figure 3.10: *This illustrates the logic for converting transforms to portal space. This is done by simply combining the defined functions in Figures 3.7, 3.8 and 3.9 to create a new transform.*

This conversion system of functions can now be used to construct a solution for making the portal views dynamic. This is done by making both portal cameras move and rotate relative to the player camera, using the conversion functions that are now defined.

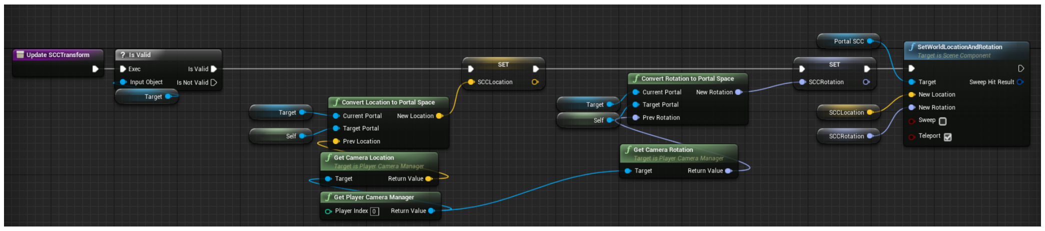


Figure 3.11: *This illustrates the logic for making the portal cameras move relative to the player's camera. This code/logic is contained within the portal class. "Self" refers to the portal object executing the logic (much like the keyword "this" in many other programming languages). "Target" refers to the portal that "this" portal is linked to. "PortalSCC" is a reference to "this" portal's camera component (its own camera).*

Now the portal views are not static anymore and its perspectives change based on the player's camera transform, much like how the perspective in a doorway changes when a person looking at it moves.

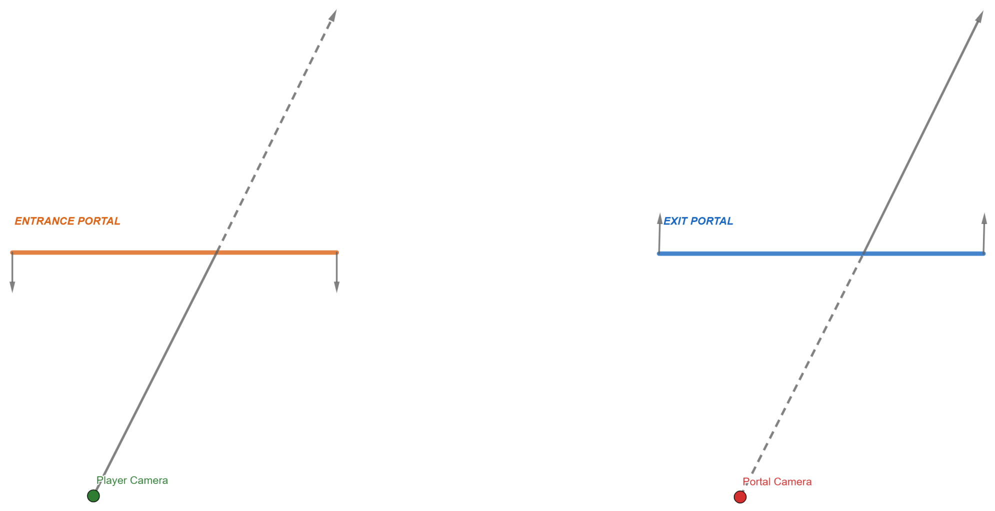
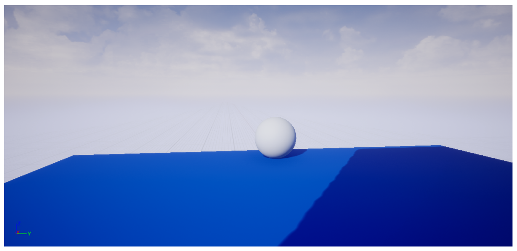


Figure 3.12: *This image illustrates how the player camera and the portal cameras are now in sync with each other. The "entrance portal" is the portal that the player character is looking at and intends to enter. The "exit portal" is the portal that the character will come out from upon entering the entrance portal.*

3.1.3 Remedying portal view distortion

After the implementation of the dynamic portal view rendering, the portal views can appear a bit distorted. This is because the portal camera and the player camera can have different field of views (FoV) during run-time. This statement can appear strange considering that both the player camera component and the portal camera component both have their respective "Field of View" setting set to Unreal Engine 4's default value of 90°, but it is important to keep in mind that the "field of view" setting in camera components only affect the horizontal field-of-view as Unreal Engine 4 uses a "Vert-" (vertical minus) field-of-view scaling method. In other words, the horizontal field-of-view is fixed, but the vertical field of view is not [23].

The vertical field-of-view changes depending on the aspect ratio of the game window screen (in-game screen resolution) and can therefore make the portal view appear squished or stretched (as the UV-mapping of our portal view is mapped to the screen instead of the portal mesh via the portal surface shader). This causes a distortion that ruins the illusion of connecting two non-adjacent spaces with these portals. The portal view can also appear blurry if the resolution of the render target is lowered sufficiently.


(a) *The view from the player cameras perspective.*

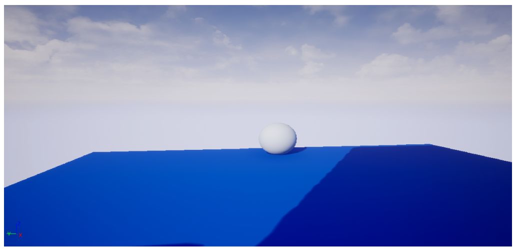
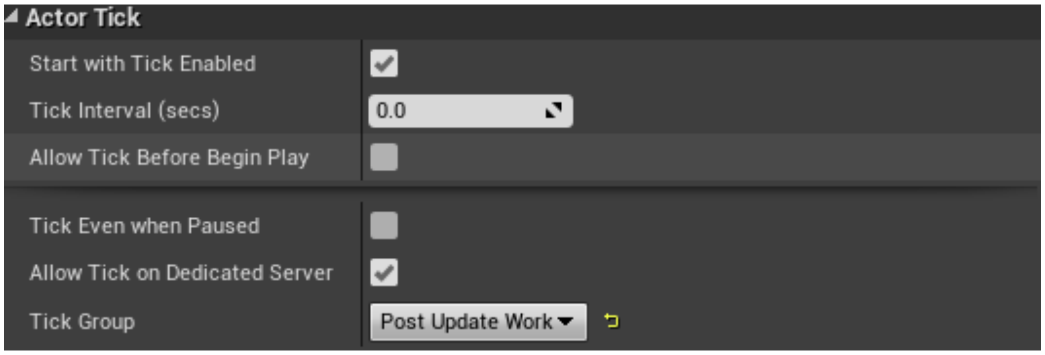

(b) *This image illustrates what the player sees when looking at a portal view that has the exit portal placed in the exact some position as in Figure 3.13a.*

Figure 3.13: *Illustration of the distortion in the portal view. Notice how the sphere appears slightly squished and farther away in the right image compared to the left image, despite both views being seen from the same angle and distance.*

In order to fix this issue, one restricts the portal camera's render target resolution to the resolution set by the game (in-game screen resolution). Now the field-of-view of both the player and portal camera match, and the visual parity between both views is much closer.

3.1.4 Two-frame portal view delay

At this point of the implementation the portal views look as expected if the player camera is not moving while looking at a portal view, but as soon as the camera moves, the player will notice that the portal view has a slight delay and is out of sync with the view of the player camera. The delay is a "two-frame" delay to be exact [18]. To fix this, one needs to make sure that the game logic that controls the movement of the portal cameras (see Figure 3.11) is executed after the portal cameras have been updated. This can be done by changing the tick group of the portal class to "PostUpdateWork" (see Figure 3.14). A "tick group" in Unreal Engine 4 is an enumerator that exists in every Unreal Engine 4 game object class (every class that can be instantiated into the game world as a "game object") that tells the engine when, during a frame, the game logic (code) of that class should be executed (for example, after the game physics have been calculated, before the game physics have been calculated, after the visuals have been rendered, before the visuals have been rendered etc.). The "PostUpdateWork" option tells the engine to execute the class's game logic after all other tasks/logic in the current frame have already been executed.



Actor Tick

Start with Tick Enabled	✓
Tick Interval (secs)	0.0
Allow Tick Before Begin Play	
Tick Even when Paused	
Allow Tick on Dedicated Server	✓
Tick Group	Post Update Work

Figure 3.14: *This is what the tick group settings of the portal object should look like. These settings can be found in the "Details" panel of the portal blueprint under the "Actor Tick" category.*

This ensures that all game logic (code) in the portal class gets executed after everything else gets executed (physics, animations, rendering etc.). The Unreal Engine 4 documentation itself recommends that all effects that rely on knowing where a camera is positioned or pointed should be placed in the "PostUpdateWork" tick group [24].

3.1.5 Near-clip plane adjustment

Any geometry that is placed between the portal camera and the portal surface (flat mesh plane) will be rendered in the portal view (keep in mind that the portal camera is always located behind the portal surface of an exit portal when a player looks at the surface of an entrance portal). This is problematic because the portal view should not view anything that exists behind the surface of an exit portal (see Figure 3.16a).

This occurs because the near-clip plane (see Figure 3.15) of the portal camera is disabled. To solve the problem, the near-clip global plane needs to be enabled (player camera's near-clip plane) in the project settings of Unreal Engine 4. Doing this will not solve the problem, but it will enable us to solve it, because once this option is enabled, it will not only allow the player cameras clip plane to be edited, but the near-clip plane for every camera in the project can now be enabled and have its position and direction modified.

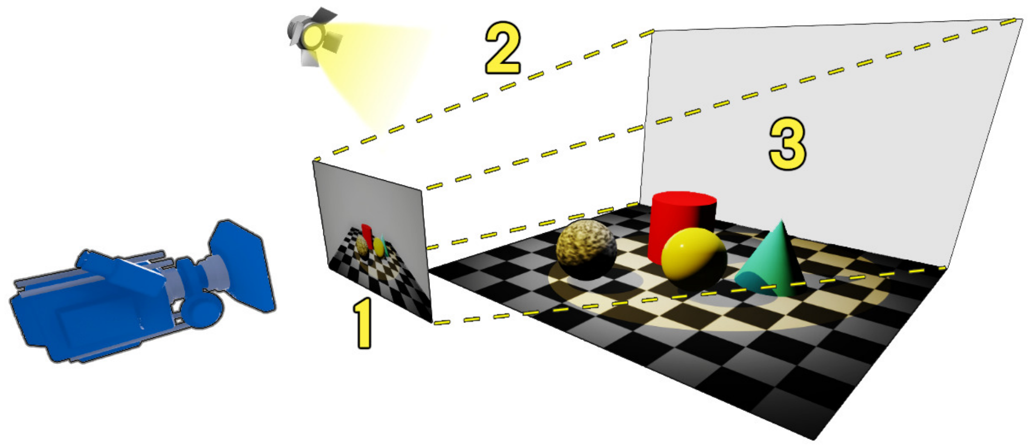
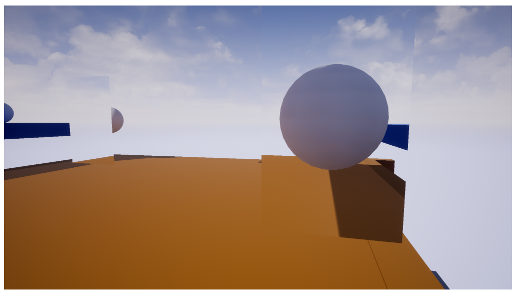


Figure 3.15: *Illustration of how a camera view frustum works in (most) 3D rendered applications. The number "1" represents the near-clip plane. The number "2" represents the camera frustum (dotted lines). The number "3" represents the far-clip plane. Everything that is within the camera frustum (pyramid-shaped space) gets shown and rendered in the camera view. Everything outside of the camera frustum does not. The near-clip plane, combined with the far-clip plane determine the size and distance of the camera frustum. If the near-clip plane is disabled, then it is equal to the near-clip plane distance to the camera being zero.*
Source:
`https://docs.unrealengine.com/en-US/RenderingAndGraphics/VisibilityCulling/index.html`

Once this is done, the near-clip planes for the portal cameras can be enabled (see Figure 3.15). Once the near-clip plane for the portal cameras is enabled, it will have a default local base position of $(0, 0, 0)$ (same position as the portal camera itself) and its normal surface vector set to $(0, 0, 1)$ (pointing in the direction of the portal camera's Z-axis). This is incorrect. The base position needs to match the position of the portal surface (flat plane mesh) and it should face the direction of the portal surface's normal. Since the portal position might be able to change during run-time, the clip plane position and direction needs to be updated continuously via in-game logic (see Figure A.5).


(a) *The sphere is seen in the view of the right portal despite the sphere being placed behind the left portal, creating a confusing and illogical perspective.*

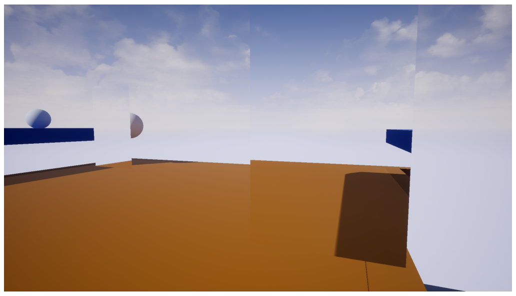

(b) *This is what the portal view should look like when the near-clip plane fix is applied. The portal camera is now correctly culling the sphere as it is no longer visible in the portal view.*

Figure 3.16: *A before/after comparison of the near-clip plane adjustment fix.*

3.2 Seamless portal teleportation transition

This section of the thesis will focus on the implementation of the teleportation logic for the portals and how to make the portal teleportation transition seamless (the player should not be able to notice that they've been teleported to a new space).

3.2.1 The "PortalManager" class

In order to properly implement the teleportation logic for the portals, a new class has to be introduced called "PortalManager". The purpose of this new class is to be able to manage each portal separately. The class should have member variables that store the reference to two portal object instances (this essentially links the two portals to each other). This allows for greater control over the execution of code, by now being able to identify which portal instance an object or player character is interacting with. For example, sometimes you might want to execute a function or set of logic on only one of the portals but not the other (based on some set of conditions). This would not be possible if you wrote the same set of logic within the portal class itself as it would execute that set of code logic on every instance of the portal class.

3.2.2 Collision filtering

The portal class needs to have its collision filtering properties set up before having its teleportation logic implemented. Collision filtering determines how each physical object type in a game should interact with every other physical object type in the game when they collide with each other (collision behaviour). Object types in Unreal Engine 4 are called "collision channels". Some examples of collision channels are

- WorldStatic: Objects that do not move (such as walls, pillars etc.).
- WorldDynamic: Kinematic objects that can move, either through code or animation (such as doors, elevators etc.).
- Pawn: Any player or AI controlled entity.
- PhysicsBody: Any object that moves due to physics simulations (bullets, soccer balls, ropes, vehicle suspension etc.)
- Vehicle: Vehicles.
- Destructable: Any destructable mesh.

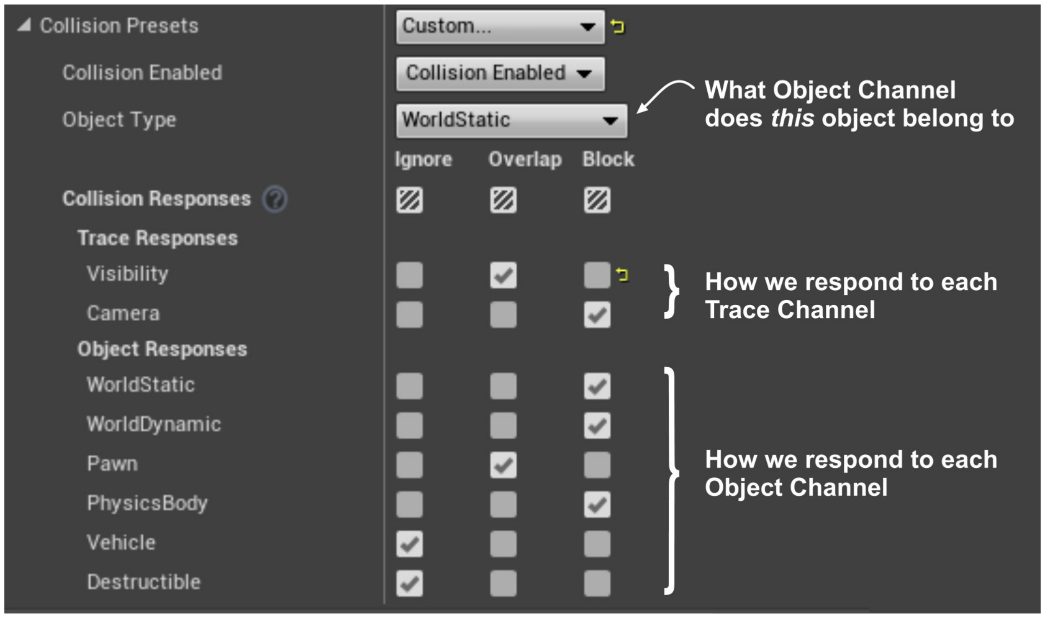
and each every object can have three different collision responses to each collision channel. These three are [26]

- Ignore: This object ignores objects that belong to that collision channel (phases through them upon overlap as if they did not exist).
- Overlap: This object overlaps objects that belong to that collision channel (same as "Ignore" but the overlap fires off a "Overlap Event" which can, in turn, execute a set of custom code).
- Block: Any object from that specific collision channel that touches this object and has a "Block" response setup for this object's collision

channel, will collide with it (this fires off a “Hit Event” which can execute custom code, just like the “Overlap” response).

Table 3.1: *This is a table of the resulting collision responses that can occur based on the different combinations of collision response settings between two objects (“Object A” and “Object B”).*

B \ A	**Ignore**	**Overlap**	**Block**
Ignore	Ignore	Ignore	Ignore
Overlap	Ignore	Overlap	Overlap
Block	Ignore	Overlap	Block


What Object Channel does *this* object belong to
How we respond to each Trace Channel
How we respond to each Object Channel

Collision Presets	Custom...
Collision Enabled	Collision Enabled
Object Type	WorldStatic

	Ignore	Overlap	Block
Collision Responses	▨	▨	▨
Trace Responses			
Visibility		✓	
Camera			✓
Object Responses			
WorldStatic			✓
WorldDynamic			✓
Pawn		✓	
PhysicsBody			✓
Vehicle	✓		
Destructible	✓		

Figure 3.17: *This is what the collision settings look like for an object in Unreal Engine 4.*
Source: `https://www.unrealengine.com/en-US/blog/collision-filtering`

The collision settings in Figure 3.17 say that the portal surface is seen as a static world object and that it only responds when objects from of the collision channel (object type) "WorldDynamic" and "Pawn" overlap with it and ignores the rest. The reason a "Block" response is chosen instead of "Overlap" is because you might want some objects in your game to not be able to go through a portal (these reasons are strictly for gameplay and have nothing to do with the portal system itself).

3.2.3 Teleportation logic

The teleportation logic consists of two parts. The first part is to track and check when an object/player should be teleported and the second part is to properly teleport the object/player to the other portal. The logic will be different depending on if the object to be teleported is the player character or not. It also needs to be made sure that one portal cannot teleport another portal.

3.2.3.1 Player teleportation

In order to set up the logic to teleport the player, it must first be decided when the player should be teleported (what the conditions should be) and what position (point in world space) on the player character should be tracked. The player should be teleported to Portal B when the center of the player's camera crosses the flat plane mesh (portal surface) of Portal A. This is logical because the player camera represents the player's point of view and is therefore a good point of reference. This seems fairly simple as all it would require would be to add a collision shape to the center of the camera and then check if the collision shape overlaps the collision shape of the portal surface. The problem with this approach is that every collision shape has a size (and shape) and therefore does not accurately represent the center point of the player camera. This will cause the point of collision (position) on the portal surface to never be equal to the center point of the player camera. It will always have a slight offset as the collision shape cannot be infinitely small. Even a slight error like this breaks the seamlessness of the portal transition (teleporation) as it will feel like the player "skips" a step upon teleportation. The length of this "skip" will be equal to the offset.

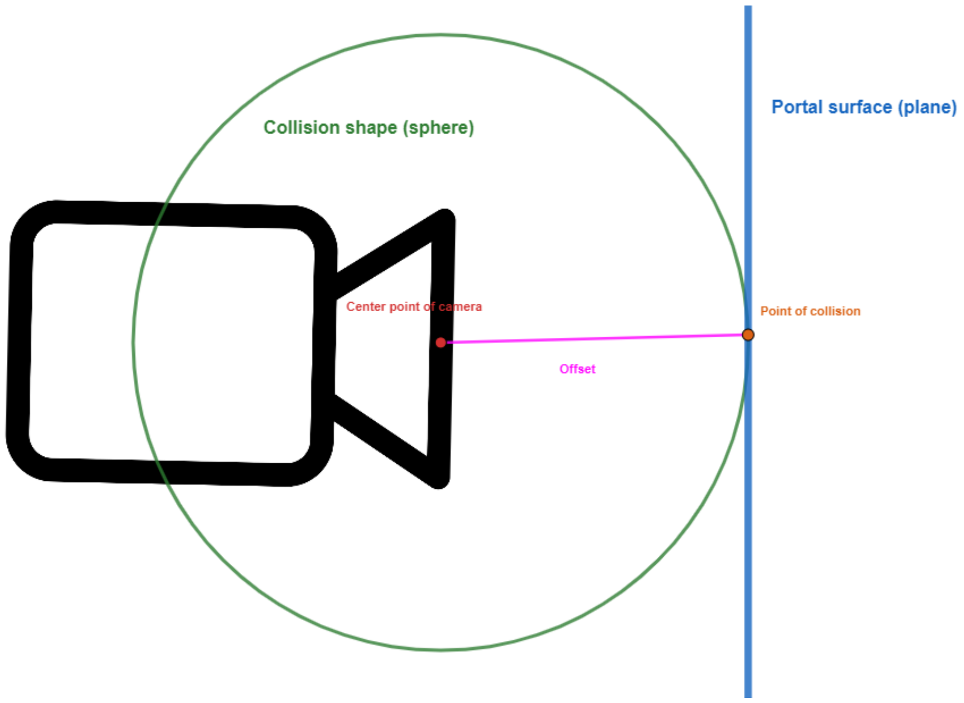


Figure 3.18: *This illustrates the issue of using a collision sphere to determine when the player's camera has crossed the portal's surface.*

Instead, the solution for checking if the player camera has crossed the portal surface must be done purely mathematically.

$$D_1 = ((C_{Pos} + C_{Vel} * \Delta F) - P_{Pos}) \cdot P_{Norm} \tag{3.4}$$
$$D_2 = C_{VelDir} \cdot P_{Norm} \tag{3.5}$$

C_{Pos} is the position (in world space) of the player camera (center-point). C_{Vel} is the velocity of the player character. ΔF is the frame delta-time. P_{Pos} is the position (in world space) of the portal surface. P_{Norm} is the normal vector of the portal surface. C_{VelDir} is the unit vector of C_{Vel} (velocity direction). $C_{Pos} + C_{Vel} * \Delta F$ is the player camera's position in the next frame.

Using the dot products D_1 and D_2, the conditions can be set up to know when to teleport the player. The player gets teleported to the other portal if (and only if) D_1 and D_2 both are negative. D_1 is negative only when the point $C_{Pos} + C_{Vel} * \Delta F$ is positioned behind the entrance portal's surface. In other words, if it is known that the value was positive in the previous frame (the point $C_{Pos} + C_{Vel} * \Delta F$ is positioned in front of the entrance portal's surface) and it is negative in the current frame, it can be ensured that the player has crossed and intersected the portal's plane. D_2 being negative ensures that the player is not crossing the portal back-to-front, but instead front-to-back. These calculations assume, however, that the portal plane's (surface) extent is infinite. Therefore, these conditions should only be checked while the player character is overlapping the portal's surface (this can be done via collision detection in the game engine).

Next step is to determine the logic for the teleportation itself. When teleporting the player, the player character's transform needs to be converted to the portal space of the target portal (the portal the player is teleporting to), then this new transform will become the new transform of the player character. After that, the player's velocity direction needs to be converted to the portal space of the target portal. Lastly, the player's control rotation needs to be converted to the portal space of the target portal.

"Control rotation" is a term unique to Unreal Engine. It is the rotation of the "player controller". The player controller component of an object is essentially the interface between the human player and the player character. The player controller handles all of the input from the players controls (mouse, keyboard, controller etc.) and then tells the player character how to move/what to do based on that input (essentially sending commands to the player character based on input). One could say that the player controller represents the "will" of the human player. Think of the player character as a puppet and the player controller being the puppeteer. For example, when a puppeteer raises one of his/her fingers (input), it causes the puppet to raise their arm (command). Using this analogy, the "control rotation" is essentially the direction of where the puppeteer (player controller) is looking (the rotation of their head) while the "actor/object rotation" is the rotation of the puppet (the player character) itself. The player controller is not tied to a single player character, but can detach and "posses" any other character, allowing the player to switch player characters during run-time (think of this as the puppeteer untying the strings of a puppet and then tying those strings unto another puppet). The "control rotation" in first person shooter games is often the same thing as the camera rotation since the player character follows the direction of where the camera is looking (technically, the camera is following the control rotation and the player character is following the direction of the camera).

3.2.3.2 Non-player teleportation

The teleportation logic for the player character has been implemented but a similar logic now needs to be applied for every other arbitrary (non-player) object to enable any object to teleport through portals.

First step (as before, in the player teleportation implementation) is to apply some logic for checking when the object should teleport. This logic is not much different from the previous player teleportation logic. One simply needs to replace the player camera position C_{Pos}, in equation 3.4, with the position of the arbitrary object in question and also replace the player velocity C_{Vel} with the velocity of the tracked object.

The teleportation logic (second step) itself is a bit different. It starts off the same as the previous player teleportation logic by transforming the object transform to the target portal's portal space, but then, instead of converting the player velocity direction and the player controller rotation to portal space, something different occurs. After the object transform gets converted to the portal space of the target portal, every component of the tracked object, that is simulating physics, will have its linear and angular velocity direction converted to portal space of the target portal and then have its current linear and angular velocity directions overwritten by the new converted velocity directions.

3.2.3.3 Cloning portal overlapping meshes

The teleportation of objects has been implemented, but the teleportation is still not seamless. This is because the parts of the object's mesh that are overlapping the entrance portal are not seen in the view of the exit portal (see Figure 3.19). This will look strange as half of the object's mesh will not be rendered in the other space (the space the exit portal resides in), as if it does not exist in this other space. This is because, just like in the real world, an object cannot have two positions (exist in two spaces) at the same time. It is not only physically impossible, but also impossible for a game engine to visualize and represent an object with two positions in world space.

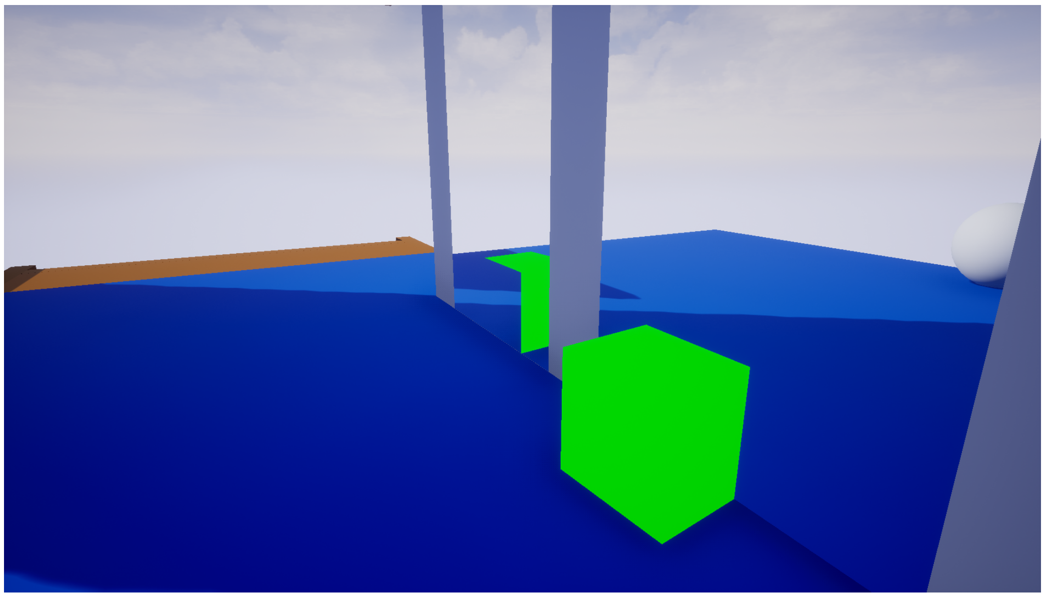
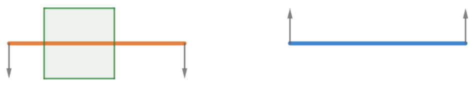
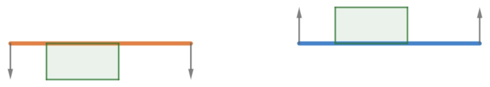


Figure 3.19: *This is a demonstration of the "mesh overlapping portal problem" described in section 3.2.3.3. Notice that the overlapping part of the mesh does not "poke out" of the left portal but instead gets cut off in the view of the left portal.*



(a) *This is the top down graph view of the problem visualized in Figure 3.19. This is what currently happens when a mesh overlaps a portal before being teleported.*



(b) *This is what it should look like.*

Figure 3.20: *Top down graph view of the overlapping mesh problem described in section 3.2.3.3.*

Perhaps the most obvious solution is to somehow cut the overlapping mesh at the intersecting line of the portal's surface, split the mesh into two meshes, move that second piece of the once unified mesh to the corresponding position (in portal space) of the exit portal and then finally combine the two meshes into a unified mesh again once the object stops overlapping the portal. This is a good starting point but it creates a big problem. It is very performance heavy. Cutting up and modifying meshes during run-time is a costly process and on top of that, this cutting process would have to be repeated every frame update (or at least every time the object moves). That is simply not feasible for an application that needs to run in real-time.

Instead, a clone of the current overlapping object can be instantiated into the game world and placed at a position and rotation relative to the relative position and rotation of the base object to the overlapped portal (portal space). After the clone object has been placed, the clone simply has to update its position and rotation to match the relative position and rotation of the base object (kind of like a server-client relationship, where the base object represents the server and the clone represents the client, if familiar with networking). While instantiating is relatively expensive compared to other game related functions, it is still far cheaper than real-time mesh manipulation. On top of that, the instantiation ("heavy" operation) only has to occur once (when the object begins overlapping the portal), while the other mesh manipulation solution has to cut the mesh each frame update. This solution only has to update the position and rotation of an object every frame, which is infinitely cheaper in comparison.

The actual implementation of this goes as follows: A map (dictionary) variable is created within the portal class, called "CloneMap". This map is responsible for linking each base object instance that is overlapping the portal to their corresponding clone. No base object instance can have more than one clone each (one-to-one relationship). This map is crucial for knowing which clones belong to which base object instance when (later) updating the transform of, or destroying the clones. Both the key and value type of the map should be an object instance reference.

Next, a function for initializing the clones is created. This function is called when an object overlaps the portal's surface. The function takes the overlapping object instance, gets the class of that overlapping object instance and using that class ID spawns another object instance of that class with the same scale as the base object instance. When the clone instance is first spawned (initialized) it should not interact with the environment or be able to be seen. Therefore, the location of the clone instance should be a value in world space where the clone instance cannot be seen as interacted with. It will be moved and handled properly after initialization. The initialization is just used to load it into memory and configure the proper properties for the instance. The initial rotation value is not important and can be set to anything. After the clone instance has been spawned it is made temporarily invisible (to avoid the player from initially seeing it spawn into the world) and its collision is disabled (the clone instance should not physically interact with the environment and should act like a phantom version of the base object as it is purely a visual instance of the base object instance). After that, the simulation of physics is disabled for every component of the clone instance (to avoid "outside forces" affecting its movement as its movement should only be affected by code, in other words, the movement of the base object instance) and the ability for the clone instance to generate overlap events is disabled (to avoid it interacting with the environment and to avoid it recursively creating clones of itself as it will later, inevitably, overlap the exit portal). The final step in the function is to add a reference to the base object instance and a reference to the clone instance as a new entry in the map variable (the base object instance is the key and the newly spawned clone instance is the value) and then finally make the clone instance visible again.

Next, a function responsible for continuously updating the transform of the clone instance is created. This function works by iterating through every key (base object instance reference) in the map and in each iteration, the base object transform is acquired (through the key's object reference). The transform of the base object is then used to convert the transform to the portal space of the target portal (the portal which the clone instance will overlap). At the end of the current iteration, the transform of the clone instance is set to the newly converted transform. This function is called every frame update.

The final function that needs to be implemented is a function responsible for properly destroying the clone instances once the base object instance stops overlapping the entrance portal. It is called when a base object instance stops overlapping a portal's surface (and takes the reference of that base object instance as an input parameter). Using the acquired base object instance reference as a map key, it finds the corresponding map value (clone instance reference) in the clone map. Once the reference from the value is acquired, the reference is used to destroy the corresponding clone instance and remove it from the clone map.

After all three functions have been implemented and called, object's that overlap a portal will appear to exist in two places at the same time, fixing the initial problem (see Figure 3.21). The same property that exists in the game Portal, shown in Figure 1.4.

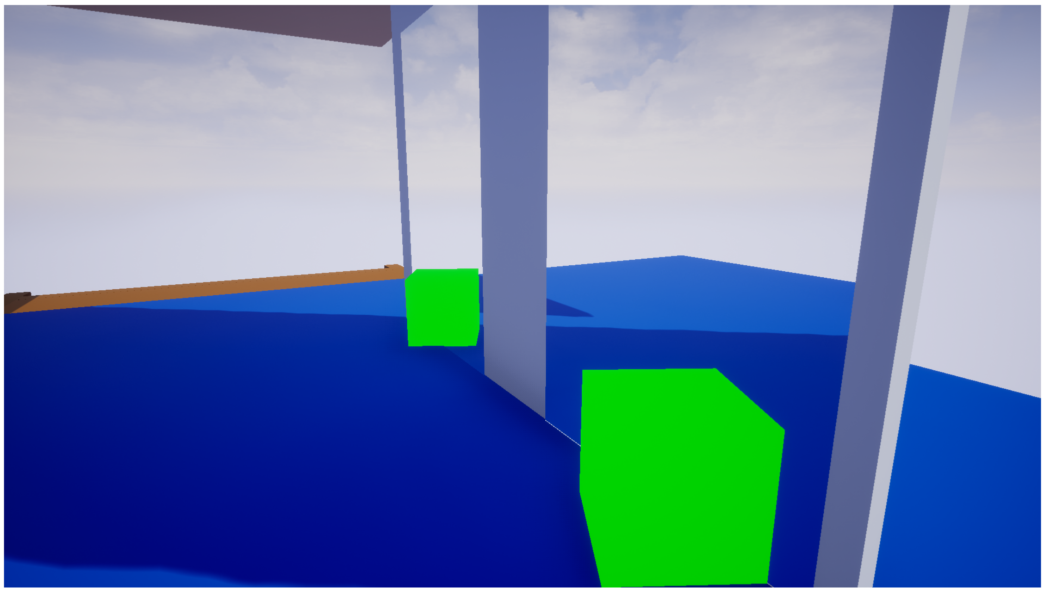


Figure 3.21: *This is what portal overlapping meshes look like when the final fix for the problem described in section 3.2.3.3 is applied.*

3.2.3.4 Resolving portal surface clipping issues

At this point in the implementation, when the player character attempts to cross from one portal to another, they will notice a slight visual "glitch" appear. For a split second, a white blank screen will flash on the screen once the player character is about half-way through the portal transition (see Figure 3.22). This is a problem that completely ruins the illusion. This is happening is because the player's camera is clipping the portal's surface mesh when walking through a portal (remember that the portal surface is just a flat plane mesh). Lowering the near-clip plane distance of the player camera to zero unfortunately does not resolve the issue and an alternative solution needs to be applied.

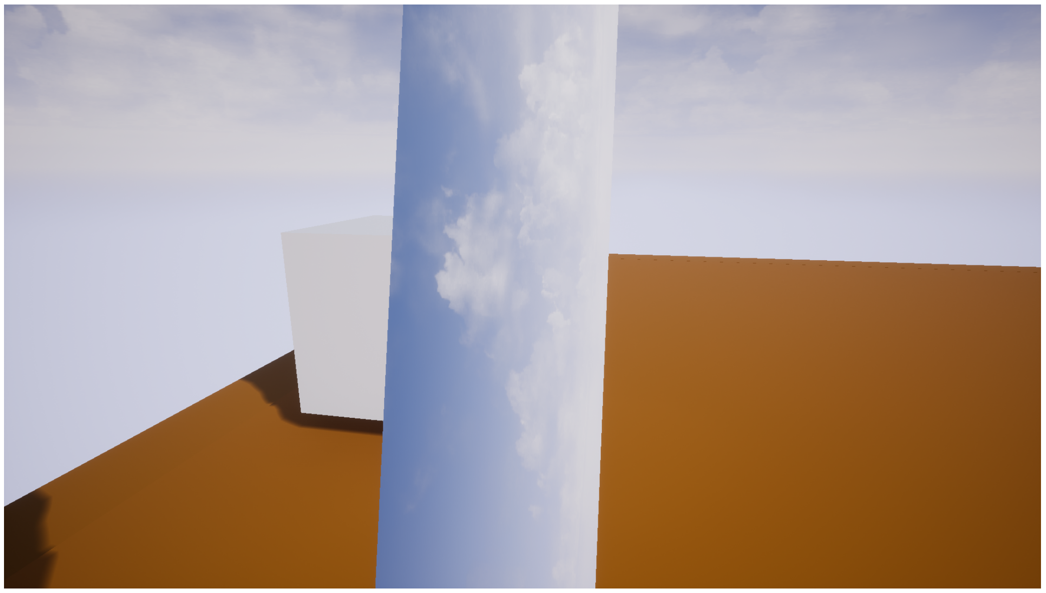
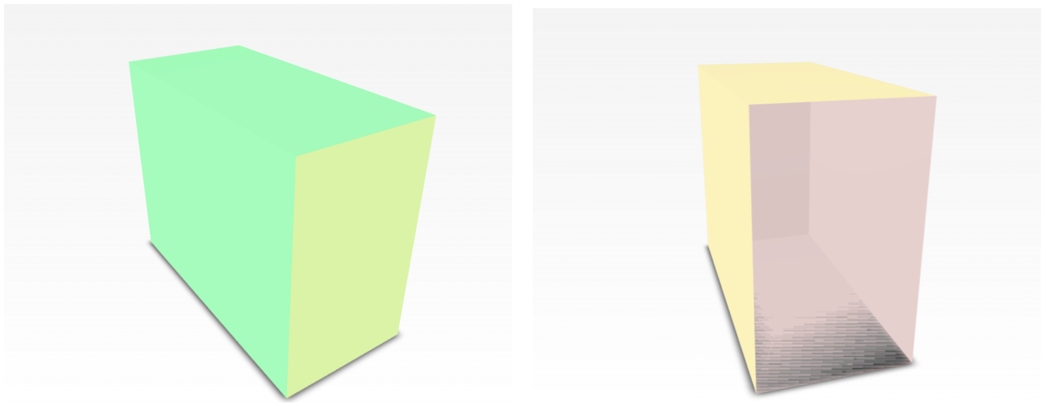


Figure 3.22: *Illustration of the camera clipping issue. The blue sky in the middle is the view of the portal. The player is standing half-way in the portal (walking in sideways) when the camera clips the portal surface mesh, allowing the player to see the white cube behind the portal, ruining the illusion.*

The solution to this problem is quite unorthodox and is not a quick and simple solution. The first part of the solution to this is to create three separate meshes in a modeling software (Blender [15] is used for this thesis). These meshes are

- Portal Depth Box Exterior: A open, hollow, 5-faced box with the opening being the size of the portal's surface. This box mesh should have backface culling enabled (in other words, only one side of each face should be rendered). This will cause the insides of the box to not be rendered (see Figure 3.23a).

- Portal Depth Box Interior: The “Portal Depth Box Exterior” mesh but with its normals inverted, placed in the exact same position as the “Outside Portal Depth Box” in world space. This mesh should also have backside culling enabled. Combining both meshes should create a “complete” open, hollow, 5-faced box with both the interior and exterior being rendered (see Figure 3.23b).
- Portal Plane: A flat rectangular plane (can be square too) that represents the portal’s surface. It should be placed in the opening of the depth box, closing it (see Figure 3.23a).



(a) *The complete depth box with all three separate meshes combined (the interior mesh cannot be seen due to the portal plane mesh blocking the view). The green mesh is the exterior mesh and the lime green mesh is the portal plane mesh.*

(b) *Same combined mesh as in Figure 3.23a but with the portal plane mesh removed allowing a view of the interior mesh (pink color). The yellow mesh here is the same as the green one in Figure 3.23a (exterior mesh).*

Figure 3.23: *Illustration of what the depth box looks like when all three depth box meshes (each mesh is marked with a separate color to tell them apart) are combined to create a single box.*

The second step of this process is to create a flat square plane that will be “glued” to the player’s camera (become a child object of it). The plane mesh should be one-sided (have backface culling enabled) and have a local position in the forward direction of the local X-axis (normal vector of the camera) of $N + 1$ where N is the near-clip plane distance of the player’s camera (if the local X position is equal or less than N, the player camera’s near clip plane will cull the mesh). This might seem completely illogical as it will make the player blind, but the logical reasoning behind it will be explained later in this section (section 3.2.3.4).

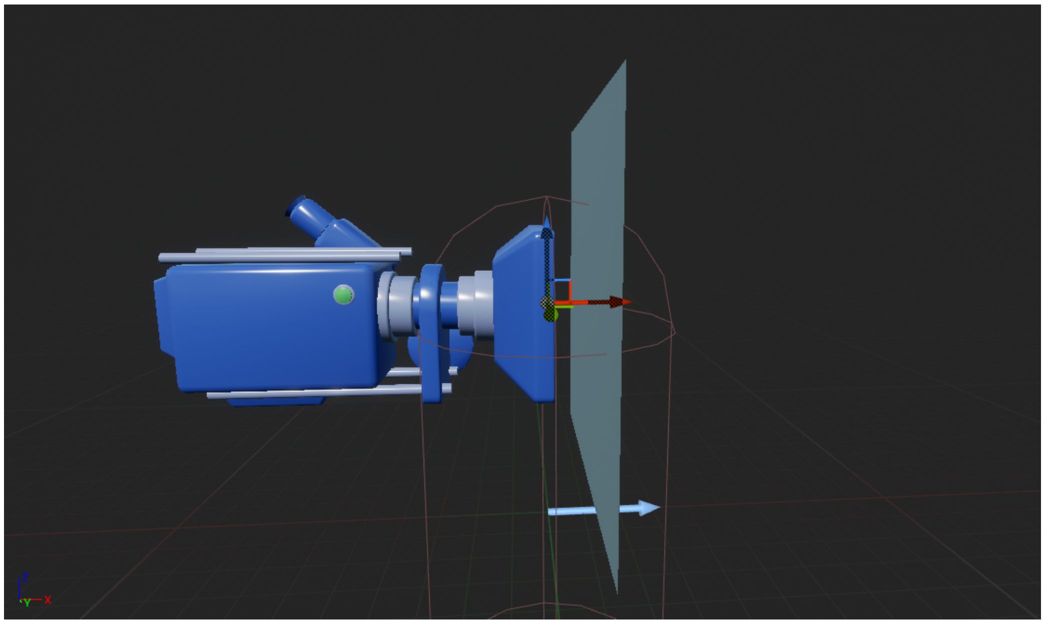


Figure 3.24: *Illustration of the flat plane (square) mesh being attached to the camera.*

The idea behind all of this is to make the front panel mesh (portal surface) temporarily invisible once the player character overlaps it (if it is invisible, the player's camera cannot clip the portal surface's mesh), allowing the player to see the inside of the box. Once the player character stops overlapping the portal surface, the portal surface becomes visible again. The player should only be able to see the interior of the box through the opening of the box (and only while overlapping the portal's surface). The box should be completely invisible (both exterior and interior) when viewed from an outside perspective, from any other angle, other than the opening itself.

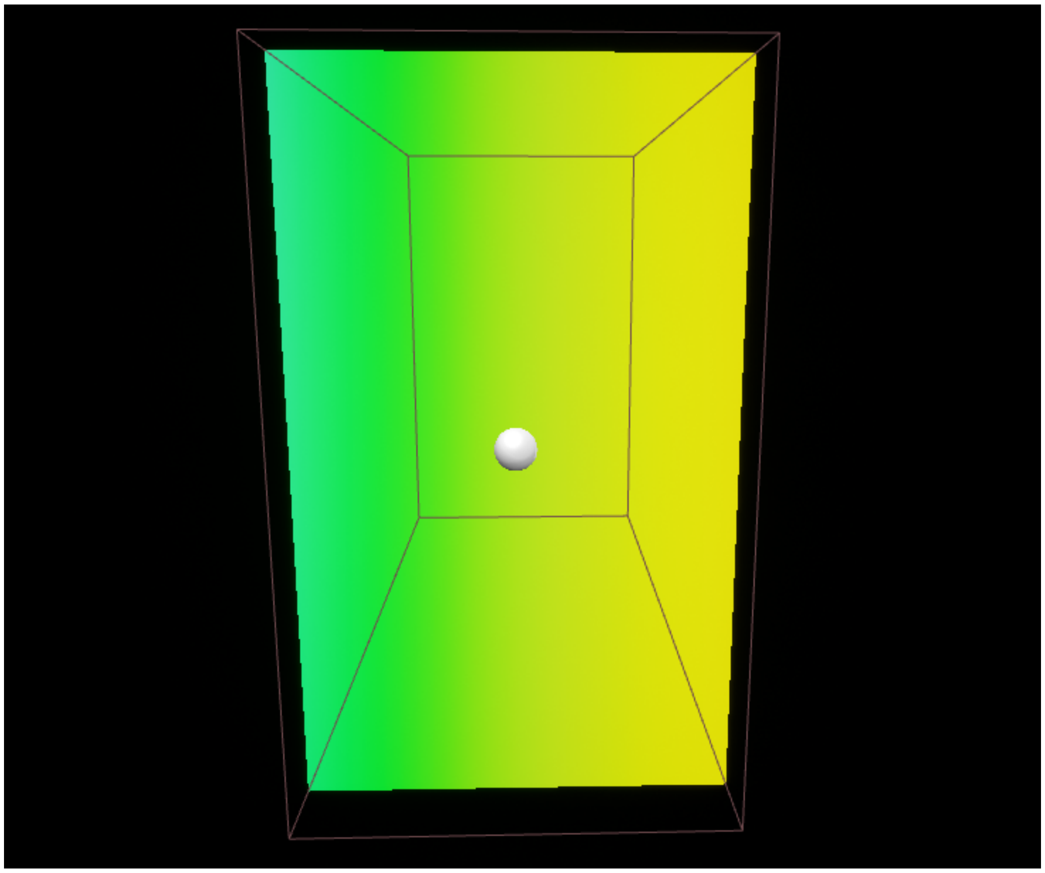


Figure 3.25: *The box interior when viewed from the front (while standing outside of it). Front panel (portal surface) is removed for demonstration purposes.*

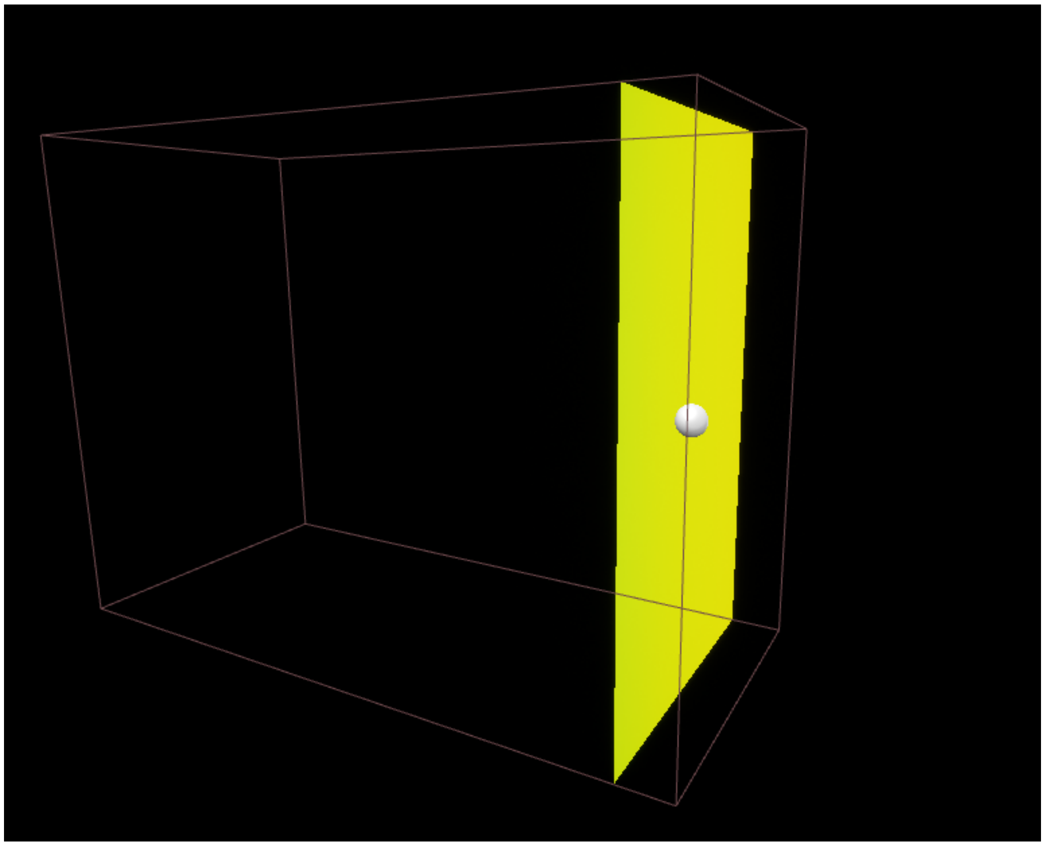
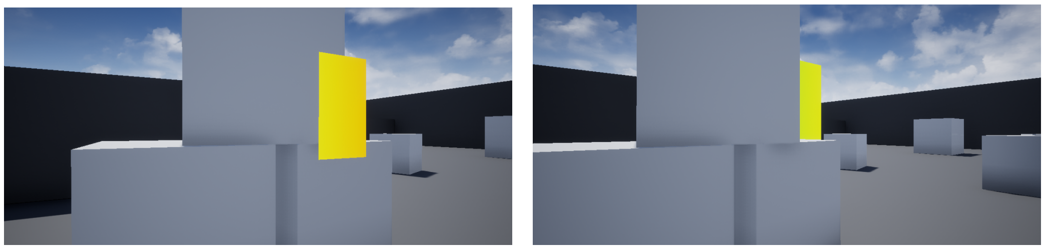


Figure 3.26: *The box when viewed from the side/any other angle. The black color means that it is not going to be rendered in-game (the red lines and white sphere are from the UE4 editor's gizmo tool and will not be visible in-game either).*

The interior mesh of the portal depth box will act as a mask for the "Portal viewer" (the flat plane that is attached to the player's camera). The portal viewer is made fully transparent so that the player can see through it. The exterior mesh is simply a shell that encapsulates the interior mesh (mask) and isolates it, ensuring that the interior of the mesh cannot be seen from outside the box (unless you look straight through the opening as shown in Figure 3.25). That is the exterior mesh's only purpose. No projection of textures or render targets is applied to the interior mesh. Instead, the projection of the render target is applied unto the portal viewer. Utilizing the stencil buffer, the interior box mesh gets masked out (a concept explained in section 2.2). This way, only a part of the render target is displayed on the portal viewer mesh (the part of the screen that the interior mesh occupies). Every other part of the projected render target is transparent. This makes it possible to render the interior of the box while not rendering the exterior. The purpose of the portal plane mesh (the separate front panel mesh of the box) is to create a visual representation of the portal surface when the player character is not overlapping the portal's surface. The interior mesh should only be visible when the player character overlaps the portal. Since the exterior and interior box meshes are not rendered in the main pass and only rendered in the stencil pass, they will not be occluded by any meshes rendered in the main pass (but can be occluded by other meshes rendered in the stencil pass). In other words, once the interior mesh is visible it will be rendered in front of every other object or mesh in the scene, regardless if that mesh or object blocks the view of the portal (see Figure 3.27). This is undesirable and therefore it is optimal if the interior and exterior meshes only are visible when the player character is really close to the portal (overlapping it). This creates a problem since that means that the portal would not be visible at all if the player is not overlapping it. That is where the portal plane mesh comes into play. The portal plane mesh will now act as a visual representation of the mesh when the portal is within the player camera's view frustum but the player character is not close enough to the portal's surface to overlap it.



(a) *In this figure, both the interior and exterior box mesh is being rendered in the stencil pass (but not in the main pass). The yellow rectangle seen is the interior box mesh being rendered. The portal plane mesh is not being rendered at all. Notice how the geometry in front of the portal does not properly occlude it (all the geometry seen, except for the portal, is not being rendered in the stencil pass, but instead in the main pass).*

(b) *Same situation as in Figure 3.27a but this time the yellow rectangle is the portal plane mesh (it is being rendered) and not the interior box mesh (the box has been sealed). This time the interior and exterior meshes are not being rendered in the stencil pass or the main pass (in fact, they are not rendered at all, they are invisible). The portal plane is not being rendered in the stencil pass either. Notice how the portal now gets properly occluded by the geometry in front of it.*

Figure 3.27: *Illustration of what happens when the portal plane mesh is invisible/visible. When it is invisible, the interior mesh is shown and does not get occluded by other geometry, creating a problem.*

While the property of the interior mesh rendering over all other meshes in the scene can be seen as a problem, it is however critical to the overall implementation of the portal system, as it has an unintended side effect which actually solves another unrelated problem. It allows the portal to be placed on other surfaces (walls, floors etc.) because when a portal is overlapped, while placed on a surface, the surface will not occlude the interior box mesh. This is desirable because if the surface did occlude the interior box mesh, the portal would no longer be rendered when overlapping it, breaking the effect.

In practice, this all is implemented by, first of all, going into the project settings and setting the "Custom Depth-Stencil Pass" option (in the "Post processing" category) to "Enabled with Stencil". This will allow access and modification of the depth and stencil passes in Unreal Engine 4. Then, inside the player character class, a one-sided (backface culled) flat plane mesh is created and made a child component of the player's camera. This will act as the "Portal viewer". It is be positioned in front of the camera with the rendered side of the plane mesh facing the camera view. The portal viewer is offset by $N + 1$ units from the forward direction of the camera, where N is the near-clip plane distance of the camera. This is to avoid the culling of the portal viewer mesh.

Next, the three meshes get imported and replace the old flat plane mesh in the portal class seen in Figure 3.2.

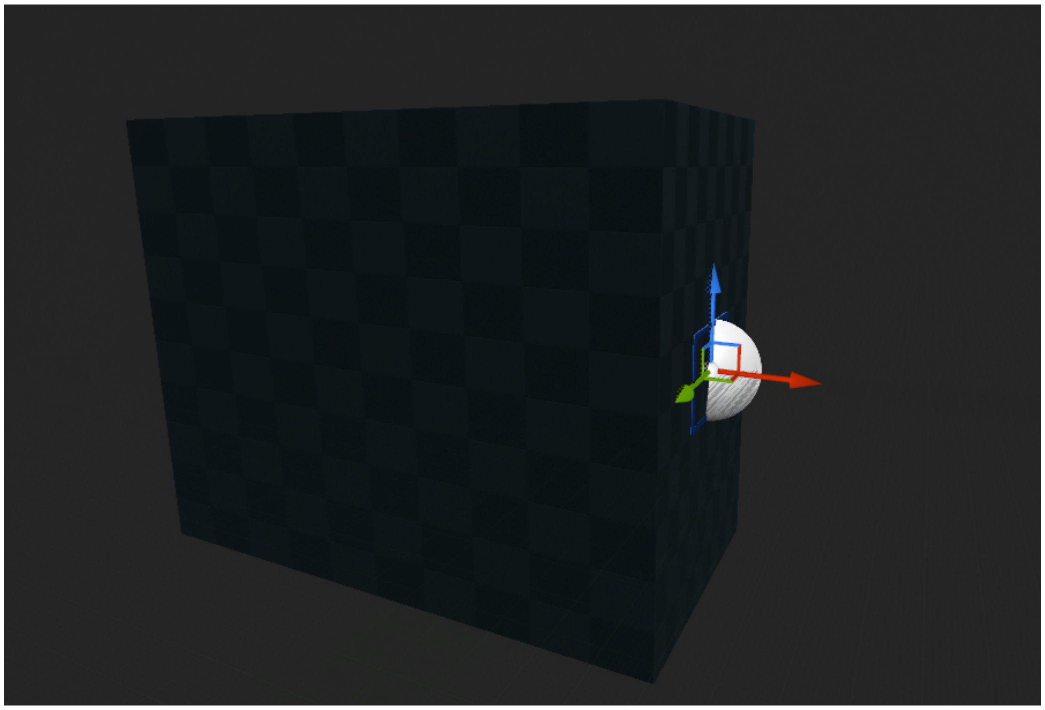


Figure 3.28: *This is how the portal class (visually) now looks like with the new portal depth box meshes (compare with Figure 3.2).*

Then, one new material (shader) is created. It will be called "M_PortalDepth" (name does not actually matter). It is a translucent material. The old material (seen in Figure A.1), "M_PortalPlane", is applied to the portal's surface mesh (front panel mesh), while the new "M_PortalDepth" material is applied to the portal viewer mesh in the player character class. The materials of the interior and exterior box meshes remain untouched. The shader code of the new "M_PortalDepth" material is very similar to the shader code of the "M_PortalPlane" material, except in this material the opacity of the material is controlled by the stencil values of the "CustomStencil" render target.

The material properties of the "M_PortalDepth" material are identical to the material properties of the "M_PortalPlane" material (Figure A.2).

The shader is useless (and malfunctioning) without any stencil values being applied to meshes and currently the portal meshes have no stencil values. Next step is therefore, to give the interior box mesh a stencil value of "1" (since that is the mesh that needs to be masked out in the stencil pass) while the rest of the meshes in the portal class (and the scene) are given a stencil value of "0" (which equates to a "M_PortalDepth" material opacity of zero, in other words, all meshes with a stencil value of zero will not be rendered in the stencil pass in the view of the portal viewer). It will also need to be made sure that the interior and exterior box meshes are not rendered in the main pass (there is a mesh setting for every mesh in Unreal Engine 4 called "Render in Main Pass" that does exactly this) because other camera's in the scene should not be able to see or render the exterior and interior box meshes.

As demonstrated in Figure 3.27, due to occlusion problems that the stencil buffer creates, the interior box mesh of the portal cannot constantly be rendered. Therefore, for practical reasons (mentioned earlier), the interior box mesh should only be rendered when the player character overlaps the portal's surface. In practice, this is done by disabling the visibility of the portal surface (front panel mesh) and enabling the interior/exterior box meshes to be rendered in the depth/stencil pass (should be disabled by default) when the player character overlaps a "trigger box" (this might be named differently in other game engines) placed near the portal surface. These two properties should be reverted as soon as the player character stops overlapping the trigger box (a trigger box does not necessarily need to be used as the collision detection of the portal's surface mesh works fine too). It is important to note that any mesh that can overlap the trigger box while the player character is overlapping it needs to have its stencil value set to "0", otherwise the interior box mesh will be rendered in front of every mesh that is inside the box (as shown by the occlusion problem demonstrated in Figure 3.27).

At this point in the implementation, the portals do not look like portals as there are no portal camera render targets applied to the texture parameters of the "M_PortalPlane" and "M_PortalDepth" materials. These should be applied and dynamically swapped during run-time. Two functions were created to solve this.

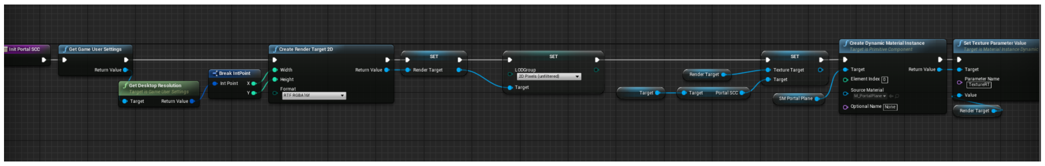


Figure 3.29: *This code is responsible for setting up the portal's scene capture component (portal's camera component). This is an updated version of the code shown in Figure A.3.*

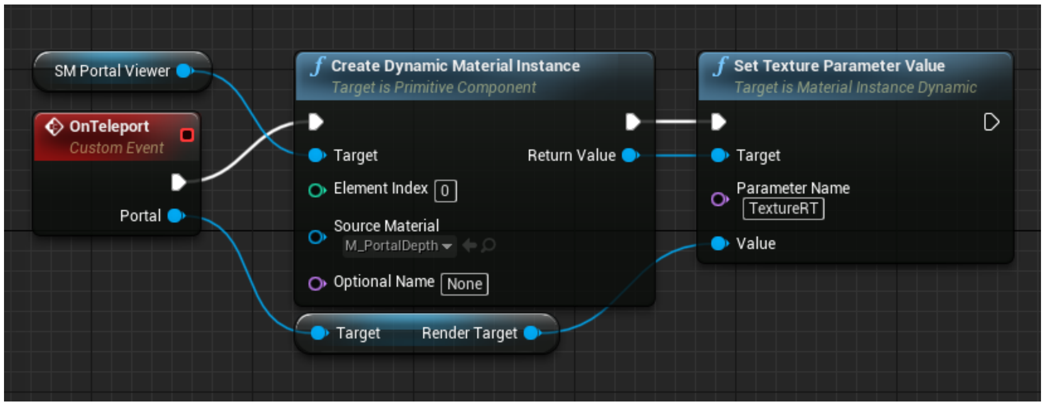


Figure 3.30: *This event code is executed whenever a player character overlaps a portal's surface. It is responsible for swapping the current texture parameter of the "M_PortalDepth" material to the projected render target of the currently overlapped portal.*

The first script (Figure 3.29) first creates a render target with an appropriate resolution (same as the in-game resolution), then gives the render target a new LOD (level-of-detail) group (this is to make the portal view as clear as possible, removing distortion, compression or any filters), then tells the newly created render target to be fed data from the feed of the target portal's scene capture component (portal camera), then the "M_PortalPlane" material gets applied to the portal's front panel mesh (portal surface) and finally, the texture parameter of the applied material gets set to be the newly created render target.

The second script (Figure 3.30) simply swaps the current texture parameter of the "M_PortalDepth" material to the render target of the portal that the player character is currently overlapping. This is so that the portal viewer knows which portal's interior mesh it should currently use as a mask.

3.3 Seamless portal interaction

This section will discuss how to create seamless physical interaction between two portals and solve some of the ambiguities that may occur when an object is placed between two portal spaces.

3.3.1 Carving holes in surfaces

Any object that attempts to pass through a portal that is placed on a surface will not be able to do that. This is due to the surface, which the portal is placed upon, being a solid, collidable piece of geometry. The object may pass through the portal itself but the surface behind the portal will stop the object from going all the way through the portal. In other words, portals that are placed upon surfaces do not (currently) physically behave as holes in a surface that objects may pass or fall through.

There are many ways to solve this problem, all with their own advantages and disadvantages. However, the best overall solution for general use cases, is to disable the collision strictly between the surface that the portal is placed upon and whichever object is currently overlapping the portal's surface (referring to the surface of the portal mesh, not the surface that it is placed upon). Keep in mind that while the collision between these two objects is disabled, it is only disabled for these two objects. In other words, any other object in the world should still be able to collide with these two objects while they are in this "disabled phase".

Unfortunately, Unreal Engine 4 does not explicitly support the disabling of collision between two specific objects in their collision filtering system, but there are ways to get around that by re-purposing another one of Unreal Engine 4's native systems (without having to modify any engine source code). Unreal Engine 4 has a built-in system for enabling physics constraints between two objects (built on NVIDIA's "PhysX" physics engine) [28]. This is usually used for things like vehicle suspension, swinging chandeliers, water wheels etc. It basically works like an invisible joint that connects two objects and physically constrains them based on a collection of user-set properties (see Figures 3.31 and 3.32).

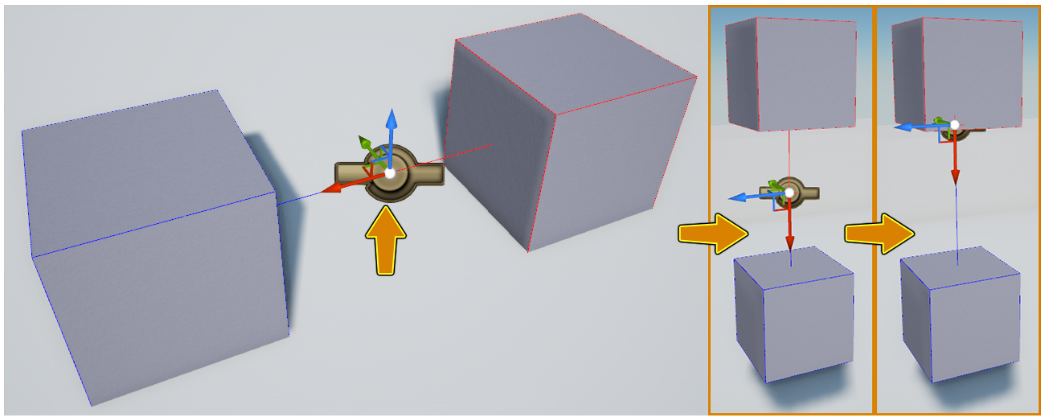


Figure 3.31: *Physics constraint example, showing two objects being constrained by a "physics constraint component" (joint).*
Source: `https://docs.unrealengine.com/en-US/Engine/Physics/Constraints/ConstraintsUserGuide/index.html`

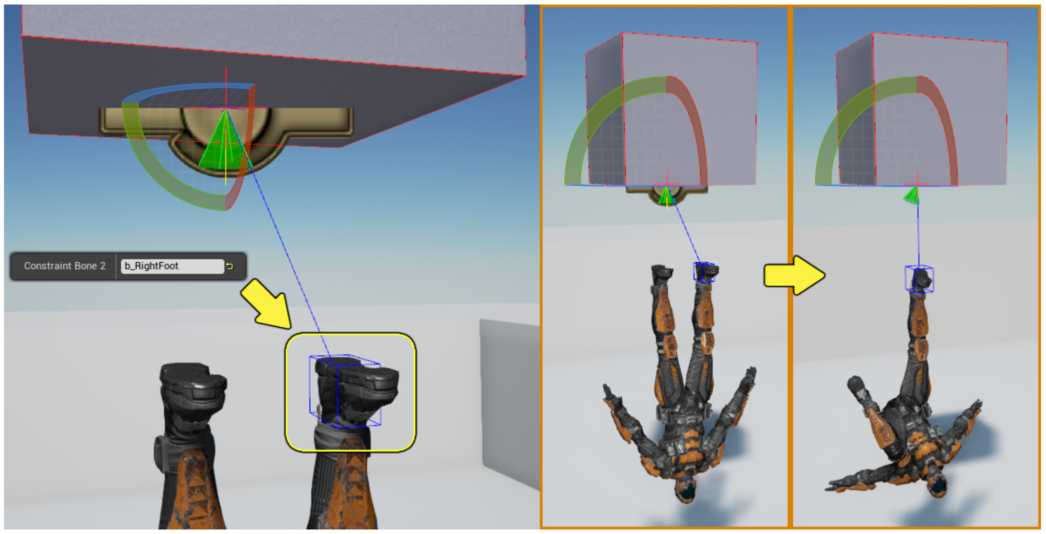


Figure 3.32: *Physics constraint example, showing a static mesh and a skeletal mesh being constrained by a "physics constraint component" (joint).*
Source: `https://docs.unrealengine.com/en-US/Engine/Physics/Constraints/ConstraintsUserGuide/index.html`

However, through some component property tweaking it is possible to nullify the physics constraints between the two objects, making them not physically affect each other whatsoever, while still being connected to each other. The properties of the physics constraint component also has a option to disable the collision between the two connected objects. These two facts coupled with the "physics constraint component", can now make it possible to disable the collision strictly between two objects while the two object are completely unconstrained, bypassing the limitations of Unreal Engine 4's collision filtering system.

In practice, this is done by creating a new class called "CollisionDisabler" that has a "physics constraint component" as a member variable and adding a map variable called "CollsionDisablerMap" to the portal class (similar to the clone map in described in section 3.2.3.3) and then making the portal class spawn an instance of this new class and adding the reference to the overlapping object as a key and the reference to the newly spawned "CollsionDisabler" instance as a value in the map every time a non-player object overlaps the portal's surface. Then when the "CollisionDisabler" instance is spawned, it executes a set of its own code which tells its "physics constraint component" to connect the overlapped object and the surface which the portal is placed upon to itself. The properties of this "physics constraint component" member variable are modified to nullify the physical constraints and disable the collision between its two connected objects (see Figure A.17).

The main purpose of the map variable is for the portal to know which "CollisionDisabler" instance needs to be destroyed when the object stops overlapping the portal. A new instance of the "CollisionDisabler" class is spawned for each object currently overlapping the portal's surface. This creates a many-to-one relationship between the surface that the portal is placed upon and each object currently overlapping the portal's surface.

The above method does not work for the player character itself. Another method is therefore needed to disable the collision between the surface that the portal is placed upon and the player character whenever the player character overlaps the portal's surface. This alternative method is relatively simple as Unreal Engine 4 does most of the work. Unreal Engine 4 has a built-in function in its API called "IgnoreActorWhenMoving". This function disables the collision between a specific object and a specific object component. This function takes three inputs; "Target" (Object component reference), "Actor" (Object reference) and "ShouldIgnore" (Boolean). "Target" is the object wants to pass through (not collide with) "Actor" and "ShouldIgnore" is a boolean that determines whether or not these two object should collide or not ("False" if they should collide, "True" otherwise). This function is called when a player character overlaps the portal's surface. The player character (overlapping object) is fed to the "Target" input and the surface which the portal is placed upon is fed to the "Actor" input. The value of "ShouldIgnore" switches depending on if the player character begins overlapping the portal's surface or if the player character stops overlapping the portal's surface. The value of "ShouldIgnore" is "True" when overlap begins and "False" when the overlap ends (see Figure A.21 and Figure A.22).

3.3.2 Adding edge collision to the portals

In order for the "carving of holes in portal surfaces" to work properly, some form of collision meshes need to be added to the edges of the portal surface (the surface of the portal object, not the surface the portal is placed on). This is because, according to the "collision disabler" solution above, an object that collides with the edge of a portal will still be able to go through the surface that the portal is placed upon, which is illogical. As an example, if you, in real life, touch the edge of a hole in wall with your hand, you should not be able to walk through the wall to the side of the hole itself just because your hand is touching the edge of the hole. That makes no sense. If collision meshes are placed around the edges of a portal, objects will now be able to collide with the edges of portals, preventing them from going straight through walls/floor when they are just touching the edge of a portal.

The addition of edge collision also allows for the player character and other objects to stand half-way through portals that are placed high up on walls. The bottom edge collision mesh prevents the objects from falling down. Similarly, this also prevents objects that are touching the edge of portals that are placed on a floor from falling through the floor.

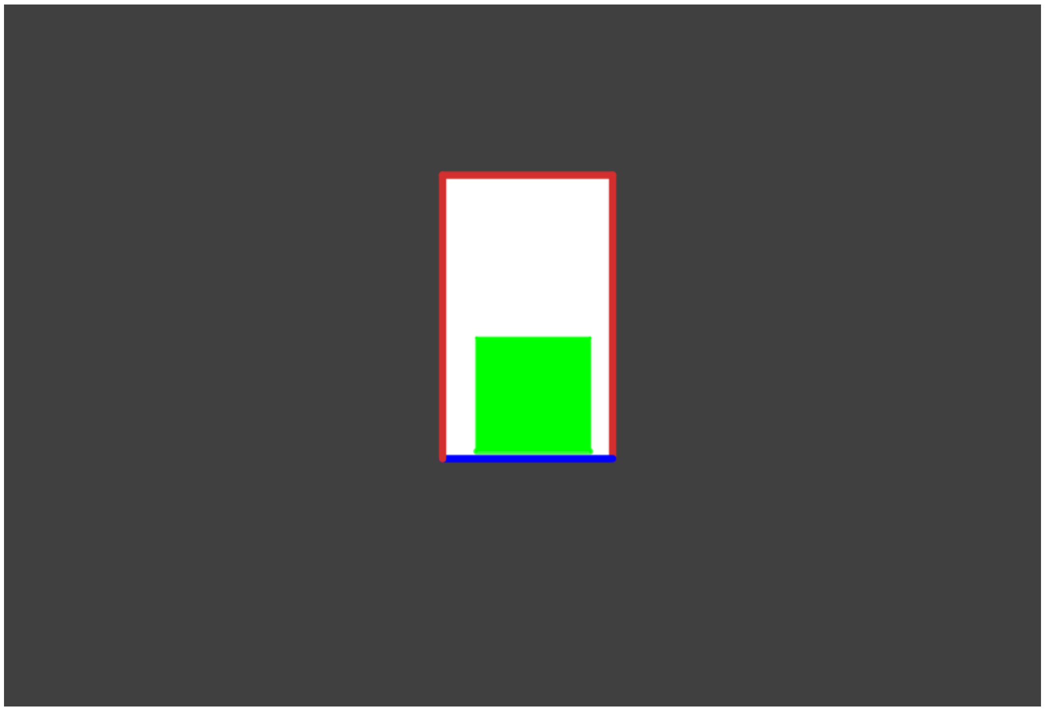


Figure 3.33: *Illustration of a portal (hole) being placed on a wall (grey) with a green cube placed on the edge of the portal. The blue collision mesh on the bottom edge prevents it from falling down, through the face of the wall.*

Adding collision meshes to the edges of a mesh can be done through a modeling software or in Unreal Engine 4 itself (using a "Box Collision" component in the object class or by adding collision via the static mesh editor).

3.3.3 Synchronizing physical interaction of base and clone objects

Looking back at the overlap clones in section 3.2.3.3, it is clear that the clones of the overlapping base object do not have any sort of ability to collide or physically interact with any other object in the world. They are purely visual clones of the base object. This will cause a problem if there exists any collidable objects in front of the exit portal (the local space of the clone). If the clone object attempts to collide with any object close to the front the of the exit portal, it will phase straight through it (ignore the collision). One might think that enabling the collision for the clone object simply solves the problem. It does not. If this is done, it will create another problem.

If you enable the collision for the clone object, the clone object will now correctly collide with the objects close to the front of the exit portal, but this will cause a desynchronization of the relative movement between the base and clone object. For example, if the clone object collides with a pillar placed close to the front of the exit portal and stops moving, the base object will not know that its clone has collided with the pillar nor that it has stopped moving and the base object will keep moving forward as there is nothing obstructing its path in its local space (the space around the entrance portal) while the clone is standing still, which causes a desynchronization of their movements, breaking the portal illusion.

The first idea is perhaps to tell the base object to imitate the movement of the clone object, but the clone object is already (through code) told to imitate the movement of the base object. This creates a logical paradox which makes no sense to the game engine. The base and clone objects are supposed to have a parent-child relationship and this is suggesting that the child object should dictate the movement of the parent object (while the parent object is told to dictate the movement of the child object), which is against all common sense as a programmer. An alternative solution needs to be found.

The alternative solution is to replicate and clone the local collision environments of each portal. In other words, whenever an object overlaps the surface of the entrance portal, the game collects references to a set of collidable objects placed near the front of the exit portal (within a reasonable range as large ranges would cause a performance hit). Using this set of references, an invisible clone of each collidable object in this set gets spawned and placed behind the entrance portal in portal space. Each collidable clone in this set objects will have custom collision properties which allow them to only collide with object that are currently overlapping the entrance portal. They ignore collision with any other object in the game world. This should be done for both static (objects that cannot move) and dynamic (objects that can move) objects. The cloned collision environment for the static set of objects only needs to be updated when whenever the portal's position changes, but the cloned collision environment for the dynamic set of objects needs to be updated each frame (because this set of objects could constantly move).

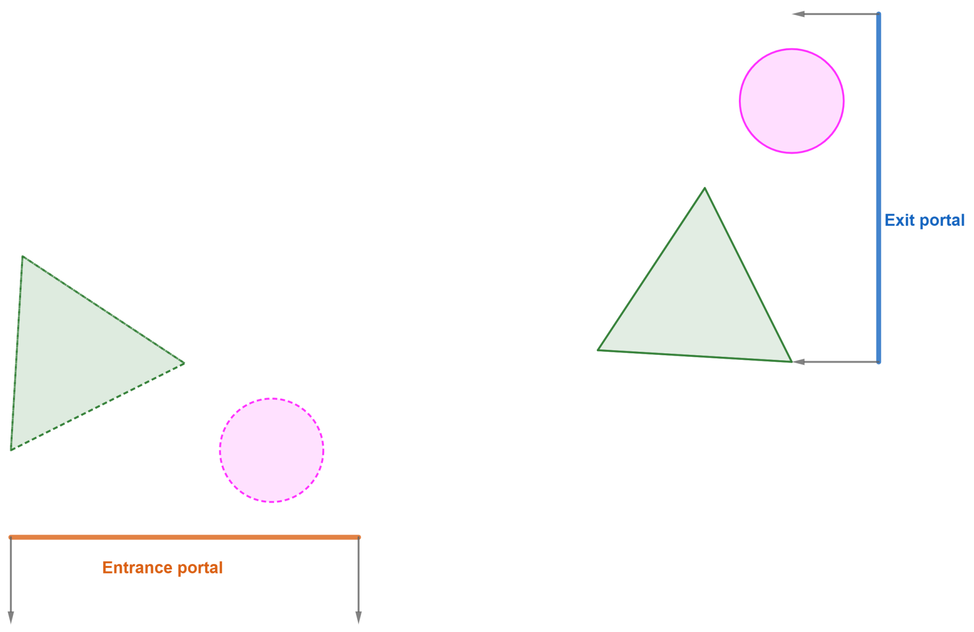


Figure 3.34: *Illustration of a cloned collision environment. The collection of shapes with dotted lines is the cloned collision environment of the set of objects that are in front of the exit portal.*

Now that the local collision environments are the same for each portal, the collision events of both the base and clone object will be identical, which means that a desynchronization of movement between them can no longer occur.

4

Results

The result of this experimental method is a portal system that is able to connect two non-adjacent spaces in a game world and make them appear adjacent. Instead of manipulating and morphing geometry in the game world in real-time to create a “real” non-euclidean space, this system is used to create an illusion of a singular non-euclidean space by connecting two non-adjacent euclidean spaces. Since the rendering and transition between these portals is seamless and the physics between the portal spaces (when an object is placed between two portals) behave accordingly, the player should not be able to tell that these portals even exist in the game world, thus maintaining the illusion of a non-euclidean space (the portals are, in practice, supposed to act as doorways between two rooms).

Below, a performance analysis will be conducted and presented, showing the performance implications of the method used.

4.1 Performance analysis

This performance analysis will be conducted using Unreal Engine 4’s built-in performance profiling tool suite (such as the “GPU Profiler” and “GPU Visualizer” tools, among others) [27]. Below, the testbed and the specifications of the computer used for the test will be presented.

4.1.1 Testbed and specifications

The testbed used will be a simple Unreal Engine 4 scene with default scene settings. The scene will contain nothing more than two portals and a couple of static meshes with simple base color materials applied to each mesh (see Figure 4.1). The scene was initially built upon the Unreal Engine 4's "Blank" game template.

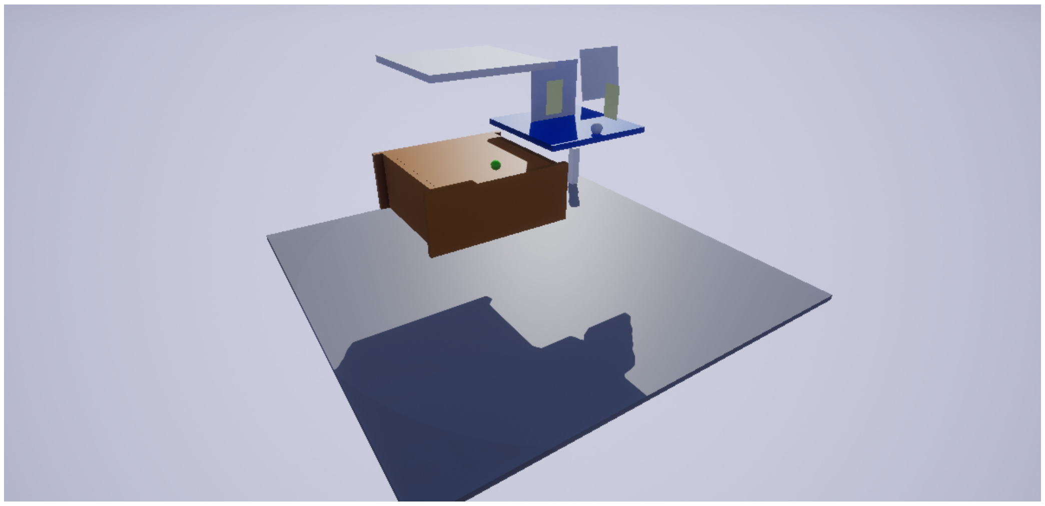


Figure 4.1: *The scene that will be used for the performance analysis. The two green planes are the (currently disabled) portals.*

Table 4.1 holds the specifications of the computer used for this performance analysis, along with the Unreal Engine 4 version.

Table 4.1: *Specifications of PC used for performance analysis*

Specifications	
`UE4 version`	4.22.3
`GPU`	NVIDIA GTX 1070
`CPU`	Intel i7 6700K @ 4.2 GHz
`RAM`	16GB DDR4 @ 3000 MHz
`OS`	Windows 10 Pro (version 2004, build 19041.508)

4.1.2 Identifying if CPU or GPU bound

Before doing any sort of performance profiling, one should check to see if the "game" (portal system and scene) is CPU (processor) or GPU (graphics card) bound. If a game is GPU bound, it means that the GPU completes its task during a frame slower than the CPU, in other words, the GPU "works harder" than the CPU and the CPU has to idle and wait for the GPU to finish its task before continuing to the next frame update. The meaning of "CPU bound" is the exact opposite. In an ideal situation, the game should be neither GPU or CPU bound (GPU and CPU should be perfectly in sync so none of them has to "wait" for the other).

Using the "stat unit" console command in Unreal Engine 4 we can get statistics over how much time it takes (in milliseconds) for each thread to finish its task during a frame. This can be used to identify whether the scene is GPU or CPU bound. The test scenario consists of the player standing still in front of two portals, which are both within the player camera's view frustum. Two tests are conducted. In the first test the portals are enabled (rendered) and in the second test they are disabled (not rendered). Below are the results.

Table 4.2: *Frametimes of each UE4 CPU/GPU thread (with portals enabled), including the framerate.*

Thread statistics (with portals)	
`CPU game thread`	0.81ms
`CPU render thread`	0.69ms
`GPU thread`	6.45ms
`Frames per second`	155

Table 4.3: *Frametimes of each UE4 CPU/GPU thread (with portals disabled), including the framerate.*

Thread statistics (without portals)	
`CPU game thread`	0.76ms
`CPU render thread`	2.53ms
`GPU thread`	2.53ms
`Frames per second`	399

The statistics in Table 4.2 show that the CPU threads finished their task much faster than the GPU. This confirms that the "game" is GPU bound when the portals are enabled. When the portals are then disabled in Table 4.3, the table shows that the frametime of the GPU thread has been significantly lowered, from 6.45ms to 2.43ms, while the frametime of the CPU render thread has increased from 0.69ms to 2.53ms, matching the frametime of the GPU thread. The reason the frametime of the CPU render thread increases is due to the drastic increase in framerate compared to the framerate in Table 4.2 which causes the CPU render thread to do draw calls and culling more frequently and thus running slower. However, the frametime of the GPU and CPU in Table 4.3 are equal which means that the "game" is neither GPU or CPU bound when the portals are disabled (which is optimal).

This all implies that the scene becomes GPU bound only when the portals are enabled, which means that the portal rendering must be GPU intensive. The built in “GPU visualizer” tool in Unreal Engine 4 will be used to profile the GPU and further analyze this.

4.1.3 GPU profiling

To profile the GPU, Unreal Engine 4’s built-in “GPU visualizer” tool will be used. This tool allows a developer to analyze the GPU as it shows the time it takes for each task in a frame to be completed. The tool is mostly used for optimization as it allows for developers to find bottlenecks in the GPU render pipeline.

The results for the frametimes of rendering each portal view (their SceneCapture components) along with the frametime of the entire scene (final frame output, excluding the two portal views) can be seen in Table 4.4.

Table 4.4: *The GPU frametimes of both portal views and the entire scene (final frame output, excluding the two portal views).*

GPU frametimes	
`Portal A (SceneCapture)`	2.18ms
`Portal B (SceneCapture)`	2.16ms
`Portal frametime avg.`	2.17ms
`Scene`	2.37ms

In Table 4.4 it can be seen that it takes almost as much time to render a single portal view as it does to render the whole game scene (with frustum culling). In other words, the rendering of the a single portal view is very expensive. By doing some basic calculations we can get the cost of a single portal and the cost of both portals in percentages.

$$P_A = 2.18ms \tag{4.1}$$
$$P_B = 2.16ms \tag{4.2}$$
$$S_C = 2.37ms \tag{4.3}$$
$$S = P_A + P_B + S_C = 6.71ms \tag{4.4}$$
$$R = (P_A + P_B)/S = 4.34/6.71 \approx 0.65 \tag{4.5}$$

P_A is the time it takes to render portal A. P_B is the time it takes to render portal B. S_C is the time it takes to render the scene (without the portals). S is the combined time that it takes to render P_A, P_B and S_C. R is the ratio between the time it takes to render portals and the time it takes to render everything in the scene (portals included).

Both portals take up 65% of the time it takes to render the final frame output (6.71ms). This result is, however, logical because when you are rendering a portal view you are capturing the entire scene (within the view frustum of the SceneCapture component) again but from a different perspective and applying the final result unto a render target (texture). In other words, when you are rendering a scene with two portals in view, you are rendering the scene three times instead of one.

Taking the framerate values from Table 4.2 and 4.3 and calculating the ratio $1 - (155/399) = 0.61$, it is seen that the calculated ratio gets very close to the calculated percentage above (65% compared to 61%) and further confirms the calculated rendering cost of the portals.

Since the render time of a portal view and the entire game scene has a 1:1 ratio, we can conclude that

$$T_f \approx (n + 1) * S_f$$

where n is the number of portals in the game scene, S_f is game scene frame-time (time it takes to render the scene in a single frame) and T_f is the total frame-time (time is takes to render the final frame output).

5

Discussion

Although a method was found and implemented for creating non-euclidean spaces, the method used is not perfectly optimal. First of all, the method is not actually creating non-euclidean spaces but is faking them by creating the illusion of a non-euclidean space. It is arguable whether this makes any actual difference since, these two possible approaches are indistinguishable from the perspective of the player, which is what actually matters in the end. It is also arguable whether is is practical or even possible to create real non-euclidean geometry and spaces in a game engine that is inherently euclidean (along with all of its game engine systems that are also euclidean as they assume that they are being applied upon a non-euclidean game world), without modifying any source code.

5.1 Drawbacks and limitations

The portal system has a fair amount of drawbacks and limitations, which can limit the type of games that can use the portal system. The developer implementing a system such as this needs to be mindful of these limitations and design their game around the portal system rather than the other way around. This is not a system that can simply be dropped into an existing game and be expected to work seamlessly. These numerous limitations and drawbacks are listed below:

- The current implementation does not work with Temporal Anti-Aliasing methods (TAA). It will cause minor visual artifacts in the portal view upon player camera movement. It does however work with other anti aliasing methods such as "Fast Approximate Anti-Aliasing" (FXAA) and "Multisample Anti-Aliasing" (MSAA). Support for other anti-aliasing such as "Morphological Anti-Aliasing" (MLAA) has not been tested as Unreal Engine 4 only natively supports TAA, FXAA and MSAA.

- With the current implementation, cloning of objects overlapping a portal's surface does not work with objects that have child components (subobjects) attached to them that simulate physics, objects that change their visual appearance over time and physics ragdolls.

- Portals cannot be placed on collidable surfaces that have another collidable surface directly behind the surface that the portal is being placed on (as the "CollisionDisabler" class does not currently recursively disable collision for multiple surfaces).

- Using render targets/render textures for the rendering of the portal views will consume VRAM (video memory) for each view that is being rendered as texture data is being stored on the GPU (graphics card).

- The portals do not transfer light from one portal to another. Therefore, teleporting an object between two spaces with highly contrasting lighting conditions can appear strange.

- Portal recursion does not work. You cannot recursively see the view of a portal through the view of the portal the player is looking through.

- Planar reflections (mirrors, water reflections etc.) appear broken when viewed through the view of a portal.

- If a mesh overlaps a portal surface, the part of the mesh that is sticking out behind the portal's surface gets rendered. In other words, that part of the mesh does not turn invisible. This can sometimes cause the portal illusion to break when a portal is placed out in the open (not placed on a opaque surface).

- The portal system does not currently support virtual reality.

Keep in mind that some of these limitations might have to do with the fact that the system was implemented in Unreal Engine 4. If this system was implemented in any other game engine, some of the limitations and drawbacks might be different due to the engine render pipeline being different or it having features and tools that Unreal Engine 4 does not. The opposite also applies. Unreal Engine 4 might have tools and features that other game engines are missing.

5.2 Error sources

Throughout this thesis there may have been some error sources that have degraded the overall quality of the result.

One of these error sources is the limited experience with game development and Unreal Engine 4, along with the lack of experience with shader programming from the author.

Another error source is caused by the subject itself being very niche and unusual, hence there being a great lack of documentation and proper references for portal systems built in game engines, as this is something very few people have seemingly attempted and documented online. Neither Valve or any other game developer that has created a portal system, has provided any official documentation of its implementation.

5.3 Further use-cases

Beside using this portal system for making impressive effects and creative game mechanics, it can have other use-cases. One such use-case is to improve player movement in virtual reality applications and games.

Most modern virtual reality headsets have support for some form of "room-scale locomotion". Room-scale locomotion means that the virtual reality headset is (in some way) mapping the player's real-life play-space and tracking their movement in the play-space (room), to then translate that movement to movement in the virtual space (game world etc.), allowing players to move around in the virtual reality world by moving around in the real-life play-space. The obvious limitation to this is that they cannot move an infinite distance as they are bound to the physical size of their play-space (room) and thus the virtual play-space in the virtual reality game/application has to be made smaller to match the size of the physical play-space. VR-developers that want to make larger-scale games usually combat this by allowing alternative locomotion options for the player, such as the ability to walk using a button on the accompanied virtual reality touch-controllers or instantly teleporting them using a non-seamless transition, but these methods usually break player immersion to some degree.

A seamless portal system, such as the one in this thesis, would allow a virtual reality player to visit various different virtual spaces in the game by moving a much shorter distance in their physical play-space, as portals inherently shorten the distance between virtual spaces and are thus able to compress several virtual spaces into a single virtual space, if done correctly. The portal system will not completely eliminate the problem of a limited physical space constraining movement extents in a virtual reality space but will help alleviate it by allowing the player variation in their constrained virtual space, making the virtual space seem larger and more open than it really is.

Although the portal system implementation in this thesis does not support virtual reality, this could still be a potential use-case if anyone improves it and adds support with future work.

6

Conclusion

The aim of the thesis was to, if possible, figure out a practical way to represent non-euclidean spaces in an already existing euclidean game engine and making these spaces physically interactable. Several viable methods are able to be used to solve the problem, however, some restrictions were put into place in this thesis that restricted the amount of viable methods to choose from. The main restrictions are the inability to add or modify any engine source code and the inability to utilize any ray-tracing based methods. The first restriction was put into place for two reasons:

1. To ensure that the solution was practical for game developers to use and implement, regardless of team size or available resources. Editing engine source code in order to implement a game system requires extensive knowledge of the engine code-base in question and often requires extra resources in the form of specialized engine programmers. Moreover, if engine code is modified in an existing game project, these modifications risk breaking existing game systems, adding extra complexity to the implementation. The level of complexity can vary greatly from project to project.

2. To ensure the integrity of the term "euclidean game engine". If modification of source code is allowed, the game engine can theoretically be gutted or morphed too much, to a point where it is no longer considered euclidean. Theoretically, anything is possible if unlimited source

code modification is allowed, negating the entire point of the question to be answered in this thesis.

The second restriction is due to real-time ray-tracing technology being expensive and being in its very early stages (both in development and adaptation). Moreover, according to Steam's hardware survey, only around 12% of PC users currently own a graphics card capable of supporting real-time ray-tracing [21]. Consoles only recently started supporting real-time ray-tracing with the release of the Xbox Series X and Playstation 5.

The method that is ultimately chosen is a portal-based method which does not explicitly create non-euclidean spaces but is able to trick the player into believing that they are looking at, residing in or physically interacting with a non-euclidean space by placing the portals in certain configurations. Essentially creating an illusion of non-euclidean space (see Figure 6.1). This allows the representation of non-euclidean spaces in euclidean game engines, all while not breaking any of the game engine's logic or scripting API as it assumes that it is operating upon a euclidean game world.

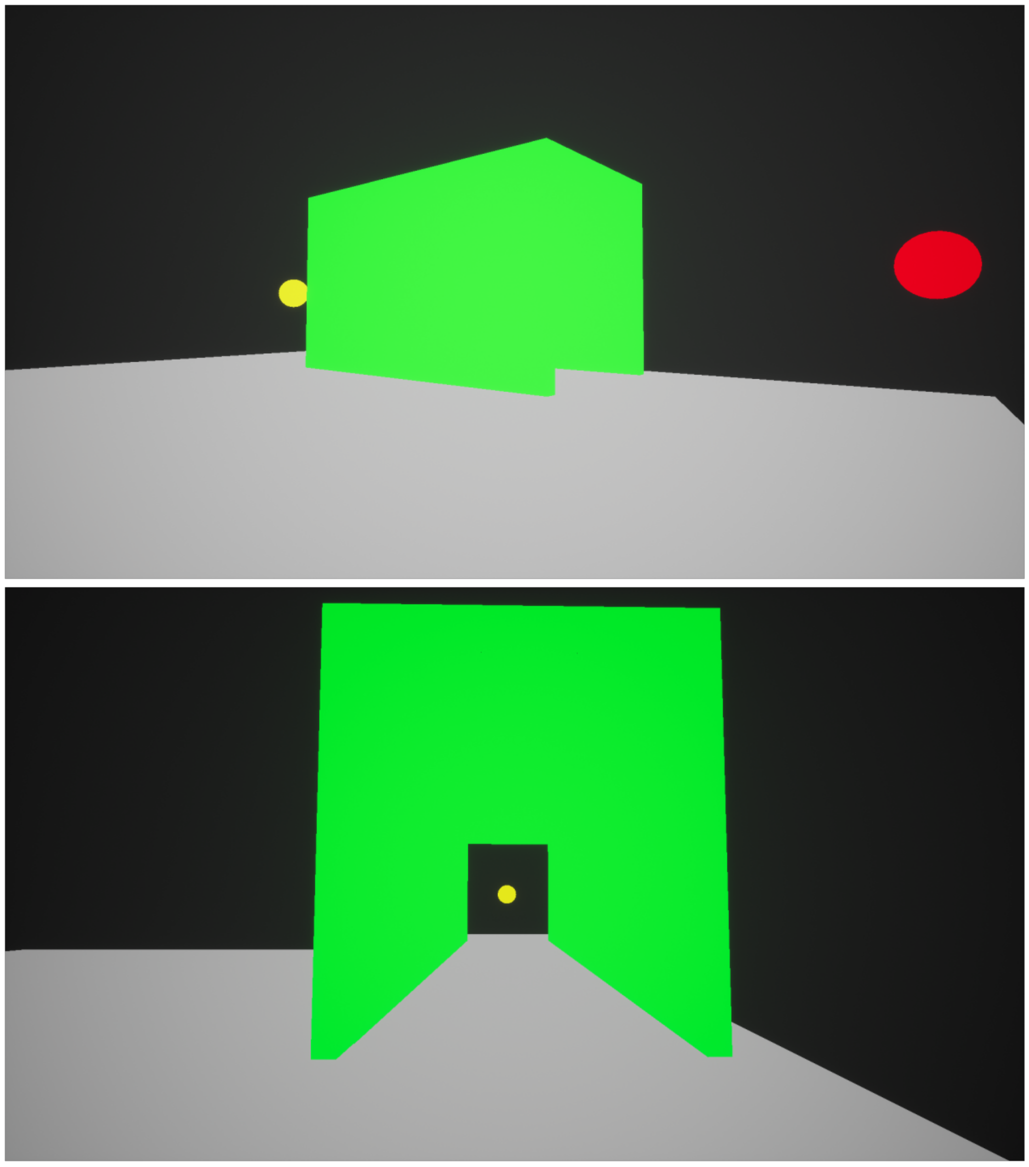


Figure 6.1: *Example of a possible portal configuration to create the illusion of non-euclidean space. This green tunnel is shorter on the outside than the inside. It takes a longer time for an object to travel through it than around it (at equal velocity) and thus the shortest path between two points is no longer a straight line.*

While this method is found to be versatile and can be used to create many different types of non-euclidean spaces, it is not without its flaws and limitations. The most prominent flaw is its heavy performance impact. It is very GPU heavy as it takes up just as much frame-time to render a single portal view as it does to render the whole game scene. In other words, the rendering time of the entire game scene and one portal view has a 1:1 scaling ratio. For example, if it takes X milliseconds to render a game scene, it will take X milliseconds to render the portal view, making the total frame-time $2X$ milliseconds. The more complex the game scene is, the more time it takes to render the portal view.

Some of the limitations of the system can be fixed or improved given enough time, while some other limitations may be impossible to solve, either due to game engine limitations or an inherent limitation of the method itself.

In the end, a method was found that could be used to represent non-euclidean spaces and let players and the game environment interact with these spaces in a believable way. Although the resulting system is not optimal and has its flaws, it is still able to currently be used for certain games in a somewhat limited capacity. If the limitations of the system are improved or fixed this system has the potential to be used universally for games which need to portray non-euclidean concepts.

7

Future work

The final portal system has a clear list of limitations and missing features, most of which can be improved upon. The severity of these limitations (and missing features) vary greatly and thus it is best to make a prioritization list of the limitations (and missing features) that are the most critical to solve or improve.

1. **Performance optimization**

   The performance impact of rendering a portal is immense. It takes the same amount of time to render a portal view as it does to render the whole scene. In other words, there is a 1:1 performance impact ratio. If a game scene gets more complex to render, the portal view will become equally complex to render (scales linearly). Essentially, the time complexity of the portal rendering is currently linear. This is not feasible for large and visually complex game environments, and thus the system is currently only viable to use in simple, enclosed game environments, or in games with low visual complexity. This greatly decreases the usage versatility of the system. Beyond basic optimization (for example, not rendering the portals when the player is not looking at them) it is unclear how much optimization can be done without affecting visual clarity of the portals (which is essential for the system to be effective) or breaking the portal view projection (for example, the projection of the portal view is dependent on the resolution of the portal's render target, so scaling down the resolution of the portal view would not work). It is even more unclear how much optimization can be done without modifying any of the engine's source code.

2. **Portal view recursion**

   The portals cannot render each others views recursively. In other words, a portal that can be seen within the view of another portal will not have its view rendered in the view of that other portal (see Figure 1.3 for a visual example of a proper portal recursion). This should work recursively (Portal B should be rendered within the view of Portal A and Portal C should be rendered within the view of Portal B and so on), up to a certain upper limit (computers cannot recurse infinitely). Ideally, this upper limit should be decided during run-time based on the current hardware that the game is being run on. Without portal view recursion, some non-euclidean spaces or concepts cannot be represented in a game. For example, if you need to create an infinite staircase effect, the portal at the bottom of the staircase needs to render the portal at the top of the staircase within its own view. Otherwise the effect is impossible to do seamlessly. Unfortunately, recursive portal rendering also has a big performance impact (rendering multiple portal views within a single portal view is not cheap on performance) and thus the above (non-recursive rendering) performance optimization problem needs to be solved before implementing a solution for recursive rendering.

3. **Reduce VRAM consumption**

   The portal system is currently using a render-target for each portal view that it needs to render. Since a render-target is essentially a normal game texture, the data for each render-target will be stored on the GPU's VRAM (VRAM stands for "Video Random Access Memory" and is essentially the memory/RAM of a graphics card). This is currently not a big problem but may become one if portal recursion is ever implemented since a separate render-target needs to be created for each recursive portal view. If the portal views could instead be rendered in a single render-pass (for example, with a stencil buffer), it would only require a single, relatively smaller memory allocation, thus saving a lot of VRAM. Unfortunately, it seems that this is not currently supported by Unreal Engine 4 (without modifying source code) as the render-target capture taken by a "SceneCaptureComponent2D" component cannot be accessed within UE4's shader editor. This limitation is thus, of course, game engine dependant. This may not be a limitation in other game engines.

4. **Virtual reality support**

   The portal system does not currently work in virtual reality applications.

5. **Portal light transfer**

   The portals are not currently able to transfer light from one portal to another, resulting in strange visual artifacts in teleportation transitions when teleporting objects between two game environments with very contrasting lighting conditions. Since the rendering of light is baked into a game engine's render pipeline and not able to be modified through normal engine usage (in the case of most modern game engines), it is unclear if this problem is even solvable without modifying game engine source code.

The prioritization list may change slightly based on the use-case of whomever intends to use the portal system, but these five limitation are the most general ones. All other limitations and missing features mentioned in section 5.1, can also be considered for future work, but are not currently as critical to solve as the ones listed above.

References

[1] J. Playfair. *Elements of Geometry: Containing the First Six Books of Euclid, with a Supplement on the Quadrature of the Circle, and the Geometry of Solids; to which are Added, Elements of Plane and Spherical Geometry.* W. E. Dean, 1846. URL: `https://books.google.com/books?id=xjcPAAAAYAAJ`.

[2] E. Beltrami. *Saggio di interpretazione della geometria non-Euclidea.* 1868. URL: `https://books.google.com/books?id=G_RZAAAAcAAJ`.

[3] William P. Thurston and Silvio Levy. *Three-dimensional geometry and topology.* Princeton mathematical series 35. Princeton, N.J: Princeton University Press, 1997. ISBN: 9780691083049.

[4] *DirectX 12.* Mar. 2014. URL: `https://devblogs.microsoft.com/directx/directx-12/` (visited on 03/27/2021).

[5] Euclid, Thomas Little Heath, and Dana Densmore. *Euclid's Elements: all thirteen books complete in one volume : the Thomas L. Heath translation.* English. OCLC: 1109844025. 2017. ISBN: 9781888009194.

[6] *Falcor.* July 2017. URL: `https://developer.nvidia.com/falcor` (visited on 03/27/2021).

[7] *NVIDIA RTX™ platform.* en. July 2018. URL: `https://developer.nvidia.com/rtx` (visited on 09/01/2020).

[8] João Rodrigo, André Silva, and Alves Silva. "Ray Tracing in Non-Euclidean Spaces". MA thesis. 2018.

[9] Eric Haines and Tomas Akenine-Möller, eds. *Ray Tracing Gems.* `http://raytracinggems.com`. Apress, 2019.

[10] Luiz Velho, Vinicius da Silva, and Tiago Novello. "Immersive Visualization of the Classical Non-Euclidean Spaces using Real-Time Ray Tracing in VR". In: University of Toronto: Canadian Human-Computer Communications Society / Société canadienne du dialogue humain-machine, 2020, pp. 423–430. ISBN: 978-0-9947868-5-2. DOI: 10.20380/GI2020.42.

[11] *Antichamber - A Mind-Bending Psychological Exploration Game.* URL: http://www.antichamber-game.com/ (visited on 09/03/2020).

[12] *Antichamber — Unreal Engine interview.* en-US. URL: https://www.unrealengine.com/en-US/blog/antichamber (visited on 09/03/2020).

[13] *Distributing Source Engine Games (Steamworksdocumentation).* URL: https://partner.steamgames.com/doc/sdk/uploading/distributing_source_engine (visited on 09/04/2020).

[14] *Euclid — Biography, Contributions, & Facts.* en. URL: https://www.britannica.com/biography/Euclid-Greek-mathematician (visited on 09/01/2020).

[15] Blender Foundation. *blender.org - Home of the Blender project - Free and Open 3D Creation Software.* en. URL: https://www.blender.org/ (visited on 04/06/2021).

[16] The Khronos Group. *OpenGL - The Industry Standard for High Performance Graphics.* URL: https://www.opengl.org/ (visited on 04/14/2021).

[17] The Khronos Group. *Vulkan - Industry Forged.* en. URL: https://www.khronos.org/vulkan/ (visited on 04/14/2021).

[18] *Intel Software Engineers Assist with Unreal Engine.* en. URL: https://www.intel.com/content/www/us/en/develop/articles/intel-software-engineers-assist-with-unreal-engine-419-optimizations.html (visited on 09/16/2020).

[19] Microsoft. *Stencil Buffer Techniques (Direct3D 9) - Win32 apps.* en-us. URL: https://docs.microsoft.com/en-us/windows/win32/direct3d9/stencil-buffer-techniques (visited on 09/06/2020).

[20] *Portal (game).* URL: https://store.steampowered.com/app/400/Portal (visited on 09/04/2020).

[21] *Steam Hardware & Software Survey*. URL: https://store.steampowered.com/hwsurvey/videocard (visited on 12/12/2020).

[22] *Unreal Engine — The most powerful real-time 3D creation platform*. en-US. URL: https://www.unrealengine.com/en-US/ (visited on 09/01/2020).

[23] *Unreal Engine 4 API - UCameraComponent*. en-US. URL: https://docs.unrealengine.com/en-US/API/Runtime/Engine/Camera/UCameraComponent/index.html (visited on 09/15/2020).

[24] *Unreal Engine 4 Documentation - Actor Ticking*. en-US. URL: https://docs.unrealengine.com/en-US/Programming/UnrealArchitecture/Actors/Ticking/index.html (visited on 09/16/2020).

[25] *Unreal Engine 4 Documentation - Camera Components*. en-US. URL: https://docs.unrealengine.com/en-US/Engine/Components/Camera/index.html (visited on 09/10/2020).

[26] *Unreal Engine 4 Documentation - Collision Response Reference*. en-US. URL: https://docs.unrealengine.com/en-US/Engine/Physics/Collision/Reference/index.html (visited on 09/17/2020).

[27] *Unreal Engine 4 Documentation - Performance and Profiling*. en-US. URL: https://docs.unrealengine.com/en-US/Engine/Performance/index.html (visited on 10/05/2020).

[28] *Unreal Engine 4 Documentation - PhysX Constraint User Guide*. en-US. URL: https://docs.unrealengine.com/en-US/Engine/Physics/Constraints/ConstraintsUserGuide/index.html (visited on 10/01/2020).

[29] *Unreal Engine 4 Documentation - Render Targets*. en-US. URL: https://docs.unrealengine.com/en-US/Engine/Rendering/RenderTargets/index.html (visited on 09/10/2020).

[30] *Unreal Engine 4 Documentation - Rendering Overview*. en-US. URL: https://docs.unrealengine.com/en-US/Engine/Rendering/Overview/index.html (visited on 09/10/2020).

[31] *Unreal Engine 4 Documentation - Scene Capture 2D*. en-US. URL: https://docs.unrealengine.com/en-US/Resources/ContentExamples/Reflections/1_7/index.html (visited on 09/10/2020).

[32] *VALVE, Source SDK 2013 license*. URL: https://github.com/ValveSoftware/source-sdk-2013 (visited on 09/04/2020).

[33] *World space, object space, and local space — Maya 2018 — Autodesk Knowledge Network*. en-US. URL: `https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2018/ENU/Maya-Basics/files/GUID-A63AC5C8-8822-42AC-827E-164B5266DA03-htm.html` (visited on 09/12/2020).

Appendix

A Project properties and Blueprint code

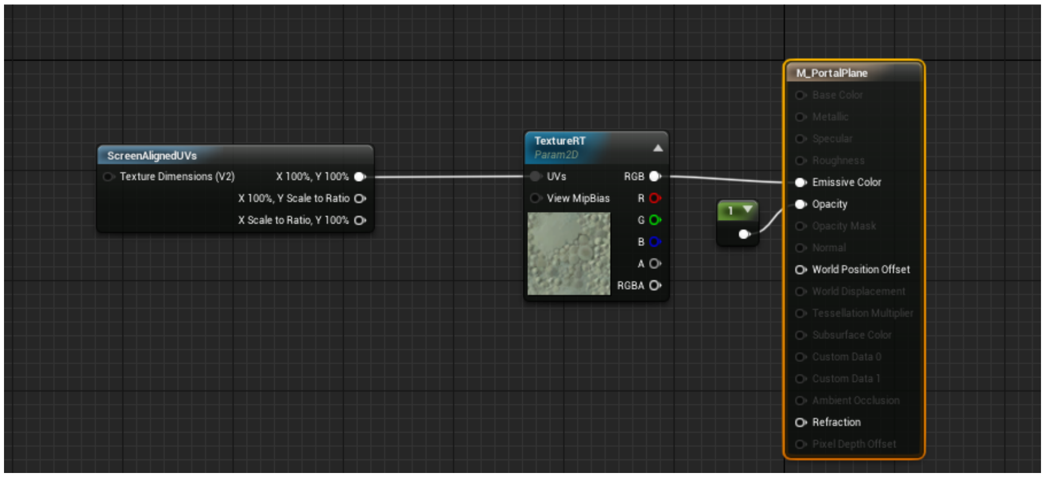
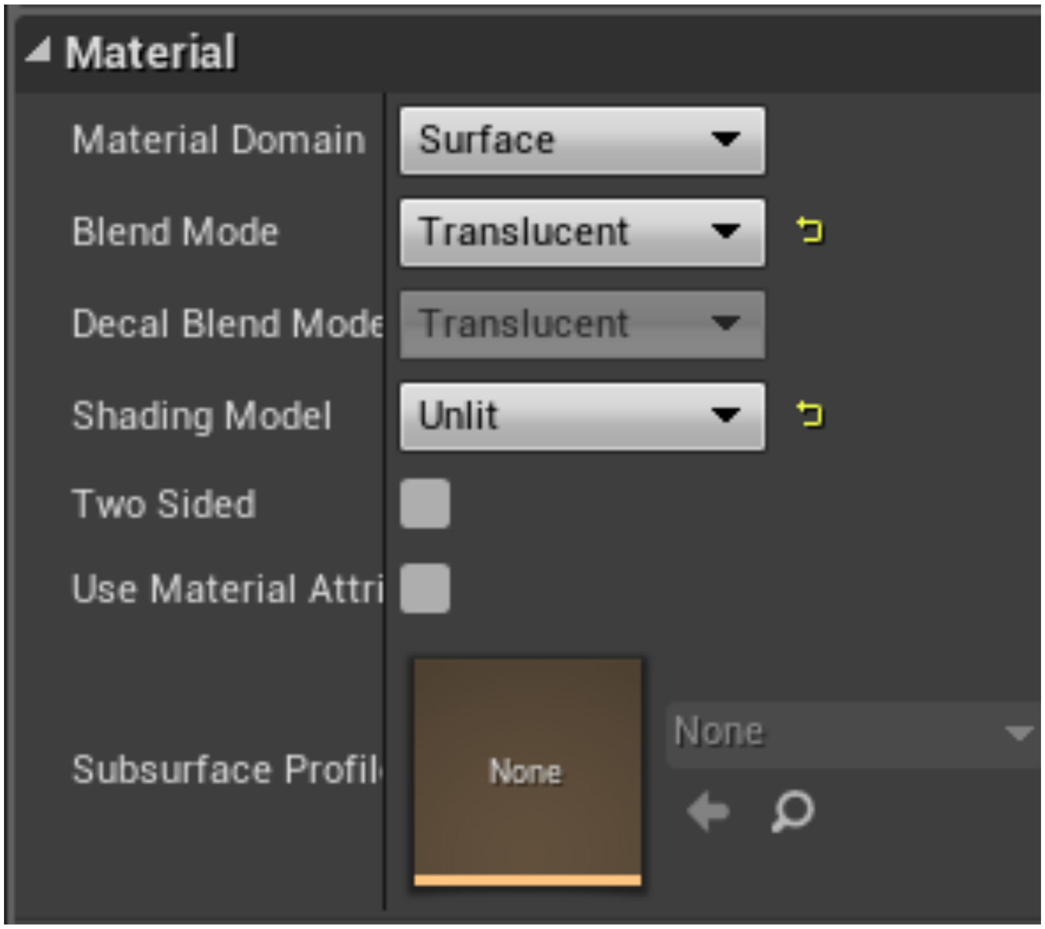


Figure A.1: *This is what the shader of the portal surface looks like in Unreal Engine 4's blueprint editor (visual scripting system).*



Material

Material Domain	Surface
Blend Mode	Translucent
Decal Blend Mode	Translucent
Shading Model	Unlit
Two Sided	☐
Use Material Attri	☐
Subsurface Profil	None

Figure A.2: *These are the property settings of the portal surface shader.*

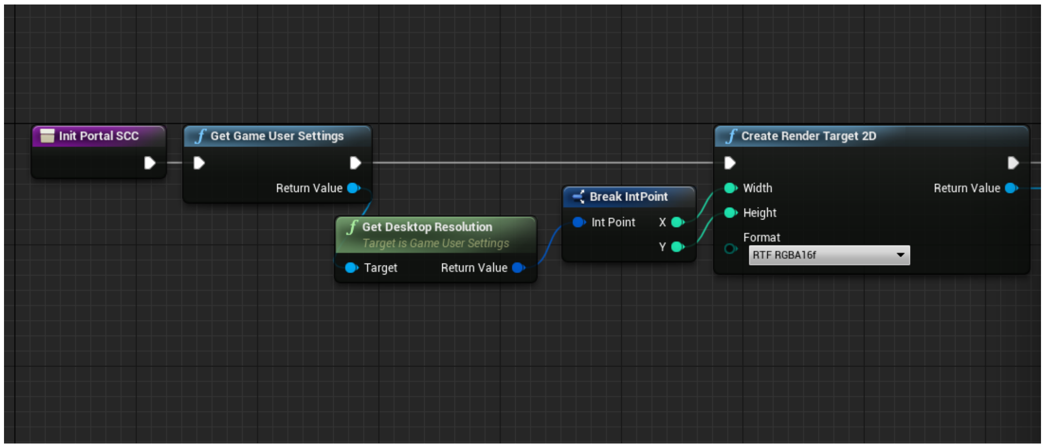
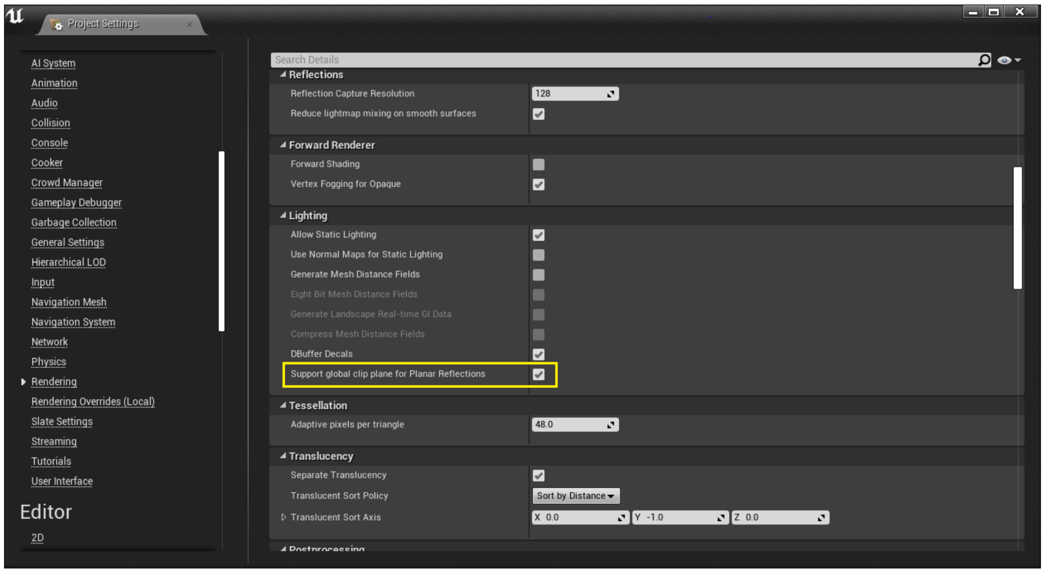


Figure A.3: *Script for setting up render target resolution in Unreal Engine 4. This code runs only at the start of the game (program).*



Reflections	
Reflection Capture Resolution	128
Reduce lightmap mixing on smooth surfaces	✓
Forward Renderer	
Forward Shading	
Vertex Fogging for Opaque	✓
Lighting	
Allow Static Lighting	✓
Use Normal Maps for Static Lighting	
Generate Mesh Distance Fields	
Eight Bit Mesh Distance Fields	
Generate Landscape Real-time GI Data	
Compress Mesh Distance Fields	
DBuffer Decals	✓
Support global clip plane for Planar Reflections	✓
Tessellation	
Adaptive pixels per triangle	48.0
Translucency	
Separate Translucency	✓
Translucent Sort Policy	Sort by Distance
Translucent Sort Axis	X 0.0 Y -1.0 Z 0.0
Postprocessing	

Figure A.4: *This image shows which checkbox that needs be ticked in the Unreal Engine 4 project settings in order to allow near-clip plans to be modified*

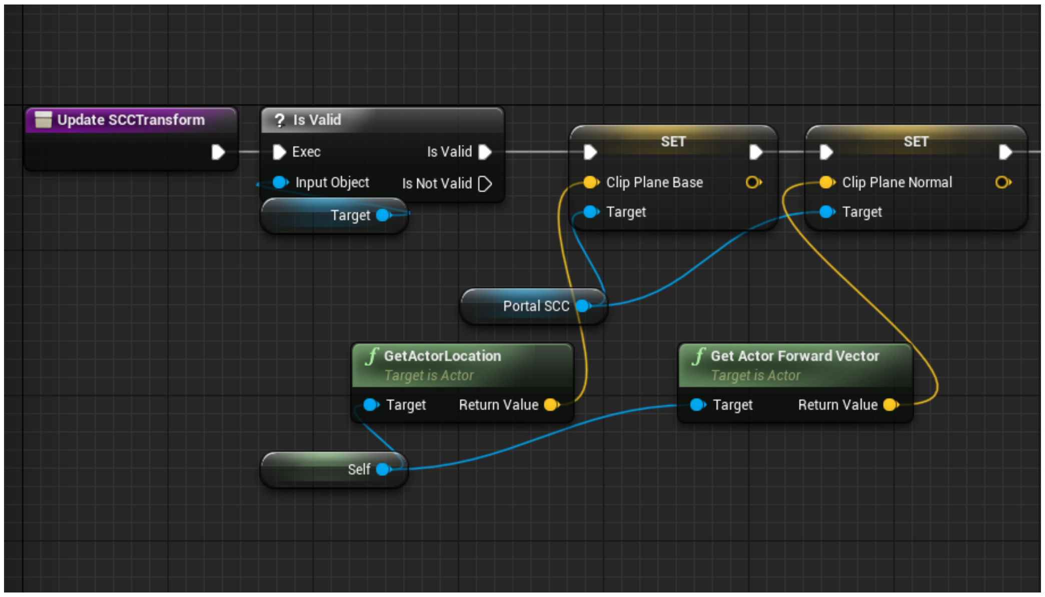
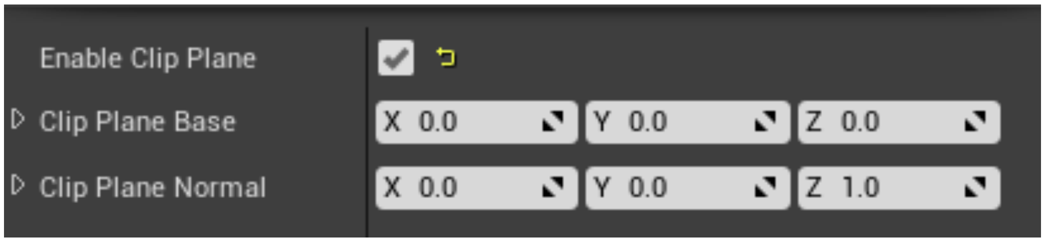
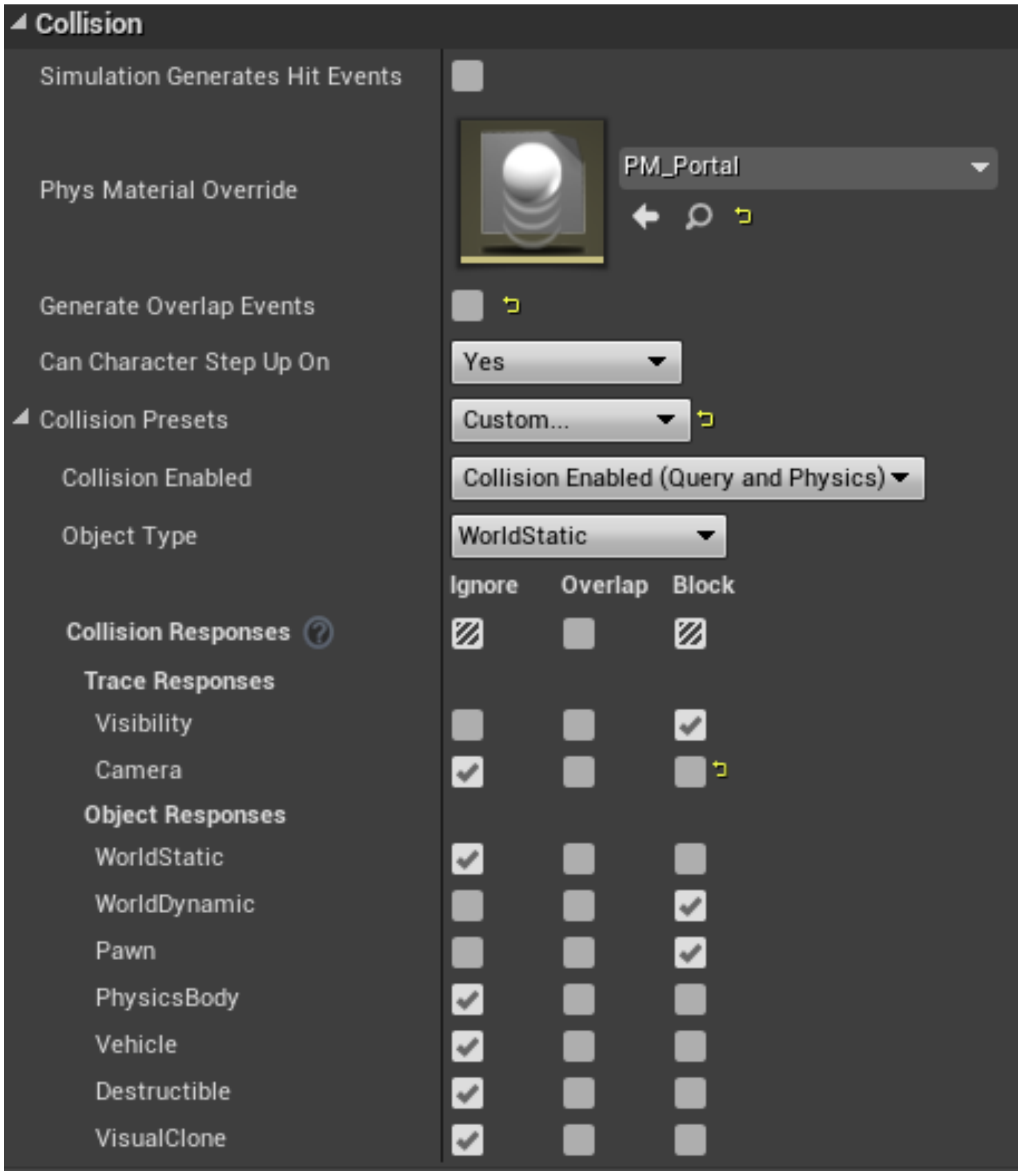


Figure A.5: *The script code (blueprint) for fixing the position and direction of the portal camera's near-clip plane. This is an updated version of the script in Figure 3.11. The function "UpdateSCCTranform" gets called and executes every frame. The function is defined in the portal blueprint class.*



Enable Clip Plane	✓		
Clip Plane Base	X 0.0	Y 0.0	Z 0.0
Clip Plane Normal	X 0.0	Y 0.0	Z 1.0

Figure A.6: *This image shows which tickbox needs to be ticked in order to enable the portal camera's near-clip plane. This setting is found in the "Scene Capture" category of the portal cameras component properties (in the portal blueprint).*



Collision

Simulation Generates Hit Events	☐		
Phys Material Override	PM_Portal		
Generate Overlap Events	☐		
Can Character Step Up On	Yes		
Collision Presets	Custom...		
Collision Enabled	Collision Enabled (Query and Physics)		
Object Type	WorldStatic		

	Ignore	Overlap	Block
Collision Responses	▨	☐	▨
Trace Responses			
Visibility	☐	☐	✓
Camera	✓	☐	☐
Object Responses			
WorldStatic	✓	☐	☐
WorldDynamic	☐	☐	✓
Pawn	☐	☐	✓
PhysicsBody	✓	☐	☐
Vehicle	✓	☐	☐
Destructible	✓	☐	☐
VisualClone	✓	☐	☐

Figure A.7: *This is what the collision settings look like for the portal surface.*

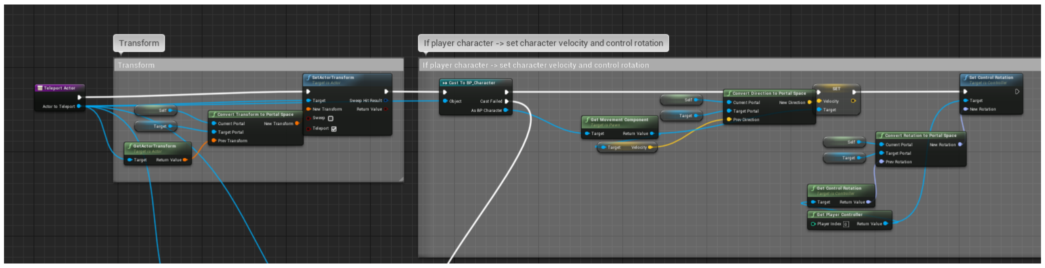


Figure A.8: *This is the blueprint logic (script) used for the "TeleportActor" function responsible for the teleportation of the player character. The function is defined in the Portal class.*

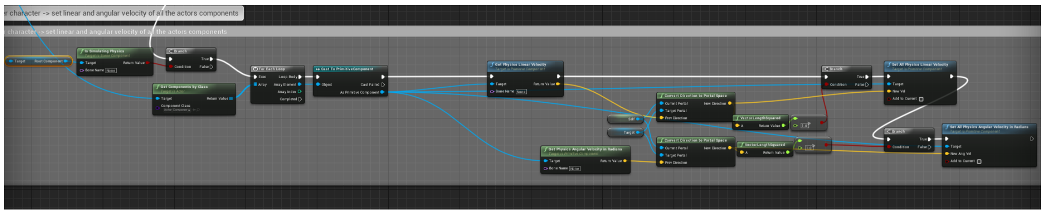

r character -> set linear and angular velocity of all the actors components
Is Simulating Physics
Branch
Get Components by Class
For Each Loop
Cast To PrimitiveComponent
Get Physics Linear Velocity
Convert Direction to Portal Space
VectorLengthSquared
Set All Physics Linear Velocity
Get Physics Angular Velocity in Radians
Set All Physics Angular Velocity in Radians

Figure A.9: *The blueprint logic (in "TeleportActor" function) for teleporting non-player (arbitrary) objects.*

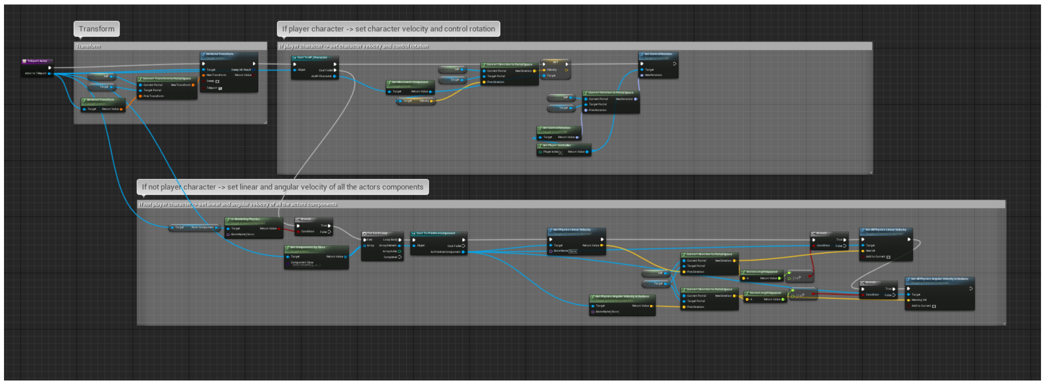

Transform
If player character -> set character velocity and control rotation
If not player character -> set linear and angular velocity of all the actors components

Figure A.10: *The full blueprint logic of the "TeleportActor" function.*

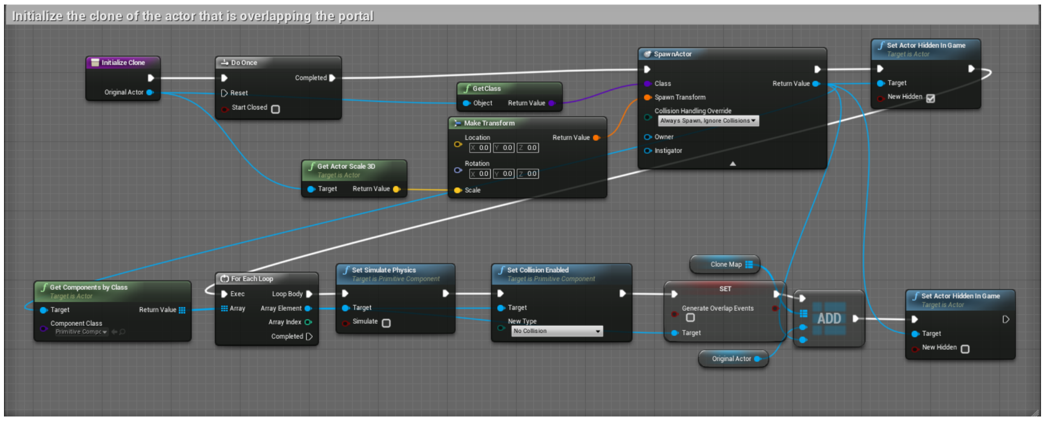


Figure A.11: *The script for the function responsible for initializing a clone instance.*

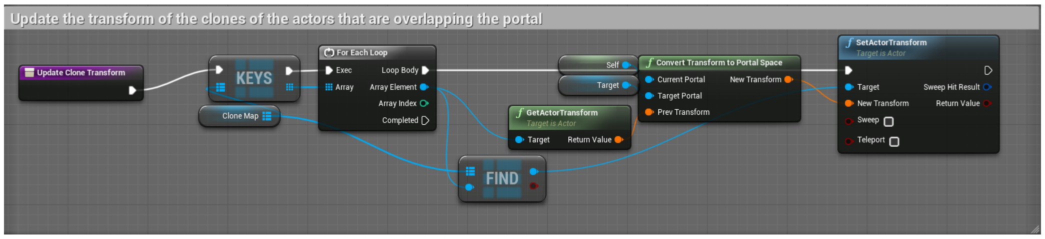


Figure A.12: *The script for the function responsible for continuously updating a clone instance transform.*

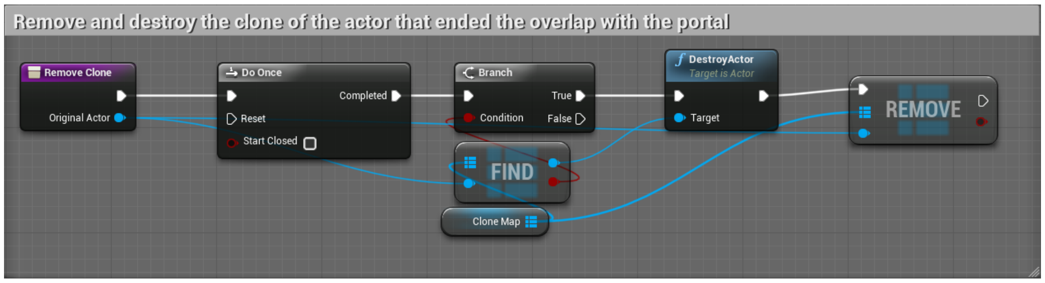


Figure A.13: *The script for the function responsible for removing a clone instance.*

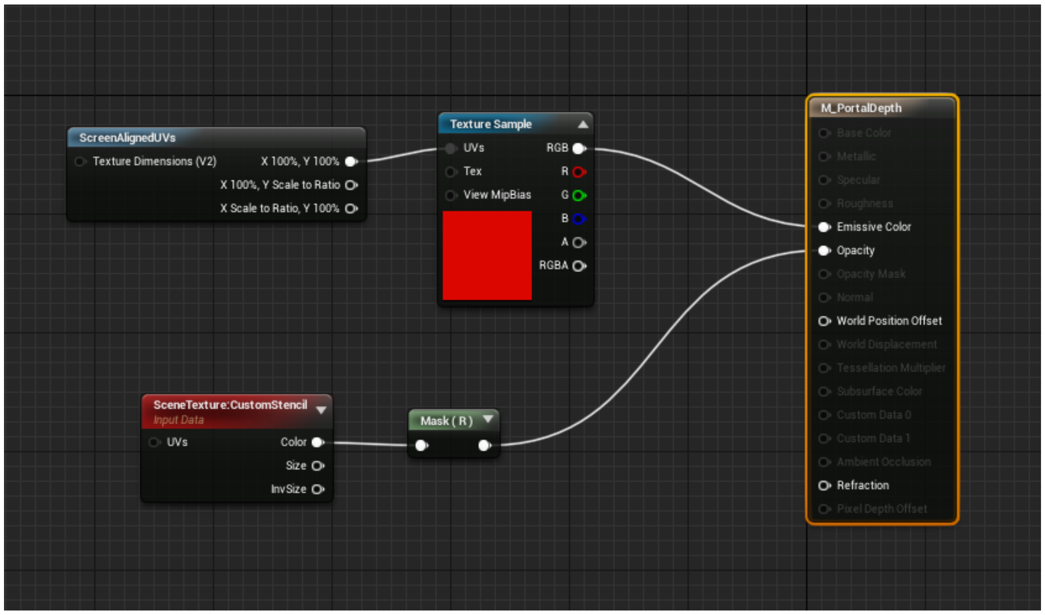
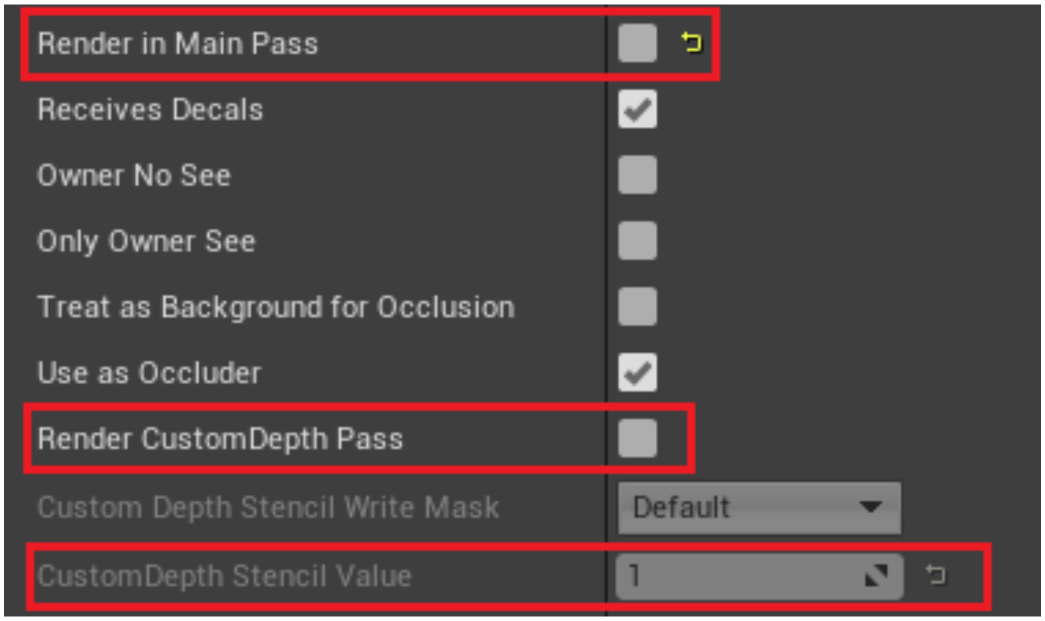


Figure A.14: *This is the shader code of the "M_PortalDepth" material. The opacity value is controlled by the stencil values of the "CustomStencil" render target.*



Render in Main Pass	☐
Receives Decals	✓
Owner No See	☐
Only Owner See	☐
Treat as Background for Occlusion	☐
Use as Occluder	✓
Render CustomDepth Pass	☐
Custom Depth Stencil Write Mask	Default
CustomDepth Stencil Value	1

Figure A.15: *These are the properties of the interior box mesh of the portal. Notice how the "Render CustomDepth Pass" property is false by default as it should only be controlled by in-game logic (when the player character overlaps the portal's surface).*

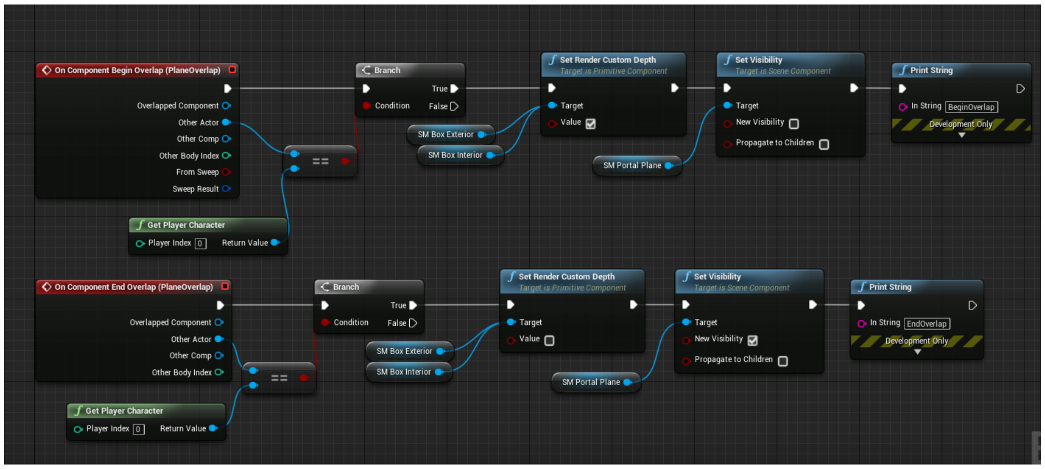
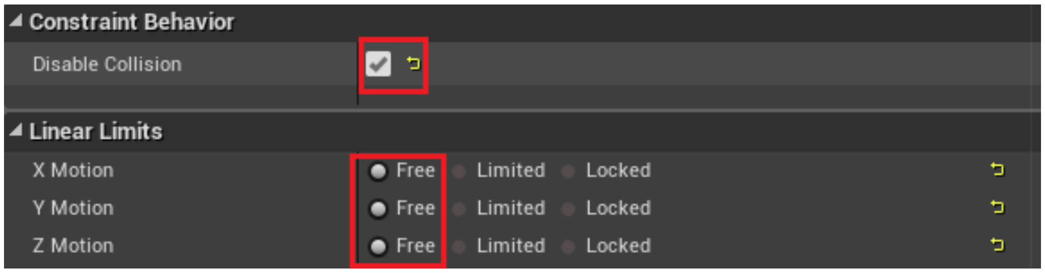


Figure A.16: *This is what is executed whenever an object overlaps or stops overlapping the trigger box.*



Constraint Behavior

Disable Collision ✓

Linear Limits

X Motion [✓ Free] Limited Locked

Y Motion [✓ Free] Limited Locked

Z Motion [✓ Free] Limited Locked

Figure A.17: *These are the properties need to be modified in the "physics constraint component" of the "CollisionDisabler" class to disable the collision between the overlapping and the overlapped object, and to free the overlapping object from any physical constraints.*

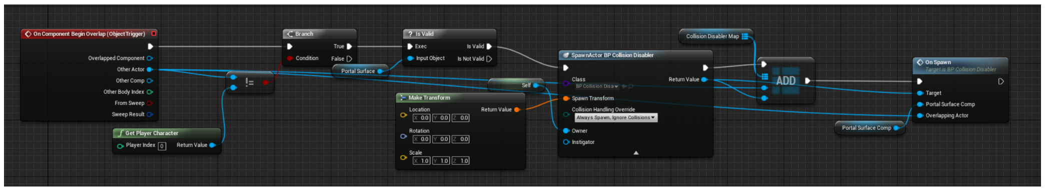


Figure A.18: *The code within the portal class that executes when a object overlaps the portal's surface.*

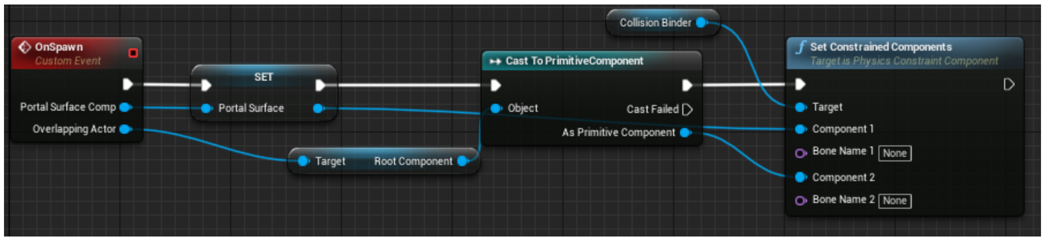


Figure A.19: *The code that the "CollisionDisabler" instance executes when it gets spawned.*

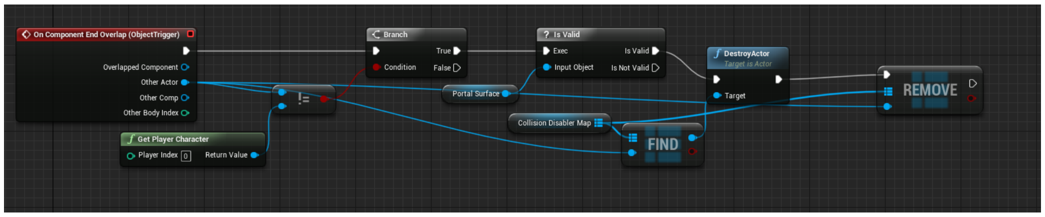


Figure A.20: *The code within the portal class that executes when a object stops overlapping the portal's surface.*

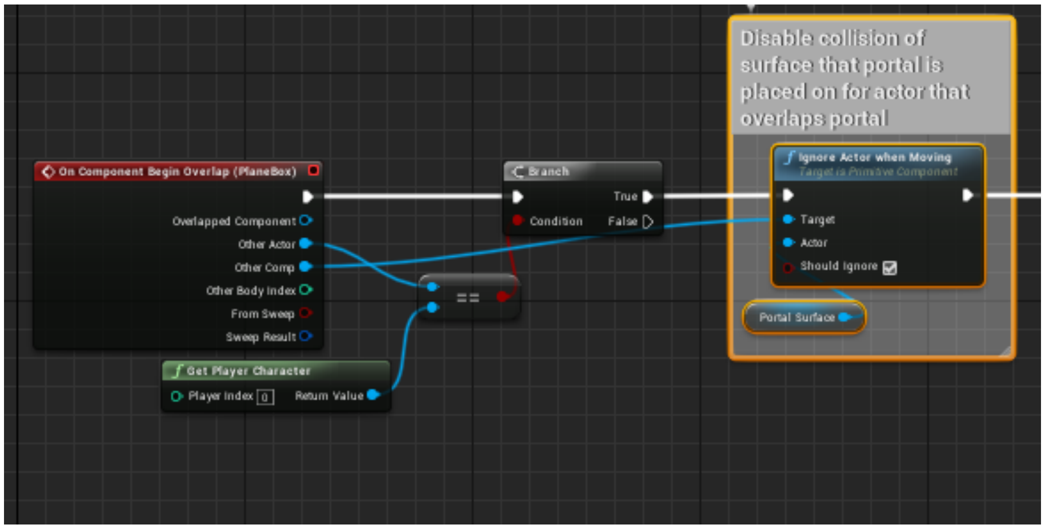


Figure A.21: *This code shows how the "IgnoreActorWhenMoving" function is utilized when the player overlaps the front of the portal.*

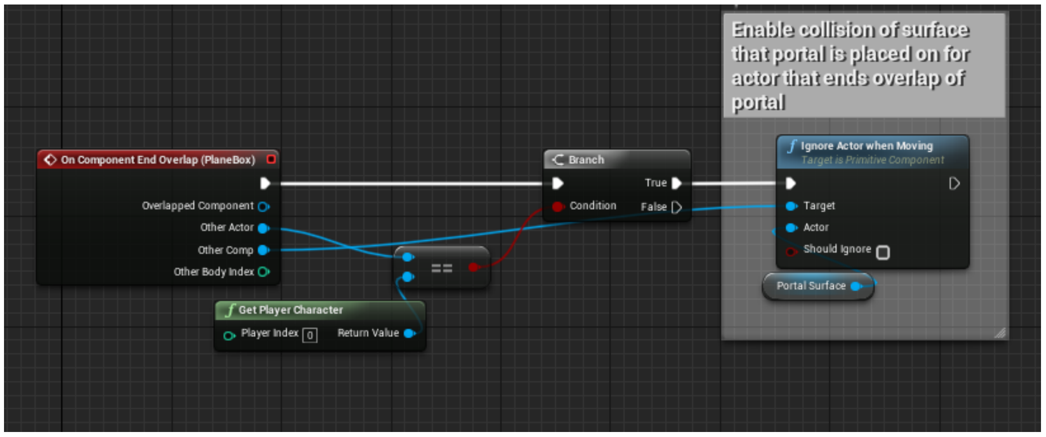


Figure A.22: *This code shows how the "IgnoreActorWhenMoving" function is utilized when the player stops overlapping the front of the portal.*