Non-Euclidean Geometry to Expand Virtual Spaces

Matthew Hullstrung

I. Inspiration

Currently, there is no way to explore a world in Virtual Reality without the use of a player teleporting themselves. I sought to solve this problem with the use of non-Euclidean geometry. The idea was simple, coming from a YouTube video Non-Euclidean Worlds Engine from a channel called CodeParade, but the implementation would be able to immensely expand possibilities for developers building spaces in VR.

Example of non-Euclidean engine:

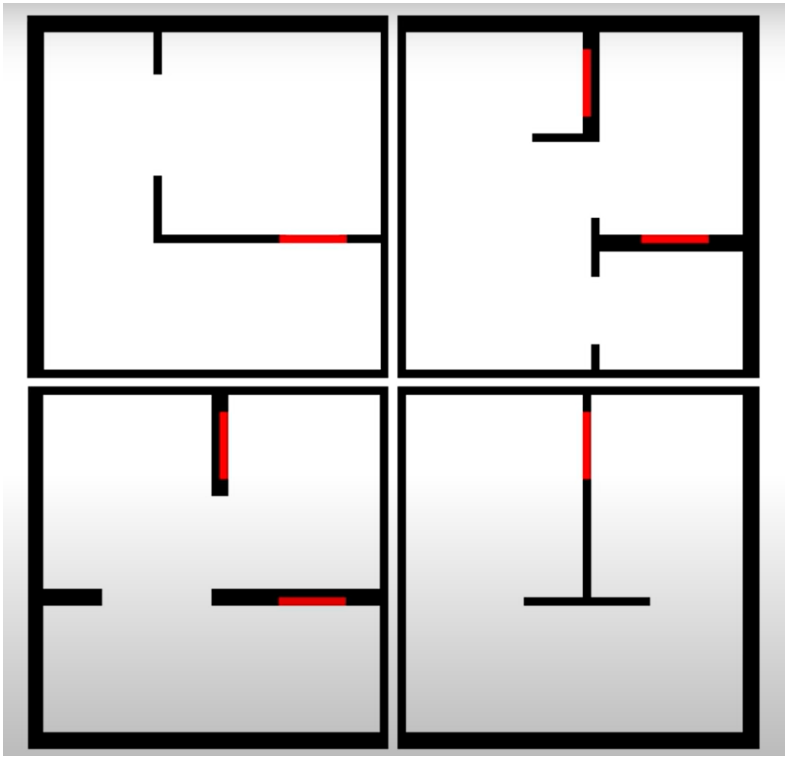


Figure 1.1, https://youtu.be/kEB11PQ9Eo8?t=268

Is compressed into:

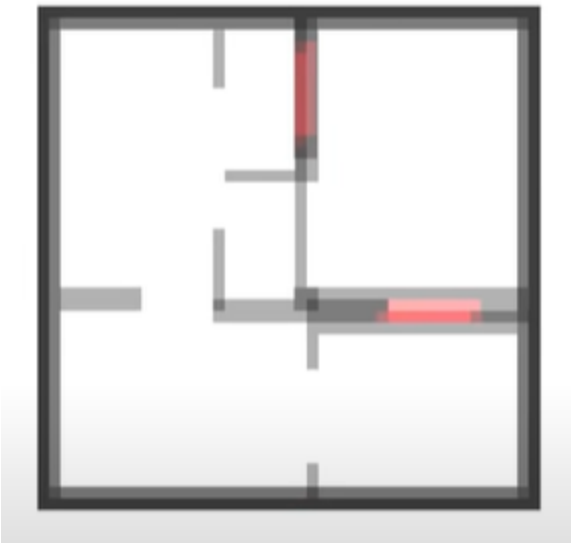


Figure 1.2, https://youtu.be/kEB11PQ9Eo8?t=268

II. Implementation

Though the idea was simple, the implementation could get a little tricky. I decided to use portals to create the non-Euclidean effect we were looking for. The portals themselves would be built from a screen with a render texture that renders from a camera positioned at the portal to which it is connected. However, even with portals, implementation came with its challenges.

A. The Teleportation

To create a working portal, I first needed to figure out how to teleport a character and when to do so. One way to do it is to check on every frame the dot product of the forward vector of the portal with the offset from the player. If the sign of the dot product changes between frames, we have moved to a new side of the portal. An example of this is shown below:

Figure 2.1

B. More Complex Issues

Issues would get more complex as the environments in which these portals were put through would. I had to make sure physics would work properly as objects went through the portal, as well as the issue of not seeing an object on the other side of the portal as it is being pushed through. Currently, with only a teleportation and render texture, you would just see a player model teleport from one side to the other as soon as it reached the middle of the player model. This would not be convincing, as the model should exist on both sides of the portal until it has fully gone through. The ideal way to fix this was to clip/slice the object using surface shaders. The process was as follows: Clone the object on the other side of the portal, do the opposite slice, and when no longer needed (it has fully clipped), destroy the clone. This idea is shown in the figure below, where the circle must exist in two places at once, meaning there is the real object and a clone, both convincingly sliced to show their proper models throughout their travel through the portal:

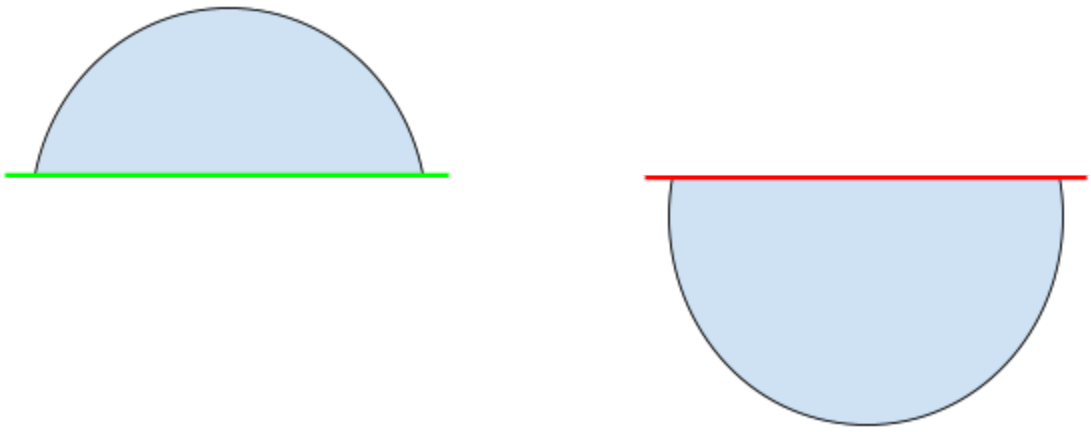


Figure 2.2

C. Optimization

Then comes the issue with optimization. How do we handle recursive portals (portals visible through other portals)? How can we optimize having to render potentially many portals at once? Well, we handle recursive portals by defining a limit, and not allowing the render texture to render if we hit that limit. We can optimize having to render many portals by controlling when we render them, as they only need to be rendered when they are visible from the player's camera. Basically, only enable the render textures of portals visible to the player. An example of this optimization is shown below, with dashed lines being disabled portals and solid lines being rendered portals.

Figure 2.3A

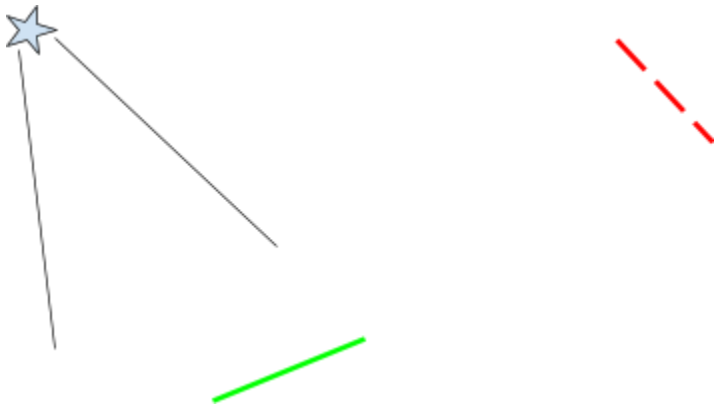


Figure 2.3B

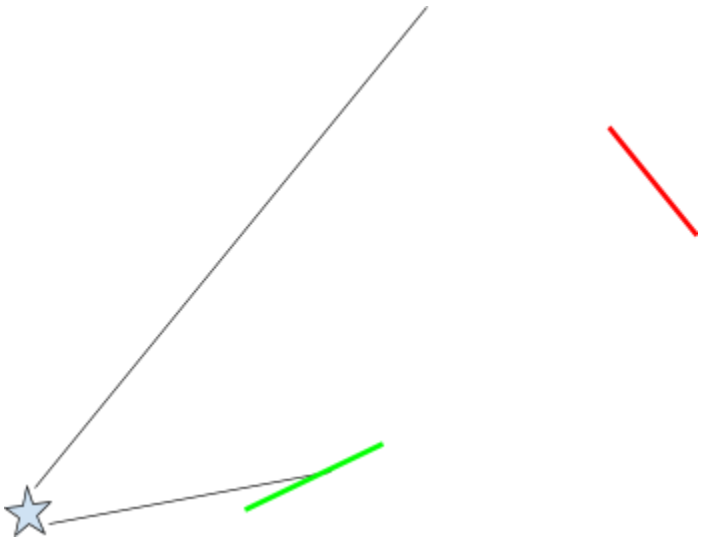


Figure 2.3C

III. Building a Working Demonstration

With the theoretical implementation, I started building a working demonstration in Unity.

A. Scene Setup

The scene is set up as 4 separate environments, each with a different color. The four floors here are the same size to demonstrate a VR play area, though these rooms can theoretically be any size you wish (though if the space is larger than the play area, it defeats the purpose of using non-Euclidean geometry to expand VR play areas). An overhead view of the setup is shown below:

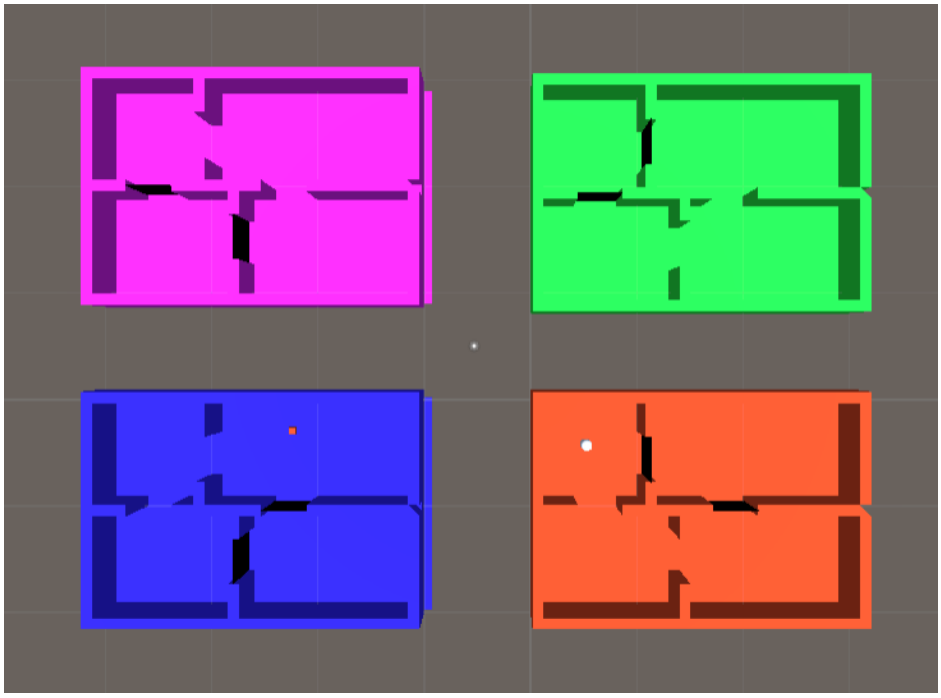


Figure 3.1

B. Portal Setup

Each floor has 4 rooms. Each floor has 2 portals, making one room unreachable (or 3 rooms if you start inside the unreachable room). The color-coordinated flow of portals in terms of their forward vectors is shown below:

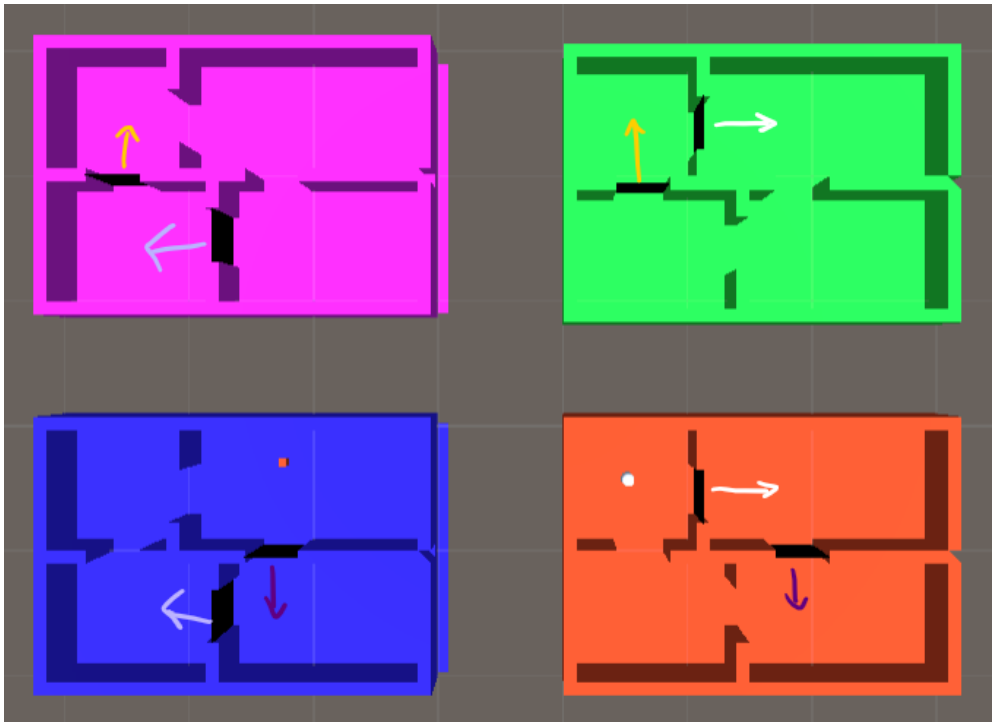


Figure 3.2

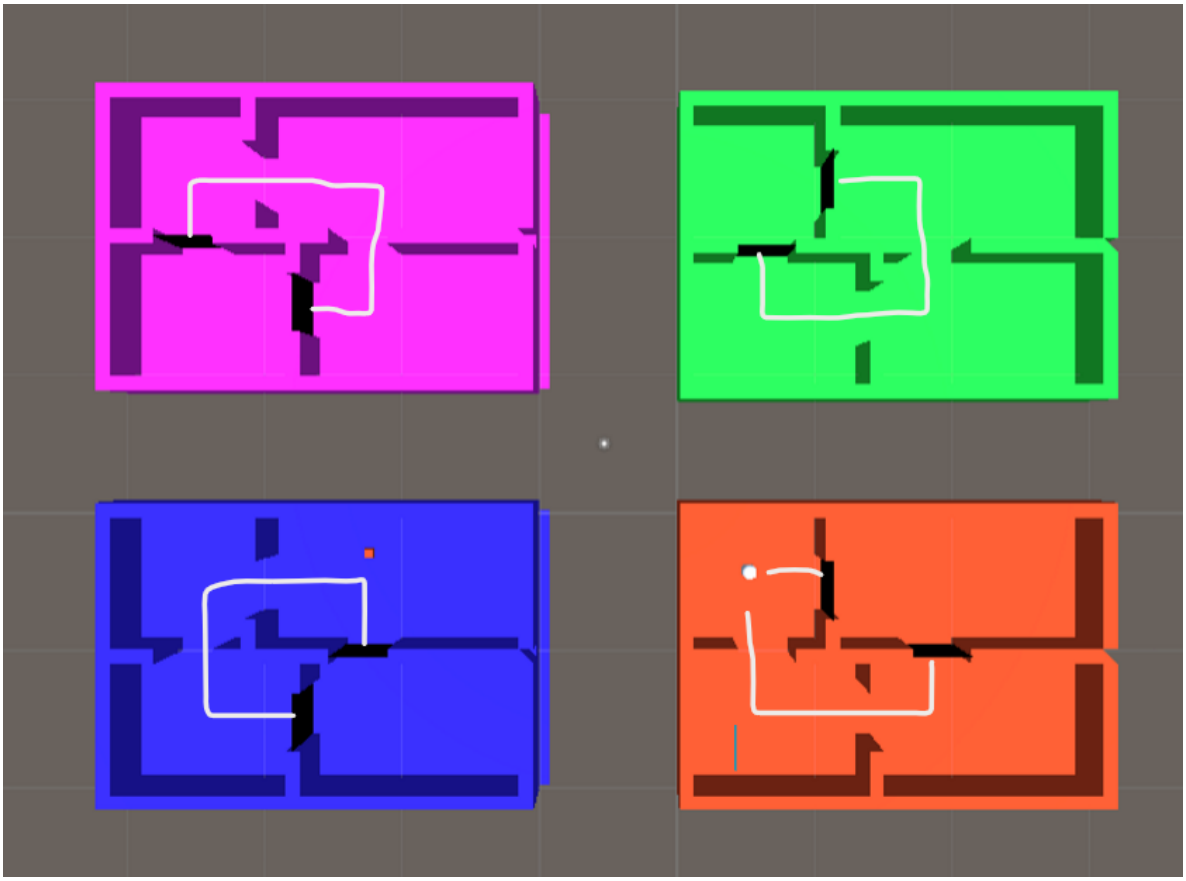
As shown, the “formula” for the placement and linking between portals is to:

> Add 2 portals to a room with 2 doors. This room is now unreachable.
>
> Link portals facing **toward** their floor’s unreachable room to portals facing **out** from their floor’s unreachable room.

This demonstration only shows 4 rooms, but in theory, the number of possible floors is infinite. Adding an nth floor is as simple as using the same formula.

With an infinite number of rooms, a player in VR is now able to reach a theoretically infinite amount of virtual space with a bounded physical play area.

Demo Path



VIDEO DEMONSTRATION LINK

https://youtu.be/udq6UltG_oI

IV. Further Work

I plan to implement more to this concept, mostly for automation and randomization of portal placement. Randomization of portal placement can allow for games such as a non-Euclidean game of tag, where players can randomize or even manually choose which floor they would like their portal to link to. Automation of portal placement would allow for easier development as well as new possibilities for development, overall creating an easier workflow for developers.