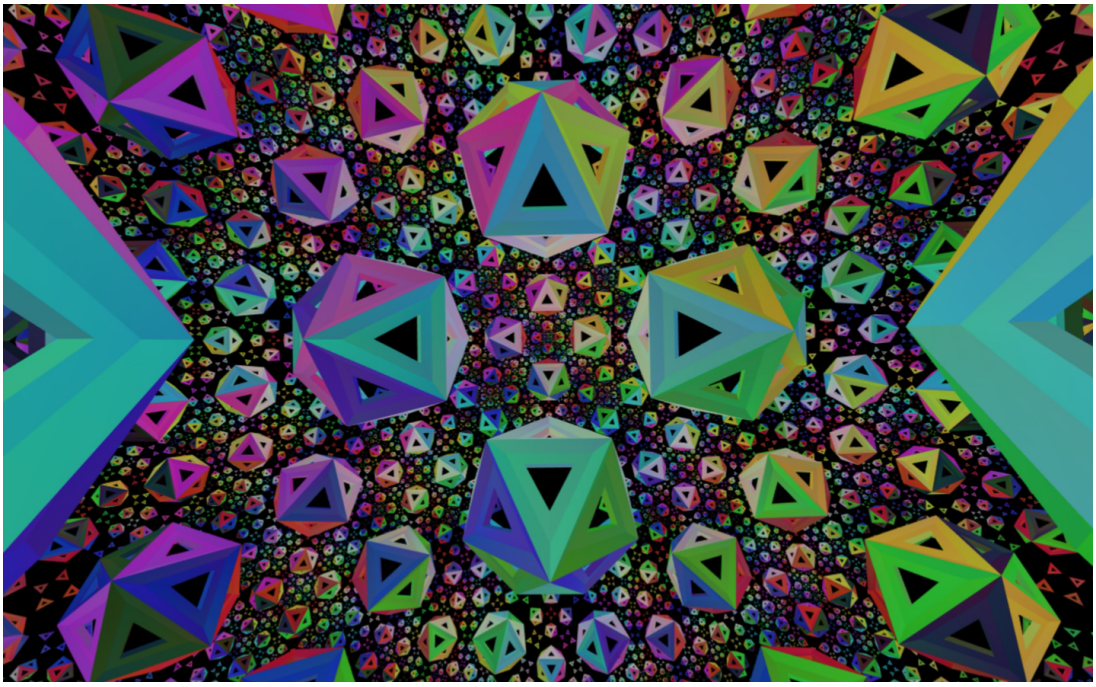
“Holonomy”: a non-Euclidean labyrinth game in virtual reality



The “Holonomy” Team
Baran Ozan Yarar
Bo Bakker
Ravi Snellenberg
Riley Slotboom
Wenkai Li

Client: Martin Skrodzki
Coach: Bart Gerritsen
TA: Nathalie van de Werken

June 2022

TU Delft

Preface

The report is written by a team of five bachelor computer science students at Delft University of Technology. For our software project in the final quarter of the second year, we developed a labyrinth game with hyperbolic geometric features in virtual reality. This report is an assignment for the integrated technical writing module of the software project, and serves to introduce the design and development of the game.

Readers are assumed to have basic technical background. Required mathematical knowledge on hyperbolic geometry will be explained in section 3.2. Considerations on design choices can be found in chapter 3, and actual implementation details are laid out in chapter 4.

We would like to thank our client, Dr. Martin Skrodzki from the Computer Graphics and Visualization Group at TU Delft, for laying out clear requirements for the product and offering consistent valuable feedback during development. We also want to thank our technical writing lecturer, Maarten van der Meulen, for his instructive lectures and constructive feedback on the assignments. We are thankful to our coach, Dr. Bart Gerritsen for his insightful suggestions on the report as well as the game. We are also very grateful to our TA Nathalie van de Werken, for monitoring the progress of the project and giving helpful feedback on game experience.

Delft, 7 June 2022
The "Holonomy" Team:
Baran Ozan Yarar
Bo Bakker
Ravi Snellenberg
Riley Slotboom
Wenkai Li

Summary

Hyperbolic space is a representation of space that is different than what most people are accustomed to. In the past few centuries, it was almost exclusively researched by mathematicians and physicists. But recently there has been increasing interest in visualizing hyperbolic space in virtual reality. It will not only help people understand the concept of hyperbolic space in a more intuitive way, but will also open up new opportunities for experiments in other fields of research.

The goal of this report is to describe the design and development of an interactive VR game that lets a player experience non-euclidean geometry. The implemented game will serve as a prototype for behavioural experiments to facilitate research on navigation and exploration abilities in non-euclidean geometry. These researches can contribute to understanding how orientation is hard-wired in the human brain and whether different modes of orientation can be learned.

The game was created based on a list of requirements that outline what features the game should have in order to serve its research purposes. The most important requirements are hyperbolic room connections, aligning physical walls, and displaying a minimap. Firstly, the rooms are connected in a hyperbolic way, to avoid the need to implement a full hyperbolic render engine. This means that the rooms themselves use euclidean geometry, but the player needs five turns to return to their original location. Secondly, the walls in the virtual world are aligned with those in the physical play area, allowing the player to reach out to a virtual wall in front of them and feel the corresponding physical wall. Lastly, a map is shown on the player's HUD to help them visualize nearby tiles and locate desired items. This map uses the Poincaré disk model, a way of projecting a hyperbolic space onto a flat surface.

Though all of the above has successfully been implemented, there is still a lot of room for improvement. In particular, some optimization can still be done to improve the performance of the game. Next to that, many mechanics can still be generalized to be more flexible. For instance, the tile grid could be extended to have a variable size, or the hyperbolic tiling could be extended to have more than five rooms around each vertex.

Contents

Chapter 1

Introduction

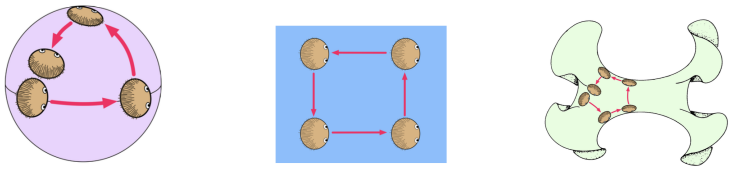
Hyperbolic space is a different representation of space than the euclidean space that people are familiar with, but what does "different" mean exactly? For example, hyperbolic space is curved such that two parallel lines would diverge from each other, rather than being equidistant like in euclidean space. Unique properties of hyperbolic space like this are described by hyperbolic geometry, a term first coined in 1871 by Felix Klein [1].

However, it remains challenging to understand the unique hyperbolic geometry without proper visualizations. Over the years, efforts have been made by computer scientists to render and visualize hyperbolic space on a screen [2]–[4]. These visualizations turned out to be not only educative but also entertaining, such that it sparked interest in game developers as well [5].

In more recent years, there has been increasing interest in simulating curved spaces in virtual reality [6]–[8]. Virtual reality not only provides a more immersive visual experience, but also allows users to walk around in this virtual hyperbolic world by tracking their movements in the physical world. Therefore, it is possible to provide an even more interactive way to explore the non-euclidean world.

The focus of this project lies on simulating one particular property of hyperbolic geometry in VR: "holonomy". Holonomy is a property that manifests when people travel in a loop in a non-euclidean space, which will cause them to be rotated away from their starting orientation (see Figure 1.1). When holonomy is utilized properly in a virtual hyperbolic world, it would allow people the possibility of traversing arbitrarily far in the virtual space while being confined to a finite space in the real world.

The goal of this report is to describe the design and development of an interactive VR game that lets a player experience non-euclidean geometry. To accomplish this, a list of requirements was elicited from the client, Dr. Martin Skrodzki, using the MoSCoW method. It was then reviewed by the client again to ensure concrete and unambiguous goals for the project. After agreement on the requirements, research was done into how the goals can be accomplished and an outline of the game structure was designed. Following that outline, the game was implemented using the Unity game engine and HTC Vive Pro 2 VR headset.



(a) Positive Holonomy (b) Zero Holonomy (c) Negative Holonomy

Figure 1.1: Illustration of the 3 types of holonomy [8, Fig. 1]

The purpose of the game is to serve as a prototype software for behavioural experiments that can help researchers understand navigation and exploration behaviours in the alien non-euclidean space. For this purpose, the use of VR is essential since it can create a highly immersive experience of navigation in the virtual world, compared to using only a screen. Allowing players to directly walk to the goal in a straight line would also defeat the research purpose since the person would not actually need to navigate or explore anything at all. For that reason, together with the need of running the program with limited physical space available, the labyrinth setup is introduced that confines the physical space to a finite bounded grid. There are existing games that utilize features of hyperbolic space, but none of them satisfy all of the requirements (see appendix C.1).

The report starts with chapter 2, which will give an overview into the goals and context of the project. With that in mind, chapter 3 will dive more into what exactly it means for a space to be hyperbolic and the mathematical foundations behind it. Design and ethical implications of the game are also detailed in this chapter. Chapter 4 will then go into the implementation details of the features that were designed for the game. Chapter 5 will go over alternative design choices (and why they were not followed) and possible avenues of further expansion of this project. Conclusions and key takeaway points are laid out in chapter 6.

Chapter 2

Exploring Hyperbolic Space in Virtual Reality

Virtual reality makes it possible to explore non-euclidean spaces and experience the unique phenomenons with great immersion. This chapter will introduce the hyperbolic context and lay out the goals of the project. Section 2.1 gives an introduction to two unique properties of hyperbolic space and an example of how these properties can be utilized in the project. After that, section 2.2 will describe the goals that were set for this project. Section 2.3 explains how those goals were formalized into requirements.

2.1 The Unique Properties of Hyperbolic Space

Hyperbolic space brings many unfamiliar phenomena. This section will introduce two interesting properties that people would experience in hyperbolic space, and give an explanation to the holonomy property utilized in the project[1].

The first interesting phenomenon one would notice in hyperbolic space is that objects in the distance appear much smaller than they actually are. They seem to be "squished" together. This effect also occurs in euclidean space, but it is exponentially magnified in hyperbolic space. For example, a ten-meter-high tree seen from only a few meters away might not appear much bigger than a person, given that the curvature of the space is sufficiently large.

In hyperbolic space, light appears to curve outwards, rather than travelling in a straight line (see Figure 2.1). This is not because light behaves differently than in euclidean space, but because it follows the curvature of hyperbolic space. At the same distance, eyes can capture exponentially larger areas when light curves outwards, compared to when light travels in straight lines. An object in hyperbolic space would therefore have a much smaller projection area on the eyes.

The second unusual phenomenon of the hyperbolic space is called holonomy [8]. If a person makes four consecutive 90° right turns, they will not find themselves in the original position. In hyperbolic space, pentagons[2], rather than squares, have 90° angles. When making 90° turns in the physical play area, a person follows the edges of a pentagon in hyperbolic space. Therefore, in order to return to their starting position, they would need to take a fifth 90° turn. This yields a total rotation of 450°. Thus, in the real world, they are now facing a different direction from when they started.

The unique property of holonomy enables people to travel arbitrarily far in a hyperbolic virtual space, while walking in limited physical space. This is achieved by repeating the holonomy process. First, a person needs to make four 90° turns in the same direction. This brings them back to the physical starting point, but gives an offset of one step from the virtual

[1] For more insight into the underlying mathematics, see section 3.2

[2] In a space with larger curvature than that is used in the game, it would be other types of regular polygons (e.g. hexagons) that have 90° angles

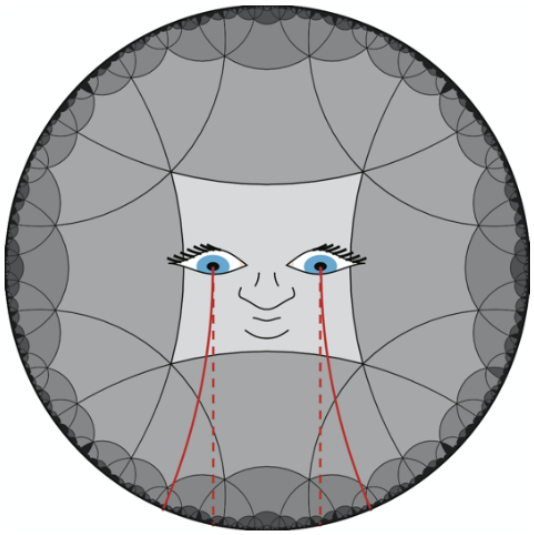


Figure 2.1: Light rays diverge in hyperbolic space [7].

starting point. Then, a person can accumulate virtual offsets in all directions by repeating this process in different directions. This allows a person to walk infinitely far in the hyperbolic space, as long as the physical space allows making four 90° turns in all directions. The simplest setup for this, would be a 3x3 grid of squares.

2.2 Description of the Problem

The problem laid out by the client consisted of 5 steps:

1. **The virtual grid**
   The first step will be to create a virtual representation of the physical play area. This will consist of a 3x3 grid of virtual rooms, corresponding to a 3x3 grid of squares in physical space. In the virtual space, walls will be rendered as boundaries and will always align with the walls in physical space.

2. **The non-euclidean grid**
   The virtual grid will be extended to have 3 different geometries available: euclidean, hyperbolic, and spherical. In the original grid, the user must make four 90° turns to end back where they started. In the hyperbolic grid, they need five 90° turns. In the spherical grid, they need three.

3. **The map**
   There will be an option to enable an in-game map of the virtual space to help players navigate in the non-euclidean space. This map shows a top-down projection of the room the player is currently in, as well as nearby rooms.

4. **Navigation tasks**
   The first game mode is navigation. In a navigation task, the player must find a way from their starting room to a destination room. The distance from the starting room to the destination room can be configured by the operator[3] of the game before game starts.

5. **Exploration tasks**
   The second game mode is exploration. In this mode, the player must find a number of

[3]For example, the behavioural researchers.

keys that are scattered within a certain distance from the starting room. Once all keys are collected, the player must return to the starting room, where they will use the keys to open a chest. Again, the distance from the starting room to each of the keys can be configured, as well as the total number of keys to be found.

2.3 Requirements Elicitation

Requirements for the product are elicited through in-depth discussion with the client. First, we introduced our understanding and envisioning of the product to the client. Second, we proposed the priorities of each feature, based on the five steps in the project statement. Lastly, we confirmed whether our ideas and expectations aligned with those of the client and adjusted accordingly. Research on the feasibility of the product was also conducted before the discussion. In this way, we were able to explain to the client with more confidence what is or is not technologically feasible.

After the first meeting, we drafted the list of requirements using the MoSCoW method and presented it to the client for final review. Each individual requirement was discussed and received approval from the client in a second meeting. The complete list of approved requirements can be found in Appendix A. The eight "Must Have" requirements are listed below.

1. The virtual space must be based on a physical grid of 3 by 3 squares.
2. The game must be able to connect rooms according to the square tiling of the euclidean plane or the order-5 square tiling of the hyperbolic plane.
3. The game must be able to map physically present walls into virtual space and align accordingly.
4. The game must start at a marked starting square in the physical grid.
5. Players must be able to navigate between rooms in the virtual space by moving around the physical space.
6. The player must have a map in their HUD that represents the location they are in, in the virtual space using the Poincaré disk model.
7. A room must have one transparent passage to each adjacent room corresponding to a physical square.
8. The game must have a main menu for access to the game, and an in-game menu for customization during gameplay.

Chapter 3

Designing a Hyperbolic Labyrinth Game in VR

This chapter aims to outline key design choices of the game. In section 3.1 our method for simulating non-euclidean space will be explained. The next section (3.2) goes over tilings and representations of the hyperbolic plane, as well as the graph generation and navigation algorithms designed for the game. Next, three approaches to the generation of the different levels in the game are explained (3.3). Section 3.4 will explain how the player is able to interact with virtual space by interacting with physical space. Finally, section 3.5 details the ethical implications of the game.

3.1 Non-Euclidean Connection, Euclidean Rendering

The game is developed to research human navigation abilities in a non-euclidean virtual space. For this purpose, rooms need to be connected as if they are on a hyperbolic plane, to realize the holonomy property. On the other hand, rendering full-blown non-euclidean geometry is not necessary, and even undesired for the purpose. Non-euclidean rendering, especially in VR, can present players with many distorted visual effects and create an unpleasant experience[1].

Implementing the room connections instead of fully simulating non-euclidean geometry still allows the player to experience the effect of holonomy. This approach fully satisfies our purpose, while leaving out unnecessary hyperbolic visual effects that can cause discomfort.

3.2 Mathematical Prerequisites

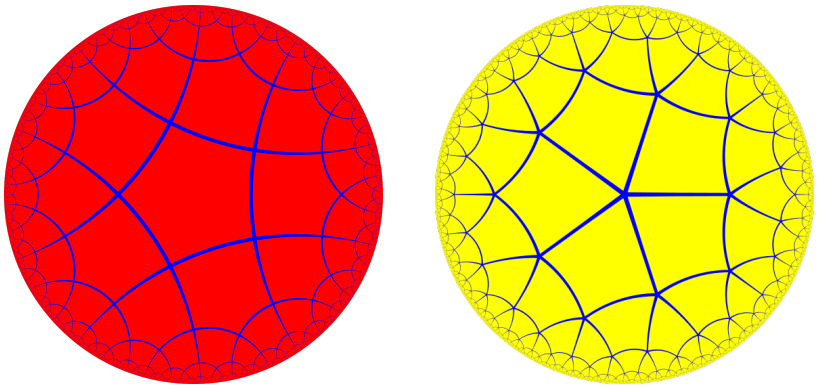
Even without the need to fully simulate hyperbolic geometry, there are substantial mathematical hurdles that have to be cleared in order to implement the game. First, the rooms are laid out according to a tiling of the hyperbolic plane, and representing the room connections is not straightforward. Moreover, in order to render the minimap, precise mathematical representation of the hyperbolic plane is still required. The mathematical knowledge needed to understand the design of the game are explained in this section.

3.2.1 Tilings of Hyperbolic Plane

There are an infinite number of ways to tile the hyperbolic plane with regular polygons. It can be tiled with any set of regular polygons so long as $\frac{1}{p} + \frac{1}{q} < \frac{1}{2}$, where p is the number of sides of the polygon and q is the number of polygons that meet at a corner.

[1] See [6] for an example of rendering light rays in hyperbolic space.

The two most important tilings for the game are the Order-4 Pentagonal Tiling and the Order-5 Square Tiling [9], as seen in Figure 3.1. Virtual rooms in hyperbolic space are represented as vertices in Figure 3.1a, and squares in Figure 3.1b.



(a) Order-4 Pentagonal Tiling (b) Order-5 Square Tiling

Figure 3.1: Tilings of $\mathbb{H}^2$, with rooms as vertices of (a) and faces of (b)

3.2.2 Models of Hyperbolic Space

In order to represent coordinates in hyperbolic space, the Minkowski hyperboloid model is used. In this model, points on the hyperbolic plane are represented as points on a hyperboloid in three-dimensional Minkowski space [10, pp. 12, 54–99]. This space has three coordinates: x, y, and t, with the hyperboloid satisfying the Minkowski quadratic form: $\{(x, y, t) : t^2 - x^2 - y^2 = 1\}$.

Converting this representation to the Beltrami–Klein model [10, p. 7] or the Poincaré disk model [10, p. 8] is straightforward. Since the hyperboloid in Minkowski space is analogous to the unit sphere in three-dimensional Euclidean space [10, pp. 61-72], a simple stereographic projection can be applied to the hyperboloid to obtain the Poincaré disk model [10, pp. 122-130], or a gnomonic projection to obtain the Beltrami–Klein model [10, pp. 188-194]. The Poincaré disk model is used to render the minimap.

3.2.3 Graph Generation and Navigation Algorithms

In order to connect the rooms according to a hyperbolic tiling, a data structure representation of the tiling needs to be constructed. This is an especially important design choice, because it directly determines the difficulty of implementing other features. The following approach was adapted from [11], with one addition to the reduction rules.

First, the Order-4 Pentagonal Tiling is chosen as the underlying representation, where rooms are represented as the vertices. This allows easy access to a room through index of the vertex. A spanning tree structure (see Figure 3.2) is used to represent the tiling, because tree structure can guarantee a unique shortest path to any node.

Next, every room needs to be identified with a unique index[2], which can be generated by taking the (shortest) sequence of steps needed to get there. Discrete representation is preferred, because numeric representations, such as coordinates, can introduce serious precision issues, especially with the hyperbolic setup. This unique index will be used to link a game object and a color to each location.

Representation of the step sequence is as follows: rooms are first indexed by a choice of branch (North, West, South or East), and then by a choice of directions (Forward, Left or

[2]This is necessary in order to implement other required features, since specific properties like colors or items need to be associated with specific rooms.

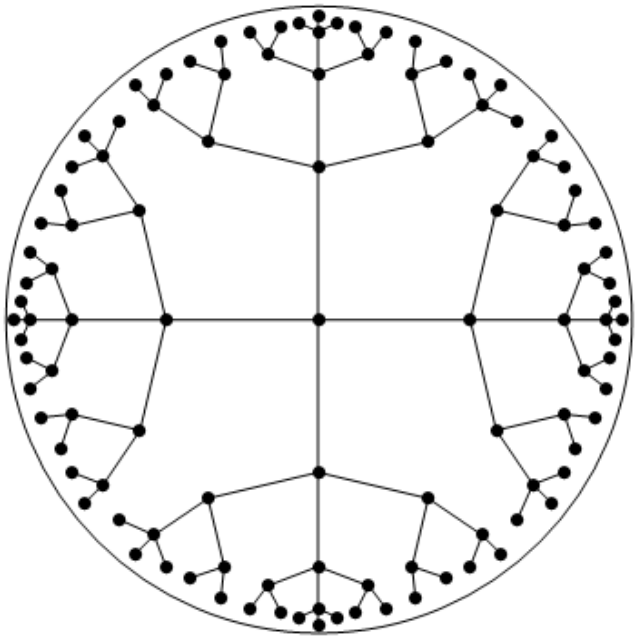


Figure 3.2: The spanning tree of the graph of room connection [11]

Right), with moving backwards corresponding to deleting the last step. Including an actual backward step can give multiple valid indices to a room and thus the uniqueness of room indices will be violated. Therefore, the first step needs a different representation since there are four valid directions of movement from the origin.

In order to ensure that the sequence of steps is unique, two algorithms are designed. The first is used to determine whether a step is valid. That is, it determines whether a step stays on the current branch of the spanning tree rather than moving to a different branch. The other algorithm is used to normalize an invalid sequence of steps into a valid one.

The validation algorithm works as follows. A forward step is always valid. A right step is invalid if and only if the previous step was also a right step. A left step is invalid if and only if there is no right step in between it and the last left step.

The normalization algorithm applies the following transformations:

$$\{x, R, R\} \to \{r(x), L\}$$

$$\{x, L, L\} \to \{\ell(x), R\}$$

$$\{x, \{R, F\}_n, R, R\} \to \{r(x), L, \{F\}_n\}$$

$$\{x, L, \{F\}_n, L\} \to \{\ell(x), \{R, F\}_n, R\}$$

where $r(x)$ rotates the previous step, either branch or direction, to the right, and $\ell(x)$ is rotation to the left. This normalization ensures that the sequence of steps corresponds to a connected and unique series of edges of the spanning tree, starting at the point of origin.

Besides the graph generation algorithms, it is also necessary to indicate directions on the minimap. In order to do this, the current location of the player, as well as the location of the target room, are converted into Minkowski hyperboloid coordinates. Using these coordinates, the direction can be simply found by taking the x and y coordinates of both points and computing the angle between these two points in two dimensions. Since the angle is only used as a visual indicator on the minimap, it does not have to be very precise. Therefore, the usual mathematical inaccuracy of this process (when the points are very far away from each other) is not an issue.

3.3 Level Generation

The game can generate a level in non-euclidean space in 3 ways: manual level creation, parameterized generation, and procedural generation using a seed.

Manual level creation is possible because the game loads its files from a JSON format. This is an easily readable format that follows a clear structure, making it easy to edit manually. In this structure a location is linked to an objective. By creating a JSON file and writing locations and objectives in accordance with the structure it is possible to manually create save files for the game.

But parameterized generation is still indispensable for this game as manual creation would take a lot of time for bigger levels. It can quickly generate similar levels by inputting desired parameters or constraints. Level generation works for both navigation and exploration task.

For navigation tasks, the operator (the person who sets up the level) can decide that the flag the player needs to find should be placed within a radius of x virtual tiles away. For exploration tasks, the operator can choose to place x number of keys between y and z tiles away from the starting point. The program will create a save file of the level after it is generated. Loading and using these generated levels is thus also possible at a later date.

It is also important to be able to load a generated level. The game is able to take a save file and output the level that was saved. The save files are designed to be very lightweight, containing only the locations of the cubes that have a special feature. At run-time, the game can spawn the corresponding object when needed and treat all the location that are not in the save file as a default cube.

Another option is to lazily generate the level around the user. This can be done by providing a seed and a custom function (which can be modified to satisfy the given input) that can use the unique identifier of a room to decide if an object should be placed. With this approach, new objectives can be generated on the fly. This has the benefit of making the virtual space seem truly infinite, though the operator no longer has precise control of object placement.

3.4 Integration with Virtual Reality

There are three steps in the integration with virtual reality: tracking the player's physical movement, detecting boundaries, and enabling interaction with virtual objects.

Firstly, when the player moves, the VR headset will track their position on a physical grid, and this will allow the player to move between rooms. When a player moves from one square to an adjacent square, the player will move into the next room in the game. The game matches a virtual representation of the physical grid against its own representation of the hyperbolic space.

Secondly, when the player reaches the edge of the physical grid, the game will erase passages between rooms and place a wall instead. The same will happen whenever there is a physical barrier between two squares in the grid. By aligning virtual walls with physical walls this way, players will have a more immersive experience, and will be less likely to hurt themselves by unwittingly running into a physical obstacle.

Finally, the player is able to interact with objects by proximity. If a player reaches a flag, they will automatically complete the corresponding navigation task. If a player stands near a key, it will be added to their inventory. If a player stands near a chest, and has the corresponding key, the chest will open automatically.

3.5 Ethical Implications

The game is to serve as a prototype for behavioural researches that will involve human participants, and may even be publicly released in the future. In other words, we may not have full control over how our product will be used by end users. Therefore, we have to take active responsibility [12] towards the potential ethical issues during development stage, and try to

address every possible concern while we can. In the following subsections, we discuss our concerns and design values regarding human participants who will be interacting with our project, those regarding the environment, health and safety of the users and those for the personal data belonging to the participants.

3.5.1 Human Participants

Human participants will be involved in the use of the game, since it serves as a tool for behavioral experiments. It is important that these participants are aware of what the game entails. Otherwise, the unfamiliarity with the non-euclidean virtual space can come with the issues outlined in 3.5.2.

During development stage, participants outside the development team may be involved to test, give feedback on and validate game design and implementation choices. When the development process is complete and the game is ready to be used by the researchers, the participants to their behavioral experiments will also play the game. In either case, the participants will be volunteers and there will not potentially be any patients, vulnerable individuals or minors.

Alternatively, the game may be publicly released, if it turns out to be enjoyable also for people beyond experiment participants. In that case, the users will still likely be volunteers, but there can be no guarantee that they do not potentially belong to certain vulnerable groups. All the game objects used in the game will made sure to be neutral-looking and suitable for all potential users.

3.5.2 Environment, Health and Safety

The planned activities in this project does not really challenge or affect societal values, nor does it manipulate the physical environment. However, it involves manipulation of the virtual environment that players experience. The game transforms a 3x3 grid in physical space to an infinite labyrinth in virtual non-euclidean space, with the help of VR technology.

As people may not be used to non-euclidean or virtual spaces, manipulating their observed environment may cause slight issues. These issues include but are not limited to: motion sickness [13], stress, confusion and so on. People with certain conditions, such as epilepsy, may also be vulnerable to visual effects in virtual reality [14], especially effects developed without taking such conditions into consideration.

To address these issues, the best practice of room scale VR development will be followed to minimize the effect of motion sickness and other sources of discomfort. Extra care will be taken to not include any stimulating visual effects in the game, to reduce risk of seizure and other conditions for potentially vulnerable users.

In addition to dealing with the issues above, the team is also responsible for making sure that the players do not hurt themselves due to obstacles in the real world. This is done by aligning the virtual walls with real walls in the physical space, so the player can view and navigate around the walls without needing to remove the headset.

3.5.3 Personal Data

Collection of personal data is another concern over the research use of the game. Personal data of the participants may be collected by researchers in order to categorize participants into different groups and carry out more detailed analysis. This can however lead to biased conclusions, and this possibility will be minimized during development.

In the game, certain metrics, such as time and number of rooms travelled, are logged for researchers to analyze and evaluate the performance of the participants. To address the personal data concern, the game provides no means of collecting participants' personal data that can be used to identify them. The only data that may be collected are the performance metrics when playing the game and are strictly for research purposes (or for entertainment if

the player likes to see how well they performed). The potential risk of collecting too much or misuse of users' personal data will be reduced to the minimum.

Chapter 4

Implementing the Labyrinth Game

This chapter aims to offer more insight into the approaches adopted in the actual implementation of the game. Section 4.1 details the underlying data structures that allow the game to represent a hyperbolic space. Once a game level is started, there are certain objectives players need to achieve. Section 4.2 goes into the objectives of navigation and exploration task. To complete the game tasks, players need to interact with game objects, such as collecting keys to open a chest. Section 4.3 will explain how the interaction with game objects works in VR. Section 4.4 introduces the loading and generation of game levels. Section 4.5 introduces the user interface of the game, including menus, metrics and minimap display. Finally, section 4.6 will introduce the details of rendering a minimap.

4.1 Representing the Hyperbolic Space

The Unity game engine is not built to support non-euclidean space, as well as most other game engines. Workarounds are therefore implemented to represent the hyperbolic space. In this section, the methods used to represent a hyperbolic space in Unity and the use of portals to mimic hyperbolic room connections will be discussed.

4.1.1 Creating Unique Tiles

The game has an underlying data structure that is used to represent the connections between different rooms. This data structure is a core part of the game as it represents its hyperbolic nature through these connections. Section 3.2.3 already goes into the logic the program uses to keep track of which tiles are neighbouring each other and how to keep each tile location unique. All of the logic that is explained in that section is implemented in a class called OrderFourPentagonal. The OrderFourPentagonal can be used to modify and normalize instances of the TreePath class, which represents locations within the graph structure.

The separation of the TreePath class from the OrderFourPentagonal class was done to separate functionality. The TreePath class only represents the location but holds multiple fields that make the operations that need to be done on it simpler and more efficient. The operations that are executed on a TreePath are specific to the tiling that is used. This means that if the game would ever be expanded to use more types of tiling a new class would need to be created with operations specific to that new tiling. However the functionality of the TreePath would not have to be changed. So separating their functionality makes it easier to add new tilings in the future.

The TreePath class still has two important methods even though all of the graph logic is implemented in the OrderFourPentagonal class. The first method returns a string representation of the TreePath. The second method uses this identifier to assign a color to the TreePath.

The string representation generated by the first method serves as a unique identifier. It is used to interact with the map (see section 4.4) and to make debugging easier. The identifier is simply a sequence of letters that follow the steps taken to get to the represented location. O for origin, N, E, S and W[1] for the initial direction and l, f, r[2] for all the steps after the initial direction.

The color returned by the second method is mostly unique to a location. It is not possible to make these colors entirely unique, since there are infinite locations but not infinite possible colors. The method uses an MD5 hashing function. There are two reasons the program uses the MD5 hashing algorithm for this. The first reason is that the color of a tile must not change between calls of the method. Otherwise, the color would not actually be representative of a location and would thus not aid in navigating the space. Hashing algorithms satisfy this property since inputting the same value multiple times always returns the same output each time. The second reason for using the MD5 hashing algorithm is that we want widely different colors from almost identical id's. The MD5 hashing algorithm is known for its widely differing outputs with very similar inputs, so using it also solves this second problem.

4.1.2 Room Connections and Spawning

The game needs a way to render the connections between different rooms in a non-euclidean way, since it is only here that the non-euclidean nature of the game becomes apparent (as explained in section 3.1). But this requires a workaround, since certain rooms (e.g. the "North-Right" and "East-Left" rooms) would occupy the same position in the euclidean game engine, despite being distinct rooms in non-euclidean space.

The workaround used is the inclusion of portals [15]. Portals come in pairs, where each portal is a plane whose face shows the view from a camera positioned at the linked portal. This allows the grid to be divided into various segments that are placed far apart in the game world, while creating the illusion that the player is looking at a connected grid. Using strategic placement of portals, the player can then see both the "North-Right" and "East-Left" tiles from the origin.

A significant downside of such portals is that their rendering is computationally expensive. Therefore, to maximize the performance of the game, portals should only be used where absolutely needed. That is, if a tile can be seen from only one side (as seen from the player's current position), that tile should be placed directly into the grid, rather than utilizing portals. The process of placing tiles and portals (illustrated in Figure 4.1) thus starts with directly placing all tiles that are in the same row or column as the player's current tile. The adjacent rows and columns then alternate between portals and directly placed tiles. Each of these portals connects to a smaller grid, on which this process is repeated.

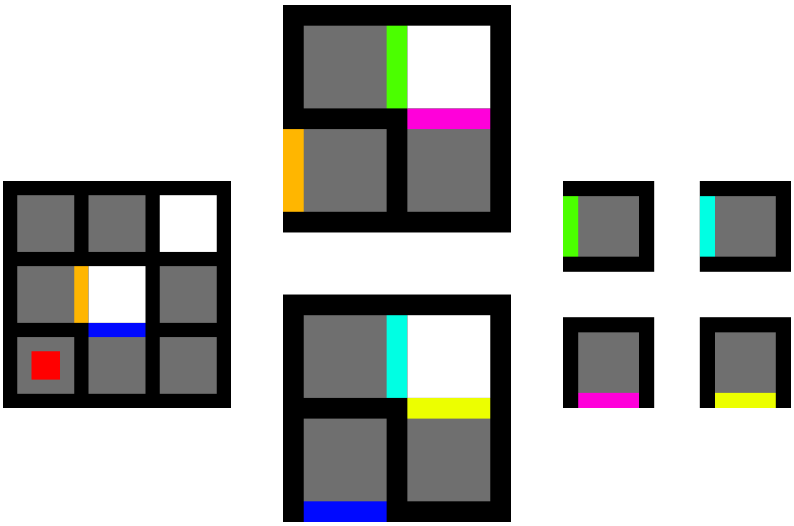
4.2 Navigating the Labyrinth

Game tasks keep players motivated while navigating the labyrinth, and provide more structured ways to analyze their performance. Different game tasks are configured by placing different special tiles in the virtual world. This section introduces the configuration of navigation and exploration tasks.

Game Starting Point

The starting point of the game is set to be the tile that the player is standing on when the game starts. This is calculated from the positional tracking data of the Head-Mounted Display (HMD). In order to designate a starting point, the operator can simply ask the player to stand in the desired tile when the game starts.

[1] N = North, E = East, S = South and W = West
[2] l = left , f = forward and r = right



(a) The main grid (b) The first layer of portal grids (c) The second layer of portal grids

Figure 4.1: Example of the placement of portals and physical tiles in the grid. The red dot marks the player's current location. Gray tiles mark physical tiles. Colored walls mark portals. Each portal is linked to the portal of the same color in the next layer.

Navigation Tasks

In a navigation task, the player needs to find their way from the given starting point to a set destination in the virtual world. This destination is marked by a special tile on which a flag is placed. Once the player reaches that location in the virtual world, they can interact with the flag to complete the game task.

Exploration Tasks

In an exploration task, a special tile with a locked chest is placed at the starting point. The player must explore the hyperbolic world and find a number of keys to open the chest. When the placements of keys are determined (see section 4.4), special tiles with keys will be spawned in those locations. Once the player has collected all of these keys, they must return to the starting point to open the chest. When they do, the "closed chest" tile will be replaced by an "open chest" tile. This way the chest stays open if the player moves away from and back to the starting tile.

4.3 Virtual Object Interaction

Interaction with virtual game objects makes the virtual world more entertaining and enables the configuration of various game tasks that players can complete. In order to offer better player experience without losing compatibility, three approaches of interaction were devised: proximity activation, proximity of controller, and controller ray interactor.

First, proximity activation takes the position of the player and activates a game object when they come close. Position of the player is measured and updated by the positional tracking data from HMD. This approach provides the best compatibility, without need for controllers. A capsule collider, 0.5m in diameter and 2m in height, is placed at the player position for proximity detection with game objects.

Second, proximity of controller is one step further from the previous approach. The player can try to reach virtual objects with their controller, and they can receive haptic feedback as confirmation of a successful interaction. Virtual hand models are rendered at the position of

tracked controllers to provide more immersion. A box collider of 0.15m on each side is attached to the virtual hand for collision detection. Upon reaching a game object, players would receive haptic feedback.

Finally, the ray interactor approach utilizes the capability of controllers to the fullest. The player can point a virtual ray with their controller. When the ray lands on a game object, its color would change as an indication. The player can then pull the trigger on the controller and activate the game object. A haptic impulse would also be sent as confirmation. With this approach, the player has a more entertaining way to interact with the game objects.

4.4 Creating the Labyrinth

This section will go into the placement of objectives. It will explain how the game keeps track of those placements across different sessions, and how the game automatically generates those configurations with input parameters.

4.4.1 Using Maps

A game level is represented by a MapDictionary data structure, which determines the placement of the objectives at runtime. The MapDictionary is a collection of key-value pairs, where the key represents the unique id of a tile location and the value represents the objective to be placed at that location. When the game needs to place a new tile at a location, it would first check the MapDictionary if it contains a key that corresponds to the location. If the key is present in the MapDictionary, the corresponding special tile will be placed. Otherwise, the game places a default tile.

It would also be convenient if game levels can be reused. Researchers, for example, may want to use the same level among different participants. After a level is generated, the MapDictionary is converted to a JSON file and saved in a dedicated folder for save files. If the same level needs to be used, loading the game with the save file will recreate the MapDictionary and give the same level.

A level generated from a seed can also be reused, but the seed is not saved locally. The operator needs to remember the seed they have used. When starting a game with the same seed, the same level is generated.

4.4.2 Creating Maps

Creating a map can be done in three ways. The first is to do it manually. The save files that store different maps are in a JSON format. This format is not hard to read and write by hand. It is thus possible to create a plain text file and write a map by hand by following the structure the save files use.[3]

The second way is to generate a MapDictionary by giving certain input parameters. These input parameters include the amount, distance and type of cubes the operator wants to place in the map. The methods that are responsible for creating the MapDictionary change depending on the distance the objective is to be placed away from the origin as there is a large overhead to make sure collisions (trying to place an objective in the same spot) do not occur. And after a certain point (that is decided by a field in the game) the chance of collisions is so small that the overhead is not worth it.

The third and final way does not require a MapDictionary. Instead, it lazily decides if an objective needs to be placed at a location. This is achieved by concatenating the id of the location with the seed and putting that into a hashing function. The outcome of the hash function is then transferred to a function that returns a certain objective with a probability equal to a user given frequency.

[3] A better guide on how to do this along with many other factors of operating the game is available in the game documentation

4.5 Menu and Visuals

The game has a few displays for varying purposes. Some are to make the operation of the game easier for the user. Others are for displaying information to the user while they are trying to navigate hyperbolic space. This section is about those displays and the elements they show.

4.5.1 Menu Options

The main menu was designed with simplicity and user-friendliness in mind. The menu uses contrasting colors to be easily distinguishable in VR. Two-way navigation is possible between menu pages, meaning the user can move back and forth if they want to.

The main menu has four options: "Play", "Tutorial", "Settings", and "Quit". Among these options, "Play" and "Settings" have their own sub-menus. Users can change the game settings in the "Settings" sub-menu, by using toggle buttons. The "Play" sub-menu allows the user to create a new level based on provided inputs, or load a previously saved level. The "Tutorial" option allows the user to initiate a level to learn the basics, and "Quit" allows exiting the application.

4.5.2 In-Game User Interface

In the exploration task, the player needs to find a certain number of keys to open the chest. The number of collected keys and the number of required keys are displayed in the top left corner of the screen to give a clear view of the player's progress.

If the operator sets a time limit on a game task, a countdown timer will be displayed in the top right corner of the screen. When this countdown reaches 0, the game will show a screen to let the player know the game has ended. The player will still be able to walk around in case they did not want the game to end yet.

4.6 Minimap Rendering

It is easy to get lost in the massive and unfamiliar hyperbolic space. A minimap that indicates the player's location and orientation can help them plan longer routes through the labyrinth. Therefore, the operator of the game can allow players to use a minimap to help with navigation. In this section, the appearance and behavior of the minimap, as well as the details of how it is generated, are discussed.

4.6.1 Minimap Layout

When the minimap is enabled, it can be displayed at the top of the screen. The minimap is circular, and it renders the Poincaré Disk model of the hyperbolic plane, with the rooms laid out according to the squares of an order-5 square tiling [9]. When the player turns, the minimap rotates along with them. An illustration of the minimap, rendered with the player at the origin and without rotation, is displayed in Figure 4.2.

The minimap is always centered on a tile, and not on the relative position of the player within that tile. This is to avoid having to recompute a new minimap every frame, which would significantly impact the overall performance of the game.

Objectives are indicated on the map to make it easier for players to find them. When an objective tile is on the minimap, it is indicated with stripes in opposite color.

4.6.2 Minimap Generation

Since hyperbolic room connections are simulated by dynamically spawning the required tiles, it is not possible to simply use an orthographic camera view from above the player. It is also not feasible to use a fully precomputed minimap, since the number of rooms grows exponentially with the distance from the origin. Because of these constraints, the minimap has to be

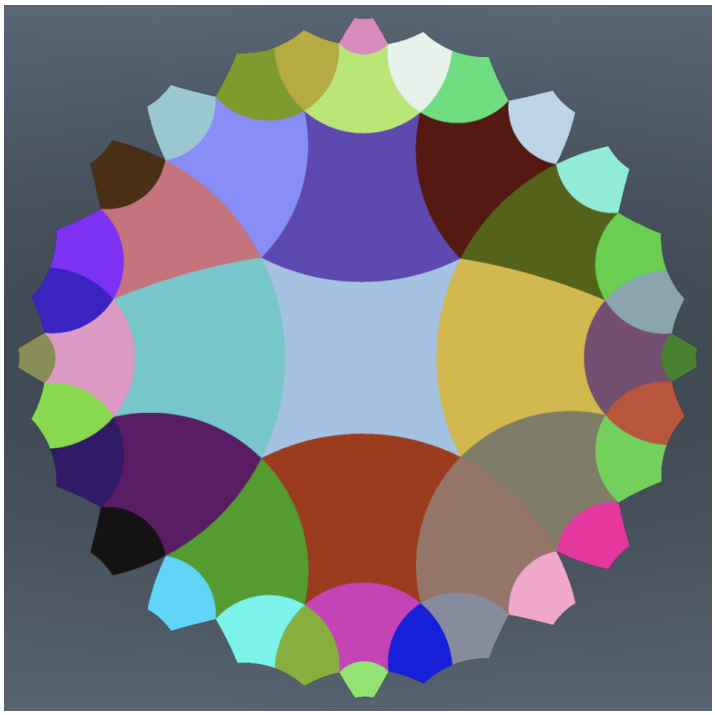


Figure 4.2: An image of the minimap, with the colors of the squares matching the colors of the tiles in the game.

generated dynamically. This is achieved as follows: when a player enters a room, the TreePath of the tile is used to find the index of a specific edge of that tile[4]. The position of this edge is then found by appplying a series of matrix transformations. The approach taken here is derived from [16], adapted to draw tiles instead of edges.

The matrices in question are listed below. The first is a rotation matrix, which rotates the hyperbolic plane by an angle θ. The second and third are translation matrices, which move a point by a distance x or y.

$$\begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & cosh(x) & sinh(x) \\ 0 & sinh(x) & cosh(x) \end{bmatrix} \begin{bmatrix} cosh(y) & 0 & sinh(y) \\ 0 & 1 & 0 \\ sinh(y) & 0 & cosh(y) \end{bmatrix}$$

In order to make it easier to manipulate the hyperbolic coordinates, they are stored as Minkowski Hyperboloid coordinates, not Poincaré Disk coordinates (see section 3.2.2). These matrices are then applied to a column vector containing the x, y and t coordinates of the hyperboloid. When the proper sequence of transformations has been applied, the positions of the other three edges are computed from the first.

After that, a texture is generated for each tile. First, points on a disk are projected onto the hyperboloid, with one point for each pixel. Next, the four edges of the tile are used to check if a given point is inside or outside of the tile. If it is inside, the color of the corresponding pixel is set to the color of the tile. This process of finding edges and generating textures is repeated for every tile reachable in n steps or less[5], and the resulting textures are merged into one. This merged texture is then rendered onto the HUD.

If the minimap must be rendered with the center at a tile other than the origin, the TileShift class is used. This class is used to compute which tile is shifted to a given location when the map

[4]Usually, this edge is the right-hand wall from the perspective of the player, but since a TreePath has four branches, and an EdgeTree has five, there are certain tiles for which this does not hold, and those must be handled separately.

[5]In practice, more than 3 steps is too computationally expensive if a new minimap has to be generated every time a player moves between tiles.

has been centered to a new tile. This new tile can then be used to determine the color of the equivalent old tile in the minimap. This way, the number of matrix transformations required to render the minimap is constant for a given depth, which prevents both performance degradation and issues with numerical precision that commonly occur when using floating-point arithmetic.

Chapter 5

Discussion and Recommendations

This chapter explains how this project can be improved and used in the future. Section 5.1 goes over potential additions and improvements to the game. Section 5.2 explains how the game can be used for various experiments.

5.1 Possible Alternatives and Improvements

This section will go over a few additions and improvements that could be made in the future. These have not yet been implemented either because they were not in the scope of this project, or because there was not enough time. They include: optimization, the usage of different tilings, customization of the grid, the addition of new game modes, and improvements to the in-game menus.

Optimization

Due to time constraints, the performance of the game has not yet been optimized. The minimap and the portals are the two features that still require much optimization. The minimap could be optimized by implementing a different algorithm to render it. The amount of portals could possibly be reduced even further to compensate for the cost of rendering them.

Different hyperbolic tilings

An interesting but non-trivial expansion would be to generalize the Order-5 square tiling that is currently used to other hyperbolic tilings. This addition would facilitate new experiments regarding the navigation of hyperbolic space. It would be non-trivial because many algorithms used in the game rely on the tiles being squares in a specific layout. The graph of the room connections (3.2.3) and the minimap (4.6) are examples of such algorithms.

A customizable grid

The game is currently always played on a 3x3 grid. In the future, the width and height could be determined by the operator or the player. Most of the algorithms that are used for the placement of rooms on the grid have already been made with a variable size matrix in mind (4.1.2). Adding this feature only requires integration with the menu and a simple algorithm that to generate a grid of variable width and height. There could also be a setting to place additional walls between grid tiles, to force the player to think of other routes.

New game modes and tiles

The game could be expanded with new game modes and special tiles. Such additions may facilitate a wider variety of research. An example of a possible game mode can be one where certain tiles are only accessible to players when they have reached some objective first. This mode would open up new possibilities as players explore the virtual world. An example of a new special tile is a teleportation tile, which would add another element to keep in mind during navigation.

Menus

One of the key requirements for the project was to have an in-game menu so the player can change settings on the fly. The game has no such thing at the moment. The player can at most toggle the minimap from visible to invisible. The main menu is also still a bit lacking as a lot of possible settings are not currently implemented. For example, the main menu cannot be used to control how many keys are needed to open a chest, or which statistics the player is able to see in their HUD.

5.2 Further Research

This project is intended specifically for behavioural experiments, but it may facilitate other types of research as well. Two of these research possibilities are introduced in this section.

For example, it might be interesting to research a practical way to utilize the hyperbolic geometry as a new way of travelling in VR. In the current state of the game, it is possible to travel arbitrarily far from the the starting point while being in a confined physical space. It would be interesting to generalize this feature to a larger variety of VR games and applications that require a lot of walking space, as they can then be used even with a physical setup of limited space.

It would also be interesting to study navigation behaviours in hyperbolic space when there is no visual cue available to players. Currently, the players can see a few tiles away through the portals. Instead, what if they cannot see any surrounding tiles at all? Will they still be able to navigate the hyperbolic space in this setup? These questions can point to the potential limitations of navigation abilities in hyperbolic space.

Chapter 6

Conclusion

The goal of this report was to describe the design and development of an interactive VR game that lets a player experience non-euclidean geometry. This game was meant to test the navigational skill of people in a non-euclidean environment. To do so, the form of the game first needed to be decided. This was done by doing research on non-euclidean geometry and existing projects that render or simulate such geometry in VR.

The core requirements of the game are as follows. The game should be based on a 3x3 grid of cubes that are connected in a hyperbolic way. The grid should be surrounded by walls that are synchronised with the boundaries that were set up in the real world. Players should be able to traverse the virtual space by moving around in the real world. Players should have access to a Poincaré disk projection of their surrounding to use as a minimap. Finally the game should have a menu so that users are able to configure the parameters of navigation and exploration levels, among other settings.

The eventual form of the game uses a 3x3 grid in the physical world. Each square in the grid corresponds to a square in the virtual world and the alignment of boundaries is also implemented. The squares in the virtual world are connected based on a discrete representation of an order-5 square tiling. This means that it would take 5 right turns to end up back in the same spot in the virtual world. Because this would cause two different squares to take the same spot in the 3x3 grid, portals are used to display different squares from different angles. To help players distinguish between different squares, each square is assigned a unique color. There are also objectives the player needs to complete, such as finding their way to a flag that is placed on the map or collect some key to open a chest. The game can generate different worlds with these objective by placing them in different locations. Furthermore, player has information displayed on the screen to help them with their tasks. For example, a minimap of their surroundings and the number of keys they have already collected. Finally the game has a main menu that allows operators to set up levels and alter setting options, while an in game menu is absent. The game satisfies the core requirements laid out by the client in its current form.

The game is in its prototype stage, and there are still a lot of possibilities of expansion. For example, it is possible to make the size of the grid more dynamic or to generalize the order-5 square tiling the game is currently using to an order-n square tiling. The game could also definitely still be improved in an optimization perspective as the portal and minimap features require a lot of computing resources. However, the current version of the game does satisfy the most important requirements, and is usable for testing navigational skills in a non-euclidean environment.

Bibliography

[1] F. Klein, "Ueber die sogenannte nicht-euklidische geometrie," ger, *Mathematische Annalen*, vol. 6, pp. 112–145, 1873. [Online]. Available: http://eudml.org/doc/156604.

[2] M. Phillips and C. Gunn, "Visualizing hyperbolic space: Unusual uses of 4x4 matrices," in *Proceedings of the 1992 symposium on Interactive 3D graphics*, 1992, pp. 209–214.

[3] J. Weeks, "Real-time rendering in curved spaces," *IEEE Computer Graphics and Applications*, vol. 22, no. 6, pp. 90–99, 2002.

[4] T. Novello, V. da Silva, and L. Velho, "Global illumination of non-euclidean spaces," *Computers & Graphics*, vol. 93, pp. 61–70, 2020.

[5] E. Kopczyński, D. Celińska, and M. Čtrnáct, "Hyperrogue: Playing with hyperbolic geometry," in *Proceedings of Bridges 2017: Mathematics, Art, Music, Architecture, Education, Culture*, 2017, pp. 9–16. [Online]. Available: http://archive.bridgesmathart.org/2017/bridges2017-9.pdf.

[6] V. Hart, A. Hawksley, E. Matsumoto, and H. Segerman, "Non-euclidean virtual reality I: Explorations of $\mathbb{H}^3$," in *Proceedings of Bridges 2017: Mathematics, Art, Music, Architecture, Education, Culture*, 2017, pp. 33–40. [Online]. Available: http://archive.bridgesmathart.org/2017/bridges2017-33.pdf.

[7] ——, "Non-euclidean virtual reality II: Explorations of $\mathbb{H}^2 \times \mathbb{E}$," in *Proceedings of Bridges 2017: Mathematics, Art, Music, Architecture, Education, Culture*, 2017, pp. 41–48. [Online]. Available: http://archive.bridgesmathart.org/2017/bridges2017-41.pdf.

[8] J. Weeks, "Body coherence in curved-space virtual reality games," *Computers & Graphics*, vol. 97, pp. 28–41, 2021, ISSN: 0097-8493. DOI: https://doi.org/10.1016/j.cag.2021.04.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0097849321000443.

[9] H. S. M. Coxeter, *The beauty of geometry: Twelve essays*. Dover Publications, 1999.

[10] J. G. Ratcliffe, S. Axler, and K. Ribet, *Foundations of hyperbolic manifolds*. Springer, 1994, vol. 149.

[11] M. Lavrov, *Description of the order-5 square tiling of the hyperbolic plane as a graph*, Mathematics Stack Exchange, (version: 2020-06-12). [Online]. Available: https://math.stackexchange.com/q/2231612.

[12] I. R. Van de Poel and L. M. Royakkers, *Ethics, technology, and engineering: An introduction*. Wiley-Blackwell, 2011.

[13] U. A. Chattha, U. I. Janjua, F. Anwar, T. M. Madni, M. F. Cheema, and S. I. Janjua, "Motion sickness in virtual reality: An empirical evaluation," *IEEE Access*, vol. 8, pp. 130 486–130 499, 2020. DOI: 10.1109/ACCESS.2020.3007076.

[14] R. S. Fisher, J. N. Acharya, F. M. Baumer, *et al.*, "Visually sensitive seizures: An updated review by the epilepsy foundation," *Epilepsia*, vol. 63, no. 4, pp. 739–768, 2022. DOI: https://doi.org/10.1111/epi.17175. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/epi.17175. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/epi.17175.

[15] L. D. Wen, *Multiple recursive portals and ai in unity*, [Accessed Jun. 7 2022], 2020. [Online]. Available: https://github.com/limdingwen/Portal-Tutorial-Repository.

[16] M. Lu, *Hyperbolic geometry for the uninitiated and curious*, [Accessed Jun. 19, 2022]. [Online]. Available: https://martinlusblog.wordpress.com/2020/07/25/on-simulating-and-understanding-hyperbolic-space-with-no-previous-knowledge-on-the-subject.

[17] Zeno Rogue Games, *Hyperrogue*, [Accessed Jun. 7, 2022], 2015. [Online]. Available: https://roguetemple.com/z/hyper/.

[18] CodeParade, *Hyperbolica*, [Accessed Jun. 7, 2022], 2022. [Online]. Available: https://store.steampowered.com/app/1256230/Hyperbolica/.

[19] HackerPoet, *HyperEngine*, [Accessed Jun. 7, 2022], 2022. [Online]. Available: https://github.com/HackerPoet/HyperEngine/.

[20] V. Hart, A. Hawksley, and H. Segerman, *HypVR*, [Accessed Jun. 7, 2022], 2017. [Online]. Available: https://github.com/hawksley/hypVR/.

[21] Tekton Games, *WalkAbout*, [Accessed Jun. 7, 2022], 2016. [Online]. Available: http://tektongames.com/walkabout-presskit/.

[22] HackerPoet, *NonEuclidean*, [Accessed Jun. 7, 2022], 2018. [Online]. Available: https://github.com/HackerPoet/NonEuclidean/.

[23] A. A. Ungar, "A gyrovector space approach to hyperbolic geometry," *Synthesis Lectures on Mathematics and Statistics*, vol. 1, no. 1, pp. 1–194, 2008.

[24] GameCI, *GameCI*, [Accessed Jun. 7, 2022]. [Online]. Available: https://game.ci.

Appendix A

Requirements

A.1 Must Have

- The virtual space must be based on a physical grid of 3 by 3 squares.
- The game must be able to connect rooms according to the square tiling of the euclidean plane or the order-5 square tiling of the hyperbolic plane.
- The game must be able to map physically present walls into virtual space and align accordingly.
- The game must start at a marked starting square in the physical grid.
- Players must be able to navigate between rooms in the virtual space by moving around the physical space.
- The player must have a map in their HUD that represents the location they are in, in the virtual space using the Poincaré disk model.
- A room must have one transparent passage to each adjacent room corresponding to a physical square.
- The game must have a main menu for access to the game, and an in-game menu for customization during gameplay.

A.2 Should Have

- The game should have a mode that adds a time limit.
- The operator should be able to select and customize the game mode the player will play (e.g with or without a map).
- The operator should be able to create and save levels.
- The operator should be to automatically generate a level using given input parameters (e.g 3 chest within a map with the radius of 10 squares)
- The game should be able to load previously saved levels.
- The game should have an option to add opaque doors between the rooms.
- The game should have a tutorial mode.
- The player should be able to pick up and interact with placed objects by proximity.

A.3 Could Have

- The game could have alternative modes, such as but not limited to:
  - a survival horror mode.
  - a mode that gradually makes tiles inaccessible to travel depending on the time the user spends to complete their task.
  - a mode where additional walls are added between certain rooms (besides the ones present in the physical space).
  - a mode where the user needs to complete a certain task before being able to go to the next room.
- The game could collect and display game metrics (e.g. time played, tiles visited).
- The game could be able to deterministically generate levels from a seed.
- The game could be adapted to a physical grid of n by m squares.
- The game could have a mode that makes a spherical space appear bigger.
- The game could allow for interacting with objects through pointing a controller and pressing a button.

A.4 Won't Have

- The game won't have true non-euclidean rendering and physics.
- The game won't have a multiplayer mode.
- The game won't have advanced graphics (which may hinder performance).
- The game won't have variable tiling (different types of tilling in one level).

A.5 Non-Functional Requirements

- We must use Unity to develop the application.
- We must use Git and Gitlab.
- The game must be runnable on an HTC Vive and/or Oculus Rift.
- The game should run on a normal computer.
- The deliverable should be able to be run on both Windows and MacOS.
- The game could be exportable to WebVR (under certain conditions/ if possible).

Appendix B

Division of Labour

	Baran Ozan Yarar	Bo Bakker	Ravi Snellenberg	Riley Slotboom	Wenkai Li
VR Setup				x	x
Graph Datastructure		x	x		
Euclidean Demo	x				
Git Integration					x
Pipeline					x
Menu	x				
Portals				x	
Walls	x				
Level Creation			x		
Level Generation			x		
Time Limit Mode					x
HUD and Metrics					x
Minimap		x			x
Object Interaction					x
Non-Euclidean Demo	x	x	x	x	x
Report	x	x	x	x	x

Appendix C

Feasibility Study

C.1 Existing products

The products that were examined and researched by the team prior to the design process are:

C.1.1 HyperRogue

Similarities. HyperRogue [5], [17] is a top-down game set on the hyperbolic plane. This top-down perspective is similar to what our game's map should look like.

Shortfalls. Movement in HyperRogue works vastly differently from what we need, as it is top-down rather than first-person virtual reality. As such there is also no connection between physical and virtual space. Lastly, HyperRogue uses an irregular tiling of hexagons and heptagons rather than a regular tiling of squares.

C.1.2 Hyperbolica

Similarities. Hyperbolica [18] is a first-person/VR game set on the hyperbolic plane. Like our game, it focuses on exploration of this hyperbolic setting. It has also created a way to get Unity to render the game in an actual non-euclidean way. The developer has released most of the source code for this Unity backend [19].

Shortfalls. While Hyperbolica can be played in VR, it does not have any physical setup like the grid and the walls we will be using. This means it does not have the connection between physical and virtual space that our game must have.

C.1.3 hypVR

Similarities. hypVR [20] uses a VR headset, both for computing user's location and movement. It also uses $\mathbb{H}^2 \times \mathbb{R}$, meaning the virtual space has a euclidean vertical axis and hyperbolic horizontal axes, which solves certain issues of using a VR headset in non-euclidean spaces.

Shortfalls. hypVR creates no barriers between rooms, while in our case, the client requested virtual walls that correspond to the physical walls in the room. It also does not have alternative game modes other than simply exploring the place.

C.1.4 WalkAbout

Similarities. WalkAbout [21] is a system for VR locomotion designed to let the user move around in physical space to move their character in virtual space. This is also what we are trying to achieve.

Shortfalls. WalkAbout achieves this locomotion by letting the user "freeze" the virtual movement. That is, the user can press a button that allows them to turn around in physical space without moving in virtual space. This means that it does not achieve our goal of aligning virtual walls with physical walls.

C.2 Time Constraint

The client pictures the product as a prototype that can be used for psychological experiments, which allows for simplification of some development strategies, such as how should we render the virtual space. Our client has very considerate expectations on what is considered as the "bare minimum" of the product. Considering the scope of the project, and our motivated and experienced team members, we are confident that we can complete the project within the time frame.

C.3 Game Engines

Game engines provide developers with a powerful framework and a set of useful tools, so that game developers can focus less on low level implementation details such as physics and rendering, and more on the actual game design. With the time span and the focus of the project, it is only reasonable that we base our development on an existing game engine rather than building one from scratch by ourselves.

C.3.1 Non-Euclidean Game Engines

However, since the real world (usually) follows regular euclidean geometry, so do most of the game engines. Over the years people have developed tools to help with the development of non-euclidean games. One game engine[22] that we discovered was developed by HackerPoet, or CodeParade. It is written for Windows, which may create some compatibility issues with MacOS (which some of our team members will use for development). This compact engine provides support for non-euclidean physics, collision detection and rendering, but it has no support for virtual reality, which will likely cause some problems if we were to use it. Therefore, this engine is not the most suitable choice given our circumstances.

Another project that could be a viable option is HyperEngine [19]. This project was also developed by HackerPoet (also known as CodeParade) and is an extension to the Unity game engine. It adapts the unity rendering and physics engine to non-euclidean space, and since it builds upon the unity engine, it should still be compatible with virtual reality development, even though it was made with a focus on regular gameplay. Integrating it with VR development and making it run smoothly could still be very challenging and time-consuming, but if we need to fully adapt the unity engine to non-euclidean geometry, using this would save us a lot of time and trouble.

C.3.2 Euclidean Game Engines

Alternatively, it is also possible to simulate non-euclidean geometry with regular euclidean game engines, by manipulating the rendering pipeline and with programming workarounds. This approach has sound mathematical basis [6], [7] and is proven feasible by multiple existing software like Hyperbolica [19] and hypVR [20]. This gives us the liberty to choose from a wide variety of game engines the one that is most suitable for our use case.

Unity is the engine proposed by the client and the one we spent most time researching. Unity is one of the most popular cross platform engines that support both Windows and MacOS. It uses C# as programming language, which our team members should be able to pick up without extensive learning. Unity has great support for integration with a wide variety of VR headsets, and a Unity project can be easily exported to WebVR, which is useful for testing.

Therefore, Unity ticks every box on our client's list of requirements and is indeed a feasible choice for the project.

Simulating non-euclidean geometry in a euclidean game engine results in a number of problems, to which special attention ought to be paid. One such problem arises because Unity does not render objects that are not in the field of view. Normally, this is an effective way to save resources. However, the methods used for this are based on euclidean geometry and will not work for our use case. If we were to use them with non-euclidean geometry, the engine may omit objects that are actually in the field of view, or waste resources on objects that are not.

Another problem of simulating non-euclidean geometry is related to the matrix transformations. Unity uses matrix multiplication for various transformations, such as translation and rotation. Again, this is based on euclidean geometry. To adapt this to non-euclidean geometry, we would need to change such matrix multiplications to use gyrovectors [23].

And not just the rendering of the scenes causes problems the physics engine of unity also does not work well with non-euclidean geometry. This happens because objects get stretched/squished depending on the distance in non-euclidean geometry while the physics engine of Unity does not work well with that at all.

C.4 Development Pipeline

Game development is somewhat different from the software development we are used to, and it often utilizes specialized software to manage the development process and pipeline. For example, the version control systems that Unity has integrated in it are Perforce and Plastic SCM, and not Git, which we are most familiar with.

There are a few motivations behind this different approach. The large number of assets involved in game development can contain large binary files, which Git was not initially designed to handle, and they can lead to merge conflicts that are very difficult to solve. Besides, Unity keeps its reference to objects using Globally Unique Identifiers. If those GUIDs are not handled properly with Git, the same code can break for other team members.

Fortunately, Git developers have kindly catered to such needs over the years, with features like Git Large File Storage (LFS) and file locking. LFS can be utilized to track and store large asset or binary files, while file locking can prevent concurrent edits to certain critical asset from different team members.

The game developers community has also come up with a more accessible solution to continuous integration for game development, which is GameCI [24]. It utilizes Docker image to support continuous integration and smooths the game development pipeline for developers.

C.5 Resources and Support

Certain resources are indispensable to the development of this project, such as virtual reality headsets and a room where we have enough space to navigate. The client has kindly made arrangements for such resources which we can start working with from as early as the third week. Additionally we may get a cardboard virtual reality headset (for use with a smartphone) for simpler testings at home.

C.6 Risk Analysis

Our group is attempting to create an unconventional product (a non-euclidean VR game) while having limited experience in developing large applications, especially games. This implies that there are a number of things that could go wrong during the development of our application. Below, we will mention possible sources of risk that may cause issues and hinder development in the course of creating our product.

C.6.1 Inexperience with Unity

We aim to create our product with the Unity game engine. Since the majority of our group has little to no experience with Unity, we can expect that there will be a learning curve at the start. This may not just lead to a slow start in the development of the product, but also to errors and bugs due to lack of insight in the way Unity compiles and runs its files. However, considering we have a member experienced with Unity and also that Unity is regarded as one of the more beginner friendly engines, the learning process may be relatively easy.

C.6.2 Dependence on physical equipment

A big part of our product is the integration of Virtual Reality. This brings with it that testing our product can not be fully done using only a computer. To properly test our product we need the equipment to run and play Virtual Reality games. The equipment for this is luckily provided by our client, but if this equipment were to malfunction, we would not be able to test our product anymore. We are also limited to testing large parts of our code on site where the equipment is available, which results in a potentially slower testing feedback cycle.

C.6.3 Unforeseen requirements/problems

While there are some precedents to the non-euclidean game we want to develop, it is still not a common and proven practice, and there are quite a few experimental parts to our product. Adding the fact that none of our group members have any experience in tasks like what we want to accomplish, there will be a decent chance that we will encounter many problems during the development that we have not yet foreseen at this stage of planning. Each of these problems may increase the risk that the product does not reach the intended quality level before the end of the deadline.

C.6.4 Testability

Testing games, and in particular virtual reality games, can be very different from how we are used to testing normal software. Sometimes the code may not be suitable for automated testing and manual testing and visual confirmation may be required. Our team members are not very experienced with the common and best practice of game testing, or with the Unity testing framework. This can lead our code to end up insufficiently tested, or may mean that we will need to spend a significant amount of time on inefficient manual testing, leaving less time for other parts of the development.