Mapping hyperbolic space for the virtual reality game ”Holonomy”

A. de Vries [1]

Supervisor(s): M. Skrodzki [1] **R. Bidarra**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

CSE3000 – Research Project – Final Report (2023)
Delft University of Technology
M. Skrodzki (Responsible professor), R. Bidarra(Supervisor) and G. Smaragdakis(Examiner) (Editors)

Short Paper

Mapping hyperbolic space for the virtual reality game "Holonomy"

A. de Vries [1]

[1]TU Delft, The Netherlands

Abstract

Non-Euclidean spaces are spaces that do not satisfy all of Euclid's postulates. An example of such a space is hyperbolic space. In this paper, a method is discussed to draw a tessellation of hyperbolic space in a manner that fits with the virtual reality game "Holonomy", a game which takes place in hyperbolic space. The main result shows that the new approach is better in terms of simplicity than the old approach and is able to draw the game world more faithfully than the old approach. The features of the new approach could still be significantly expanded upon.

1. Introduction

Recent developments have made it possible to immerse oneself into non-Euclidean spaces, i.e. spaces that do not satisfy all of Euclid's postulates. A common example of a non-Euclidean geometry is hyperbolic geometry. Hyperbolic geometry was not fully mathematically described until only the 19th century, when mathematicians such as Gauss, Schweikart, Lobachevsky and Bolyai among others published studies on the topic of non-Euclidean geometries [Cox98]. In short, Hyperbolic geometry is the geometry where the fifth postulate of Euclid's postulates is negated [Ben01]. This results in a geometry where objects live on a surface that curves away from the origin, as opposed to a flat Euclidean surface. Whenever hyperbolic geometry is introduced, it is normal to think of it as a very abstract concept, how would one even imagine what it is like to live in a curved space? This makes it interesting to visualize the effects of hyperbolic geometry with the use of virtual reality, since that makes understanding its unintuitive properties more intuitive to understand. For example, one such property that is interesting to illustrate is the effect of Holonomy, which is the idea that the world rotates around an observer when they move through it, even though the observer keeps facing in the same direction [Wee21].

The main problem in this research is improving the map of an already developed virtual reality game that takes place on the hyperbolic plane. The goal in this research is to make it simpler and faster than the current implementation. In the game, which is called "Holonomy", users navigate in 3x3 meter grid [YBS*22]. The floor in the game world is a hyperbolic plane, while the vertical direction is normal Euclidean space. Formally, the dimension in which the game takes place is thus $\mathbb{H}^2 \times \mathbb{E}$. Here, $\mathbb{H}^n$ means hyperbolic space in n dimensions and $\mathbb{E}^n$ means Euclidean space in n dimensions. While navigating this space, the goal is to collect a number of keys, and then return to the starting position to open a treasure. Another gamemode is to reach a set of flags in a level.

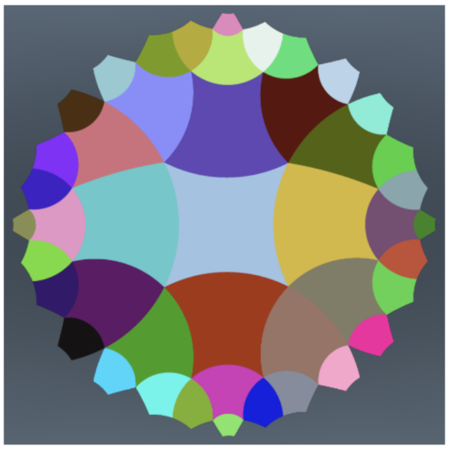
Figure 1: *Old rendering of the minimap [YBS*22]*

The game world is made up of tiles of the hyperbolic plane, such that each room in the game represents a tile of the hyperbolic plane. Players can see their current position in the game world via a minimap. Its old implementation can be seen in Figure 1. Each tile is given a unique color such that it is possible for the player to distinguish which tile they are currently at.

The old implementation of the minimap is not well documented and was assessed difficult to grasp. This is why we create a new minimap generation algorithm that can be well understood such that it is easy to extend with new features. One such feature would be making the minimap continuously rendering within the game. The old minimap is only rendered once every new tile. This means that the minimap is always centered on the room in which the player is currently present. The minimap is only updated whenever the player crosses over to a new room, at which point the minimap

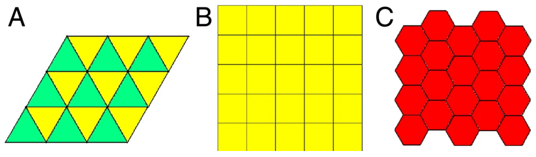

Figure 2: *Regular tessellations of the Euclidean plane [CJT11]*

is centered on that new room, because the old implementation is too slow for continuous rendering. To make the effects of hyperbolic geometry more intuitive, the map needs to be updated continuously as the player moves itself in the hyperbolic plane. This helps the player to orient themselves. Exploration of the game is not pleasant if the player is seriously disoriented [DP01].
The reason why it is challenging to draw the hyperbolic plane in a fast manner is because the number of tiles to render grows exponentially with relation to the distance from the origin tile. While all tiles in the hyperbolic world are perfectly square, we as humans do not live in a hyperbolic world but in a Euclidean world. Thus, we can only look at the hyperbolic world from our own Euclidean world, which means we need a model for the hyperbolic world. One such model is the Poincaré disk model, which is used in the rendering of Figure 1. This model distorts the size of each tile. As a result, it becomes difficult to draw a lot of these different tiles, since almost all tiles have a different Euclidean size in this model in relation to each other. Meanwhile, the size of all tiles in the Euclidean plane is the same no matter what way we project it, since us humans live in a Euclidean world. The Poincaré disk model is discussed more in Section 2.
The main aim in this research is thus to derive a simple and fast algorithm that renders the minimap in the VR game "Holonomy". In this light, the following main research question has been formulated:

What is a simple and fast algorithm to render the hyperbolic plane and is suitable for the VR game "Holonomy"?
Subquestions that derive from this main research question are:

- Is it possible to derive a simple and fast algorithm by first drawing an origin polygon, then reflecting that polygon to draw a grid?
- How much of a change in simplicity and speed would the new minimap provide compared to the old minimap?

The derived algorithm in this paper can draw the tile representation of "Holonomy" more faithfully than the old algorithm. It is also less complex than the previous solution, which makes it possible to extend the implementation more easily with extra features.
Section 2 provides an overview of previous work done in this area. Section 3 discusses implementation of the developed algorithm and how it works in practice. Section 4 compares the new solution to the old one in terms of code complexity and speed. It also discusses limitations of the implementation. Finally, section 5 draws the main conclusion.

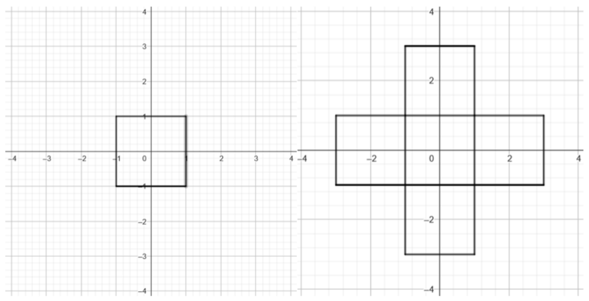

Figure 3: *Generating a Euclidean tiling by reflecting an origin tile*

2. Background information & related work

This section discusses other work that has taken place in generating tilings of hyperbolic space. These different methods all have their own advantages and disadvantages.

2.1. Theoretical background

In this paper, we are only interested in computing a *regular* tiling. A tiling is regular if it is made up of regular polygons. [GS77]. Regular polygons are polygons of which all of its angles are the same size, and whose sides are of equal length. So fundamentally speaking, a regular tiling of any plane is just a series of vertices and lines that connect those points, which make up the regular polygons. Thus, to draw a regular tiling of any plane, either of two general methods can be applied:

- First generate all vertices of the tiling, then connect those vertices via geodesics (a shortest path between points)
- Draw an origin tile whose vertices and geodesics are known, then reflect that origin tile along each of its edges to recursively draw a tiling.

As an example, consider we'd want to tile the Euclidean plane using squares (as in Figure 2). In the second method, one would draw 4 points a distance d from the origin of the grid, such that all angles between neighboring points are $\frac{\pi}{2}$ and the angles within each polygon are $\frac{\pi}{2}$. By "neighboring points", we mean points that are generated closest in Euclidean distance to one another for the remainder of this paper. Then, we would connect those points via straight lines. Now we have an origin tile. Since we now have an origin tile, we can reflect the tile along all of its 4 sides to end up with 4 new tiles. We can continue reflecting those new tiles along their sides for as long as we want to recursively draw a tiling of the Euclidean plane. This idea is illustrated in Figure 3.
There are only 3 ways to regularly tile the Euclidean plane: triangles, squares and hexagons as can be seen in Figure 2. There are in fact an infinite amount of ways to tile the hyperbolic plane, it can be tiled using any p sides per tile and q amount of sides that meet at a vertex so long as $\frac{1}{p}+\frac{1}{q}<\frac{1}{2}$. From now on, the Schläfi symbol $\{p,q\}$ is used to represent a tiling with p sides per tile and q amount of sides that meet as a vertex. The tiling used in "Holonomy" is an Order-5 square tiling, i.e. a $\{4,5\}$ tiling. The old minimap implementation uses the Poincaré disk model to display it. The Poincaré

disk model projects hyperbolic space to a 2D unit circle [KMP10]. The advantage of the Poincaré disk model is that it is conformal, meaning that angles at all vertices where the lines of the tiling meet are preserved. Small shapes are also somewhat preserved, which makes it useful for displaying the rooms used in "Holonomy".
There are other methods to model Hyperbolic space, such as the Beltrami-Klein model [Cox98] or the Poincaré half-plane model [Sta08]. These are out of scope of this paper, since these projections of hyperbolic space are not intuitive to show how navigating in a hyperbolic space works. The Beltrami-klein model has the disadvantage that the size of nearby rooms is distorted a lot more than in the Poincaré disk projection, while the room size is even more distorted in the Poincaré half-plane model.

2.2. Known algorithms to compute hyperbolic tilings

Hyperbolica [Cod] is a game similar to "Holonomy", in the sense that the primary goal is of the game is to explore walking around in a hyperbolic space. Hyperbolica also allows the player to see their current position in the hyperbolic plane on a map. However, the tiling of this map is pre-computed for performance reasons [Hac23]. This makes it unsuitable for "Holonomy", since levels in "Holonomy" are not bounded like they are in Hyperbolica.
Fast algorithms for computing hyperbolic tilings could also be created by writing code executed on the graphics card. A graphics card is suitable for tasks which consist of lots of simple tasks. Since generating a tiling is basically just computing a lot of different lines, it is useful to investigate whether a program on the graphics card can quickly compute a $\{4,5\}$ tiling of the hyperbolic plane. One such shader that can compute tilings of the hyperbolic plane is the one by [mat19]. This particular shader is a fragment shader. This means that the program receives a fragment coordinate as input, and outputs the correct color for that pixel. An advantage of this shader implementation is that it is not too difficult to make it continuously moving. We just need to make sure that we send the player coordinates as input variables to the shader, and call `hyperTranslate` of the shader code with the correct coordinates, which lets the hyperbolic plane translate. The disadvantage of the shader approach is that it is not easily possible to assign a unique color to each generated tile. This is because there is no straightforward method to detect if a given pixel is in a specific tile of the world, meaning that we would only be able to generate a minimap in which each generated tile has the same color, which is not very useful for navigating a hyperbolic space.
Another interesting method of computing tessellations is the method used in Hyperrogue, a game that takes place on the hyperbolic plane with a top-down perspective [KC17]. Hyperrogue supports many different tilings, not just the $\{4,5\}$ tiling of $\mathbb{H}^2$. Hyperrogue generates its tilings via a datastructure called a Geodesic Regular Tree Structure (GRTS) [CK22], which is a table that defines the underlying tree structure of any tiling. A tiling of any space can be seen as a tree structure. Any tile can be seen as a child of the origin and possible in-between tiles by considering when that tile was generated by which tile. See Figure 5 for the tree structure used in "Holonomy".
Figure 4 shows the tree structure of a $\{4,5\}$ tiling of $\mathbb{H}^2$ in Hyperrogue, where the green edges represent different edges of the tree with the origin tile as the root. Tiles have the same number if

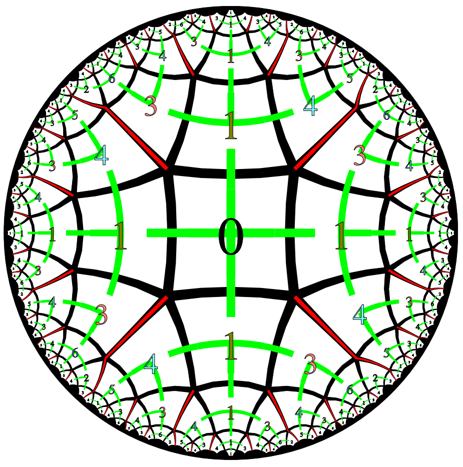
Figure 4: *Tree structure for a {4,5} tiling of hyperbolic space in Hyperrogue [Kop23]*

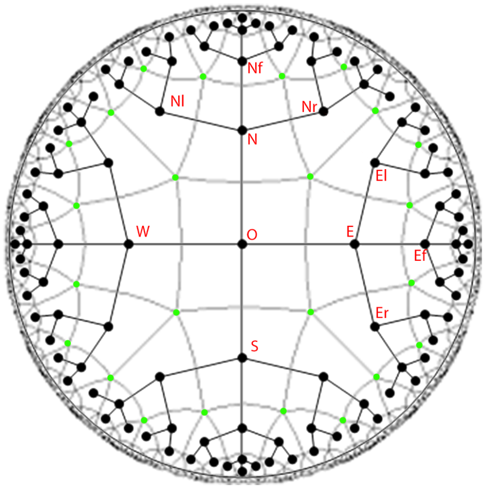

Figure 5: *Tree structure of "Holonomy". Each tile represents a room in the game, while the letters at each node represent the coordinates for that tile, which is a series of steps from the origin tile. For more information, see [YBS*22].*

their subtrees have the same shape, i.e. they are congruent. With this structure, a tiling can be generated lazily, i.e., a new tile is generated whenever we need it. Whenever a tile is generated, its information is saved as an object in computer memory, which contains information about its neighboring tiles as well as how the individual edges of the tile are constructed. To render the generated tiles, projections of hyperbolic space are used. The procedure is discussed in more detail by [CK22].
The problem with the Hyperrogue approach is that the method is too general for the scope of this paper. Hyperrogue supports a lot of different tilings of the hyperbolic plane, thus it also needs an efficient algorithm to compute many different tilings of the hyperbolic plane, whereas Holonomy has a completely different goal. Holonomy only takes place in a $\{4,5\}$ tiling of the hyperbolic plane.

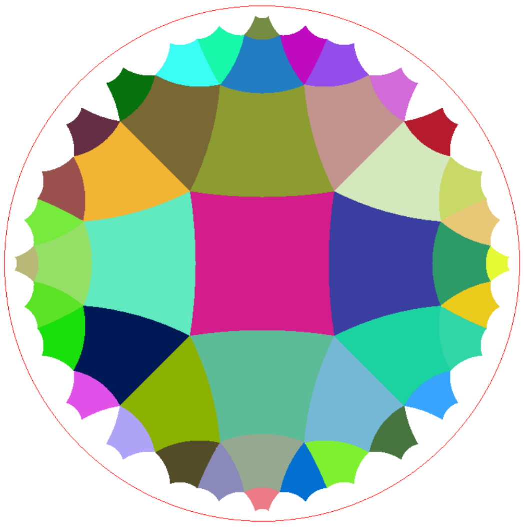

Figure 6: *Rendered result of running Algorithm 1 for 45 tiles*

Since we are only interested in a simple and fast minimap generation algorithm, the Hyperrogue method will not work for us. We need an algorithm which does not take a long time to understand and is easy to extend with other features. We can do this similarly to our Euclidean tiling example previously, we only need to concern ourselves with constructing an origin tile of the hyperbolic plane. Once we have this tile, we can reflect it along its edges to recursively build a tiling of the hyperbolic plane in the Poincaré disk projection. This is the algorithm that has been created, and we will get into that now.

3. Implementation

The main implementation is Algorithm 1. The algorithm generates an initial tile, then generates new tiles in a breadth-first manner. New tiles are generated by reflecting a tile along all of its edges. The implemented algorithm allows for generating an arbitrary amount of n tiles of $\mathbb{H}^2$. The old minimap makes use of 45 tiles, since it is rendered up to 3 steps away from the current tile of the user as seen in Figure 1. Since there are 11 tiles in each subtree from the origin tile, this means we have $4*11+1 = 45$ tiles. A rendered result of generating 45 tiles can be seen in Figure 6.
It is first necessary to explain some geometric formula's used to generate the tiling.

3.1. Geometric formula's

To invert a point A to point A' in circle c with centerpoint O and radius r, we make use of the definition of a circle inversion, the distance from O to A times the distance from O to A' must be equal to r^2 [Cox71].

$$OA \cdot OA' = r^2$$

Given 3 non-colinear points, it is possible to construct a circle that goes through those 3 points by solving a system of equations algebraically. Thus, given two points A and B in the Poincaré disk,

Algorithm 1 Algorithm to generate a Holonomy tiling with n tiles

```
Require: $n > 0, T$ is initialized ,$U$ is initialized
  $I \leftarrow$ GENERATEINITIALTILE()                ▷ This is Algorithm 2
  Add $I$ to $T$
  return If $n$ tiles have been generated
  $Q \leftarrow$ a Queue containing tiles, initialized empty
  for all $d \in$ { North, West, South, East } do
      $R \leftarrow I$.REFLECTINTODIRECTION$(d, U,$ don't care$)$    ▷ This is Algorithm 3
      Add $R$ to $T$
      return If $n$ tiles have been generated
      Add $R$ to $Q$
  end for
  while $Q$ is not empty do
      $C \leftarrow Q$.REMOVEMIN()
      for all $s \in$ { Forward, Left, Right } do
          if $C$.ISSTEPLEGAL$(s)$ then
              Convert $s$ to a direction $d$ based on $C$
              $R \leftarrow C$.REFLECTINTODIRECTION$(d, U, s)$
              Add $R$ to $T$
              return If $n$ tiles have been generated
              Add $R$ to $Q$
          end if
      end for
  end while
```

Algorithm 2 Algorithm to generate the initial tile

```
Require: InitialPoints, edges and $U$ are initialized
  edges $\leftarrow$ Array of size 4
  for all pairs of neighboring points i, j in InitialPoints do
      edge $\leftarrow$ CIRCLEBETWEENPOINTSINDISK$(i, j, U)$
      Add edge to edges
  end for
  return new HolonomyTile(edges, Direction.O, ε, no constraints)
```

Algorithm 3 Algorithm to reflect a tile into a certain direction

```
Require: edges and path are initialized, and dir, $U$, $s$ are given
  newEdges $\leftarrow$ Map of size 4
  $rc \leftarrow$ The circle of edges[dir]
  $o \leftarrow$ dir.OPPOSITE()
  newEdges[dir] $\leftarrow$ edges[o].REFLECTINTOEDGE$(rc, U)$
  newEdges[o] $\leftarrow$ edges[dir] with swapped start and endpoint
  for both $d$ of dir.ORTHOGONALS( )do
      newEdges[$d$] $\leftarrow$ edges[$d$].REFLECTINTOEDGE(rc, U)
  end for
  $p \leftarrow$path+$s$
  $G \leftarrow$ The new generation constraints based on $s$
  return new Holonomytile(newEdges, dir, $p$, $G$)
```

we can construct the hyperbolic line $\overrightarrow{AB}$ by inverting point A to get point A' and constructing the circle that goes through points A, B and A' [Goo01].
Next, we should explain the objects that make up the tiling.

3.2. Datatypes used in the implementation

The implementation makes use of two enumeration datatypes, Direction and Step. Direction is one of {North, West, South, East, Origin} and Step is one of {Forward, Left, Right, Don't care}. The options "Origin" and "Don't Care" are only present as dummy values for the initial tile.
An edge is an object in computer memory containing the following attributes:

- A circle c with a radius and centerpoint, that represents the circle to which a hyperbolic line in the Poincaré disk belongs.
- A startpoint and endpoint, which are both on the boundary of circle c, which represent that we should only draw this circle from the startpoint to the endpoint.

A HolonomyTile is an object in computer memory containing the following attributes:

- A mapping from direction to edges. The edges are saved in such a way that we can loop through the directions {North, West, South, East} to draw the polygon "without lifting the pen from the paper" intuitively speaking. This makes it easier to floodfill the tile with a color.
- A current forward direction.
- A string that is of regular expression $\varepsilon|d|ds^*$ informing the series of steps that was taken to generate this tile. Here, $d \in \{$ 'N', 'W', 'S', 'E'$\}$ and $s \in \{$ 'F', 'L', 'R'$\}$.
- Generation constraints given as booleans describing if 1) a left step has occurred in this path, 2) if we have seen a right step before another left step and 3) if the last step taken to generate this tile was a right step.

A HolonomyTiling is an object in computer memory which has the following attributes:

- Integer $P = 4$.
- Integer $Q = 5$.
- A constant 4x3 array `StepToDirection`, which is shown in Table 1.
- A unit circle U, which is the circle that represents the Poincaré disk. This should be initialized with centerpoint $(0,0)$ and radius equal to half the length of the viewport (on what we actually render our computed minimap).
- A list of points that are the vertices of the initial tile. Upon construction, the HolonomyTiling object should receive a parameter `initialRotation` that specifies how much the initial tile is rotated as an angle with the positive x-axis. The tiling used in "Holonomy" (the one in Figure 6) corresponds to an `initialRotation` of $\frac{\pi}{4}$.
- The list of known tiles T, initialized as empty.

3.3. Detailed explanation of Algorithm 1

Algorithm 1 is a method of HolonomyTiling. The list of initial points is calculated upon constructing the HolonomyTiling object.

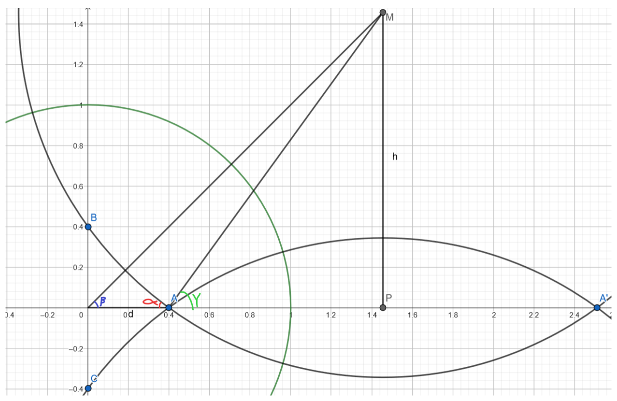

Figure 7: *Geometric constructions to calculate the initial distance d*

The first point is generated at a distance d from the origin with angle `initialRotation` with the positive x-axis. We first have to derive the formula of d.
To explain the derivation of d, we first need to draw some geometric constructions shown in Figure 7. The points A, B and C represent initial points such that the angle $\angle CAB$ is $\frac{\pi}{q} = \frac{\pi}{5}$. A' is the inversion of A within the unit circle (the green circle). The circle with centerpoint M is the circle which represents the hyperbolic line $\overrightarrow{AB}$. P is the midpoint of A and A'. We know that $\alpha = \frac{1}{2} \cdot \frac{2\pi}{q} = \frac{\pi}{q} = \frac{\pi}{5}$, $\beta = \frac{1}{2} \cdot \frac{2\pi}{p} = \frac{\pi}{p} = \frac{\pi}{4}$ and $\gamma = \pi - \alpha - \frac{\pi}{2} = \frac{\pi}{2} - \alpha$. The line segment we want to know the length of is $\overrightarrow{OA}$, which is d. P is the middle point of A and A'. We also know that $\overrightarrow{OA'}$ has length $\frac{1}{d}$, since A' is the inversion of A. Thus, we know that $\overrightarrow{OP} = \frac{d+\frac{1}{d}}{2}$. Now, we express h in terms of β, γ and lines $\overrightarrow{OP}$ and $\overrightarrow{AP}$. This results in

$$h = \tan\beta \cdot \overrightarrow{OP}$$
$$h = \tan\gamma \cdot \overrightarrow{AP}$$

Since we know β, γ and can express $\overrightarrow{OP}$ and $\overrightarrow{AP}$ in terms of d, we can use this system of equations to get a formula for d. This will result in the formula:

$$d = \sqrt{\frac{\cot(\frac{\pi}{q}) - \tan(\frac{\pi}{p})}{\cot(\frac{\pi}{q}) + \tan(\frac{\pi}{p})}}$$

This derivation is based on the one by [Chr]. Once we have d and `initialRotation`, we just need to convert from polar coordinates to Cartesian coordinates to get the position of the first point. The subsequent 3 points are similarly generated a distance d from the origin, by adding $\frac{\pi}{2}$ to `initialRotation` for each subsequent point.
Upon calling Algorithm 1, the initial tile is constructed as outlined in Algorithm 2. In Section 3.1, we described how we could construct a circle between points in the disk. Algorithm 3 shows what happens when calling `ReflectIntoDirection`. Here, the opposite direction is defined to be 180 degrees from the direction which it is called upon, and the orthogonal directions are the two 90 degree directions. So for North, the opposite direction is South, and

Step / **Current forward direction**	Forward	Left	Right
North	North	West	East
West	West	South	North
South	South	East	West
East	East	North	South

Table 1: *Step to direction conversion table for a HolonomyTiling object*

the orthogonal directions are West and East. `ReflectIntoEdge` will construct the new hyperbolic line by reflecting `this` edge into the given reflection circle. This is done by inverting the start and endpoint of `this` edge into the reflection circle to obtain start' and end'. Then we construct the hyperbolic line from end' to start' using the construction discussed in Section 3.1. The reason that this line is from end' to start' is because reflecting a $\{4,5\}$ tile in any circle causes the drawing direction to reversed as well. We need to be sure that we can draw the edge "without lifting the pen from the paper" by looping through the edges {North, West, South, East}, so that's why in our newly constructed tile, each edges[dir] will have the startpoint end' and the endpoint start'. To determine if a step is legal, so if a tile should be generated in a certain step, the underlying tree structure of a $\{4,5\}$ tiling is used as shown in Figure 5. A $\{p,q\}$ tiling of $\mathbb{H}^2$ in general can be described as an infinite tree as discussed in Section 2.2. The following two rules are applied when deciding if a tile should be generated in a certain step [Lav17]:

- No two consecutive "Right" steps are allowed
- No two consecutive "Left" steps are allowed without at least one "Right" step in between.

Each tile contains a field that tells what the current forward direction of that tile is. This is the direction that was taken to generate this tile as shown in Algorithm 3. We can think of this as an observer "looking at" the current forward direction, and deciding where to "walk" next. So when we need to expand from the current tile, we first need to check if the step we want to take is legal. Based on our two rules, a Forward step is always legal. A Left step is legal if it is the first Left step in the path of this tile or if we have seen any Right step before this Left step (the boolean describing if we have seen a Right step is set to false again after we have taken a Left step). Finally, a Right step is legal as long as the last step was not a right step.
Then, if we have assessed that the step is legal, we convert that step to a direction in which we can reflect the current tile based on the current forward direction. This conversion table is Table 1. Again, one can think of this table as an observer "facing the current forward direction" and deciding where to go next. If, for example, that observer takes a "Left" step while facing "West", that observer is going in the "South" direction in absolute terms.
Finally, to give each tile a color, loop through the generated edges of the tile in the order {North, West, South, East} to specify a region of the viewport that needs to be colored. To decide what color to give to a tile, use a hash function on the path of the tile. Then, to actually draw the tiles on the viewport, first draw the unit circle which is saved as an attribute of HolonomyTiling. Then, loop

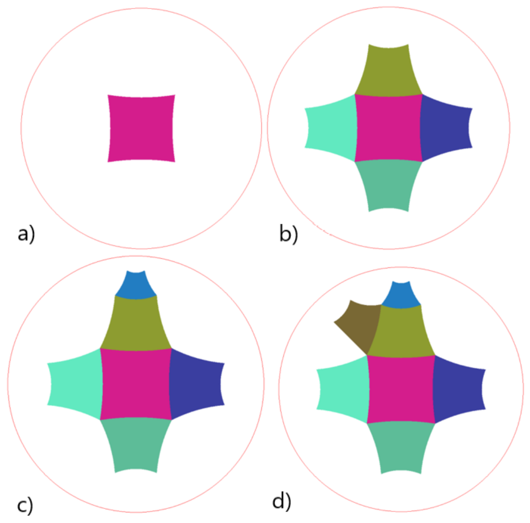

Figure 8: *Minimap generation process. a) Initial tile generated b) 4 Immediate neighboring tiles generated c) and d) Breadth-first expansion of new tiles in the directions of "Forward" and "Left" respectively*

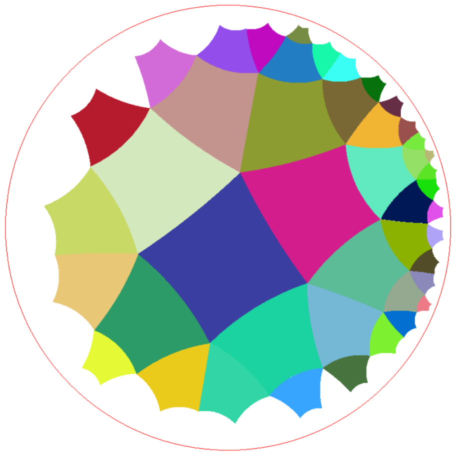

Figure 9: *Tiling with 45 tiles translated to point* $(140,40)$

through all generated tiles in T, and loop through each edge of the tile in the order {North, West, South, East}. Each edge is just a circle with two points that specify what part of that circle to draw, so we just need a graphics library that can draw circles for us. The entire process is shown visually in Figure 8.

3.4. Translating the tiling

To move the tiling to another place, we need to have a function for the HolonomyTiling object called `MoveInitialTile(B)`, such that now the initial tile is centered at point B instead of $(0,0)$. To do this, we need to construct a hyperbolic bisector (a hyperbolic middle line between 2 points) between $(0,0)$ and B such that we can reflect the initial points in this bisector to obtain new initial points

Algorithm 4 Algorithm to move the initial tile to a a new location centered at point B

```
Require: $B$ is not $(0,0), U$ is initialized and B is inside of $U$
    $B' \leftarrow$ InvertPoint$(B,U)$
    $M \leftarrow$ MidPoint$((0,0),B')$
    $c1 \leftarrow$ The circle with centerpoint $M$ that goes through $(0,0)$
    $I \leftarrow$ The two intersections between $c1$ and the unit circle
    $c2 \leftarrow$ The circle that goes through B' and the first of the intersection points of $I$
    Update the initial points of the tiling to be the originally constructed initial points inverted in $c2$
```

centered at B. Then, we can call Algorithm 1 to generate the tiling from point B instead of point $(0,0)$.
The procedure is outlined in Algorithm 4. It should be noted that circle $c2$ is constructed with Construction 1.6 from [Goo01]. A tiling with 45 tiles which is translated to the point $(140,40)$ can be seen in Figure 9. This translation can be controlled by saving a global variable B and increasing/decreasing components of B based on user input.

4. Evaluation and discussion

One can compare the old and new minimaps by comparing Figures 1 and 6. What can immediately be seen is that the old minimap is not actually a faithful Poincaré disk projection. In Section 2.1, we established that the Poincaré disk model should preserve angles. In a tiling of $\mathbb{H}^2$, every tile is the same size, thus all angles have to be of the same size as well. This is not the case in Figure 1, while it is the case in Figure 6. This means that the new minimap is a more faithful representation of the game world of "Holonomy".
We also want to compare the old and new implementations in terms of complexity and in terms of runtime in order to check if the new implementation is simpler and faster than the older one. To compare in terms of complexity, code metrics are calculated for both the old and new implementations. These metrics are calculated using Visual Studio 2022. Cyclomatic complexity (CC) is the amount of linearly independent paths through a piece of code (lower is better). For individual functions, McCabe, the creator of the metric, interprets a value between 0-10 as little risk, a value between 11-20 as moderate risk, values between 21-50 as high risk and values above 50 as very high risk. [McC08]. However, the values we calculate for CC indicate the sum of the CC for each method in that class, since we want to measure how difficult it is to grasp what a class is doing. Class coupling (CO) is the amount of classes which a single class uses (lower is better). Again, we measure this per class since we want to measure how difficult a specific class is to grasp. We also measure the maintainability index (MI) since it is a common metric to measure the relative maintainability of a piece of code [Wel01], which is a good indicator for understanding the code. For the MI, higher is better. It is a number from 0 to 100, see [Jon22] for a more extensive explanation. 0-9 indicates poor maintainability, 10-19 indicates moderate maintainability and 20-100 indicates good maintainability. Both the old and new implementations were written in the C# programming language, it is thus fair to compare them to one another with these metrics.

	MI	CC	CO	LOSC	LOEC
Circle	67	7	4	62	29
Direction	91	1	0	13	2
DirectionUtils	80	10	2	39	4
Geodesic	65	12	4	81	20
GeomUtils	63	16	4	200	76
HolonomyTile	69	18	19	182	37
HolonomyTiling	66	25	18	192	73
MainForm	72	14	23	117	38
Step	91	1	0	11	2
Mean	73.8	11.6	8.2		
Weighted mean	67.0	16.8	12.1		
Standard deviation	10.3	7.4	8.5		
Sum				897	281
Old minimap	58	33	42	314	162

Table 2: *Code metrics for the implemented solution. MI=Maintainability index, CC=Cyclomatic complexity, CO=Class coupling, LOSC=Lines of source code, LOEC=Lines of executable code*

To compare in terms of runtime, elapsed time is measured with the `System.Diagnostics.Stopwatch` C# class for the parts of the code which generate the minimap. This includes both computing it, as well as drawing it on a viewport output. This is done 10 consecutive times, such that we can be sure that the runtime is consistent and does not get influenced by external factors.

4.1. Code metrics

The computed code metrics are in Table 2. The rows "Circle" to "Step" represent code of the new solution, with their mean, weighted mean and standard deviation computed in the next 2 rows. We compute a weighted mean, since some classes skew the values quite a bit in one direction, such as the Direction and Step class. The weight of each class for computation of the weighted mean is the fraction of lines of executable code, since that gives a good idea how much work the class is trying to do. The old minimap was implemented in a single file, and its metrics are shown in the last row. It can be seen that the new solution is slightly better in terms of code metrics than the old solution. The weighed mean of CC and CO for the new minimap are quite a bit better than the value for the old minimap. The MI value for the new minimap is also a bit better than the old minimap. We can see that taking a weighted mean instead of the mean makes the value worse for all three metrics, since there are a few classes which unfairly improve the mean for the new implementation.
Lower values for the CC and CO indicate that the new minimap implementation is able to be easier grasped, since the classes generally try not to do too many things. The lower CO also indicates that the new minimap is more organized than the old one, since a low CO indicates that a class will generally do a single thing. In other words, different functionalities are not coupled as much as in the old implementation.
While the MI did improve, it did less so than the CC and CO. It is still an improvement, so based on the fact that the CC and CO improved quite a bit, we can conclude that the new implementation

	Old minimap	New minimap
Runtime of 10 different executions	17, 17, 19, 18, 19, 18, 19, 17, 17, 18	25, 27, 26, 25, 26, 25, 25, 25, 27, 25
Standard deviation	0.83	0.80
Mean	17.9	25.6

Table 3: *Runtime comparison of the old and new algorithms. Values are in milliseconds*

is slightly more maintainable.
The total of both lines of source code and the lines of executable code are greater in the new implementation than the old one. This isn't necessarily a problem however, since all individual classes of the new solution have a lower amount of lines than the total of the old implementation. Finally, we note that that the new implementation is well documented in this paper, which increases the ability for it to be grasped.

4.2. Runtime comparison

Result of the runtime comparison procedure is seen in Table 3. It can be seen that the old minimap is slightly faster than the new minimap. This is to be expected, as the old minimap makes use of a compute shader to compute some of the necessary textures for the minimap. So to make the new implementation faster, one would have to dispatch some of the computations to a compute shader. This would happen for a method like `ReflectIntoEdge` and all of the geometric formula's from Section 3.1, since graphics cards are suited for lots of simple computations like those. Implementing a compute shader was not achieved for the new implementation, since the research initially focused on deriving a simple algorithm, which ended up already taking enough time.

4.3. Unwanted behaviour

There are a couple of instances where tile edges are computed twice in slightly different manners. This happens because they are computed twice via different inversion circles. For example, the edge between the tiles "North, Left" and "West, Right" is one such edge (the edge between the orange and brown tile in Figure 10). It is first created by reflecting the "North" tile into its "West" edge, then later again by reflecting the "West" tile into its "North" edge. These reflections should result in exactly the same edge, but sometimes they are slightly different, possibly due to floating-point errors. Another such example is the east edge of tile "west, forward, forward, right" A work-around would be to ink the borders of each tile black with a sufficient width, such that it isn't noticeable to the user. This does make the tiling slower to draw though.
Translation of the tiling as described in Section 3.4 does not appear to work as expected for the vertical direction when $x = 0$ for point B. It does work as expected when point B has a non-zero x-component and a non-zero y-component, just not when it has a zero x-component and a non-zero y-component.

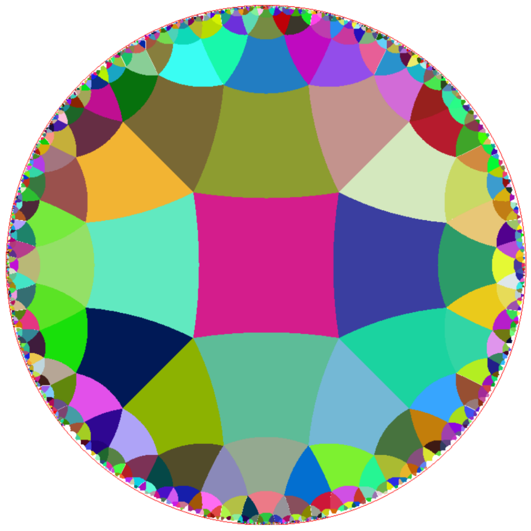

Figure 10: *Rendered result of running algorithm 1 for 1000 tiles*

5. Conclusion and future work

In short, we have a shown a new minimap generation algorithm for the game "Holonomy". It can draw the game world of "Holonomy" more faithfully than the old implementation, preserving angles of the hyperbolic plane. It is also able to translate the tiling to a new point within the unit circle based on user input. Code metrics and discussion in Section 4.1 shows that the new implementation is simpler than the old implementation. It will thus be easier to extend the new implementation with other features than the old implementation. The new minimap is slightly slower than the old minimap though.
Immediate future work would focus on implementing fixes for the unwanted behaviour described in Section 4.3. It would also be interesting to research if the derived algorithm could eventually be faster than the old algorithm, by implementing parts of the algorithm using compute shaders. To make the minimap continuously rendering based on player movements, the procedure described in Section 3.4 can be expanded upon to incorporate player movements. Finally, it would be interesting to evaluate whether a continuously rendering minimap in "Holonomy" would actually improve how players move through hyperbolic space. This could be done by comparing performance of players using a non-continuous minimap against players who use a continuous minimap.

6. Responsible research

Making a simpler and faster minimap has few ethical implications in the context of this research. Of course, if the given implementation were to be used in an actual game or application, such as "Holonomy", this map should not mislead or actively try to disorientate the user. This should not happen if an implemented version of the algorithm works as intended.
The research that has been conducted is reproducible, since the implementation of the discussed algorithm is completely described in section 3. Each step of the algorithm is explained: it is either dis-

cussed in Sections 1 and 2 or there has been given a reference which explains why the given step works as intended. Then what remains is the data gathered and discussed in Section 4. The only code metric that Visual Studio 2022 generated which was not shown in the paper was the "Depth of Inheritance". This was not very interesting to show, since both the old and new algorithms do not make use of inheritance (other than boilerplate code), and as a consequence this parameter would have been "1" for all classes, having no influence on the result. The runtime measurement of both implementations was run on the same machine one after another, which means external factors could not have influenced the result. It is also explained how the data was gathered.

Acknowledgements

We would like to thank Eryk Kopczyński for providing an extensive explanation on how Hyperrogue computes hyperbolic tessellations and for providing Figure 4. We would also like to thank Scott Jochems for providing Figure 5.

References

[Ben01] Bennett, Andrew G. *Hyperbolic Geometry | Mathematical Association of America*. July 2001. URL: https://www.maa.org/press/periodicals/loci/joma/hyperbolic-geometry (visited on 06/16/2023) 1.

[Chr] Christersson, Malin. *Non-Euclidean Geometry: Interactive Hyperbolic Tiling in the Poincaré Disc*. Malin Christersson's Math Site. URL: https://www.malinc.se/noneuclidean/en/poincaretiling.php (visited on 04/30/2023) 5.

[CJT11] Conway, John H., Jiao, Yang, and Torquato, Salvatore. "New family of tilings of three-dimensional Euclidean space by tetrahedra and octahedra". *Proceedings of the National Academy of Sciences* 108.27 (July 5, 2011). Publisher: Proceedings of the National Academy of Sciences, 11009–11012. DOI: 10.1073/pnas.1105594108. URL: https://www-pnas-org.tudelft.idm.oclc.org/doi/10.1073/pnas.1105594108 (visited on 06/16/2023) 2.

[CK22] Celińska-Kopczyńska, Dorota and Kopczyński, Eryk. *Generating Tree Structures for Hyperbolic Tessellations*. Mar. 16, 2022. DOI: 10.48550/arXiv.2111.12040. arXiv: 2111.12040[nlin]. URL: http://arxiv.org/abs/2111.12040 (visited on 06/02/2023) 3.

[Cod] CodeParade. *Hyperbolica on Steam*. URL: https://store.steampowered.com/app/1256230/Hyperbolica/ (visited on 04/25/2023) 3.

[Cox71] Coxeter, H. S. M. "Inversive Geometry". *Educational Studies in Mathematics* 3.3 (1971). Publisher: Springer, 310–321. ISSN: 0013-1954. URL: https://www.jstor.org/stable/3482030 (visited on 06/23/2023) 4.

[Cox98] Coxeter, H. S. M. *Non-Euclidean Geometry*. Google-Books-ID: usKZpDAH0WUC. Cambridge University Press, Sept. 17, 1998. 362 pp. ISBN: 978-0-88385-522-5 1, 3.

[DP01] Darken, Rudolph and Peterson, Barry. "Spatial Orientation, Wayfinding, and Representation". *Handbk Virtual Environ* 2002 (Nov. 22, 2001). DOI: 10.1201/b17360-24 2.

[Goo01] Goodman-Strauss, Chaim. "Compass and Straightedge in the Poincaré Disk". *The American Mathematical Monthly* 108.1 (2001). Publisher: Mathematical Association of America, 38–49. ISSN: 0002-9890. DOI: 10.2307/2695674. URL: https://www.jstor.org/stable/2695674 (visited on 06/17/2023) 5, 7.

[GS77] Grunbaum, Branko and Shephard, Geoffrey C. "Tilings by Regular Polygons". *Mathematics Magazine* 50.5 (1977). Publisher: Mathematical Association of America, 227–247. ISSN: 0025-570X. DOI: 10.2307/2689529. URL: https://www.jstor.org/stable/2689529 (visited on 06/16/2023) 2.

[Hac23] HackerPoet. *HyperEngine*. original-date: 2022-03-24T23:44:36Z. Apr. 22, 2023. URL: https://github.com/HackerPoet/HyperEngine (visited on 04/30/2023) 3.

[Jon22] Jones, Mike. *Code metrics - Maintainability index range and meaning - Visual Studio (Windows)*. Apr. 30, 2022. URL: https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning (visited on 06/21/2023) 7.

[KC17] Kopczynski, Eryk and Celinska, Dorota. "HyperRogue: Playing with Hyperbolic Geometry". (July 2017). URL: https://www.researchgate.net/profile/Dorota-Celinska-Kopczynska/publication/336702574_HyperRogue_Playing_with_Hyperbolic_Geometry/links/5dae0eb2299bf111d4bf8e60/HyperRogue-Playing-with-Hyperbolic-Geometry.pdf 3.

[KMP10] Kinsey, L. Christine, Moore, Teresa E., and Prassidis, Efstratios. *Geometry and Symmetry*. Google-Books-ID: fF-puDwAAQBAJ. John Wiley & Sons, Apr. 19, 2010. 960 pp. ISBN: 978-0-470-49949-8 3.

[Kop23] Kopczynski, Eryk. *Questions about Hyperrogue for a Bachelor's thesis project*. E-mail. June 4, 2023 3.

[Lav17] Lavrov, Misha. *Answer to "Description of the order-5 square tiling of the hyperbolic plane as a graph"*. Mathematics Stack Exchange. Apr. 13, 2017. URL: https://math.stackexchange.com/a/2231612 (visited on 06/13/2023) 6.

[mat19] mattz. *Hyperbolic Wythoff explorer*. June 16, 2019. URL: https://www.shadertoy.com/view/wtj3Ry (visited on 05/04/2023) 3.

[McC08] McCabe, Thomas. "Software quality metrics to identify risk". Nov. 2008. URL: https://web.archive.org/web/20220329072759if_/http://www.mccabe.com/ppt/SoftwareQualityMetricsToIdentifyRisk.ppt 7.

[Sta08] Stahl, Saul. *A Gateway to Modern Geometry: The Poincaré Half-plane*. Jones and Bartlett Publishers, 2008. 270 pp. ISBN: 978-0-7637-5381-8 3.

[Wee21] Weeks, Jeff. "Body coherence in curved-space virtual reality games". *Computers & Graphics* 97 (June 1, 2021), 28–41. ISSN: 0097-8493. DOI: 10.1016/j.cag.2021.04.002. URL: https://www.sciencedirect.com/science/article/pii/S0097849321000443 (visited on 06/17/2023) 1.

[Wel01] Welker, Kurt D. "The Software Maintainability Index Revisited". (2001) 7.

[YBS*22] Yarar, Baran, Bakker, Bo, Snellenberg, Ravi, et al. *"Holonomy": a non-Euclidean labyrinth game in virtual reality*. Tech. rep. TU Delft, 2022. URL: http://resolver.tudelft.nl/uuid:60d473f0-f327-411e-a402-95d44e27f088 1, 3.