Warsaw University of Technology

FACULTY OF
MATHEMATICS AND INFORMATION SCIENCE

Bachelor's diploma thesis

in the field of study Computer Science and Information Systems

Procedural world generation and the challenges of rendering in non-Euclidean spaces

Aleksy Bałaziński

student record book number 313173

Karol Denst

student record book number 305962

thesis supervisor

Paweł Kotowski, Ph.D.

WARSAW 2024

Abstract

Procedural world generation and the challenges of rendering in non-Euclidean spaces

Procedural generation, as a method used in creating video games, has experienced a significant increase in popularity in recent years. It has been employed extensively in acclaimed titles such as *Minecraft* and *No Man's Sky* to create virtually infinite worlds that the player is free to interact with. On the other hand, a recent game *Hyperbolica* popularized a novel idea of making the virtual worlds even more interesting by setting them in non-Euclidean spaces. This work describes the implementation of a video game that incorporates both of the aforementioned concepts.

In the game, we use and expand on the technique of transforming Euclidean space into hyperbolic or spherical space introduced by László Szirmay-Kalos and Milán Magdics in [15]. The approach allows for the use of established algorithms, developed with Euclidean geometry in mind, in other geometries. Furthermore, core parts of the game logic can be shared between different geometries, which vastly simplifies the implementation. The approach also allows for changing the degree to which the scene is visually curved in hyperbolic geometry at runtime.

The procedural terrain generation in the game is based on the marching cubes algorithm with Perlin noise used for generating the underlying scalar field.

The game also includes several classic game features, such as terrain editing, character animations, day/night cycle, and many more. As a result, the game offers a unique experience of exploring and interacting with worlds inherently different from the one we live in.

Keywords: non-Euclidean geometry, hyperbolic geometry, spherical geometry, procedural terrain generation, video games, OpenGL, C#

Streszczenie

Proceduralne generowanie świata i wyzwania związane z renderowaniem w przestrzeniach nieeuklidesowych

Proceduralne generowanie świata jest techniką zdobywającą rosnącą popularność przy tworzeniu gier komputerowych. Zostało ono wykorzystane z powodzeniem m. in. w takich produkcjach jak *Minecraft* czy *No Man's Sky*. Z drugiej strony za przyczyną gier takich jak *Hyperbolica*, szersze zainteresowanie zyskała kwestia osadzania gier komputerowych w przestrzeniach nieeuklidesowych. Niniejsza praca opisuje implementację gry komputerowej, która łączy oba te podejścia.

W grze zastosowano, z pewnymi modyfikacjami, metodę przekształcania przestrzeni euklidesowej w przestrzeń hiperboliczną lub sferyczną, zaproponowaną przez László Szirmay-Kalosa i Milána Magdicsa w [15]. Wspomniane podejście pozwala na wykorzystanie znanych algorytmów, zaprojektowanych z myślą o geometrii euklidesowej w innych geometriach. Dodatkowo, kluczowe części logiki gry mogą być dzięki temu dzielone pomiędzy różnymi geometriami, co znacznie upraszcza implementację. Zastosowana metoda pozwala również na zmianę stopnia zakrzywienia sceny w czasie rzeczywistym w przestrzeni hiperbolicznej.

Proceduralne generowanie terenu w grze opiera się na algorytmie maszerujących sześcianów (ang. *marching cubes algorithm*) z polem skalarnym określonym za pomocą szumu Perlina.

Gra zawiera również wiele klasycznych funkcjonalności, takich jak edycja terenu, animacje postaci, cykl dnia i nocy oraz wiele innych. W rezultacie gra oferuje unikalne doświadczenie eksploracji i interakcji z światami fundamentalnie różnymi od tego, w którym żyjemy.

Słowa kluczowe: geometria nieeuklidesowa, geometria hiperboliczna, geometria sferyczna, proceduralne generowanie terenu, gry komputerowe, OpenGL, C#

Contents

1. Introduction

The main goal of this thesis is to create a video game that combines the concepts of procedural generation and non-Euclidean geometry. Procedural generation is a method used to create the world by using algorithms instead of manually designing it. It was first used in the 1980s in games such as *Elite* [1] and *Rogue* [17] and has achieved a significant increase in popularity with the release of *Minecraft* [12] in 2011. Unlike procedural generation, which has a long and rich history in the video game industry, the concept of employing non-Euclidean geometries is a relatively novel idea. It has been used in only a few games, most notably in *Hyperbolica* [2], which was released in 2022. Non-Euclidean geometry differs from the usual Euclidean geometry in that it does not satisfy the parallel postulate – one of the postulates given by Euclid. This postulate states that given a line and a point not on the line, there is exactly one line through the point that does not intersect the given line. In non-Euclidean geometry, there can be zero or more than one such line. In the first case, the geometry is called spherical and in the second case, it is called hyperbolic. Changing this one postulate has a profound effect on the geometry of the space, which is what makes the concept of non-Euclidean geometry so intriguing. A world where the parallel postulate does not hold is almost impossible to imagine, which our game aims to help with. It is also a very unique concept that helps the game stand out from the crowd.

To make the game more fun to play, we included several features, such as NPCs (non-playable characters, also known as *bots*) in the form of humanoids that the player can interact with, a car that the player can drive around, and items that the player can use to perform various actions.

The game implementation of non-Euclidean space is based on an article by Szirmay-Kalos and Magdics [15], which outlines a way to adapt game engines to the rules of curved spaces. We take that core idea from this work and expand on it by adjusting the implementation of hyperbolic geometry to allow for infinite spaces necessary to facilitate terrain generation.

This document describes the implementation of the game in the following chapters:

- Functional Specification – describes the game from the user's perspective and outlines the requirements for the game.
- Theoretical foundations – describes the theoretical foundations of the game.

- Implementation – describes the implementation of the game.
- Testing – describes the testing of the game.
- User Manual – describes how the player can interact with the game.
- Results – describes the results of the work focusing on the performance, the gameplay and the depiction of non-Euclidean spaces.
- Problems – describes the problems encountered during the implementation of the game.
- Improvements – describes the possible improvements to the game.

1.1. Work distribution

This section describes how the tasks were divided among the authors in terms of implementing the application and writing the document. The work distribution for the application is presented in Table 1.1 and the work distribution for the document is presented in Table 1.2. It is important to note that all parts of the project were worked on by both authors. Each line in the application source code (and each sentence of the thesis) was written by one of the authors was reviewed by the other author in a pull request. Moreover, both authors made bug fixes and small changes to all parts of the project. These tables only show who the main author of each part was. Overall, the work was distributed evenly among the authors. This can be seen in a screenshot from the Insights section of the game's GitHub repository in Figure 1.1, which shows that the authors have the same number of commits.

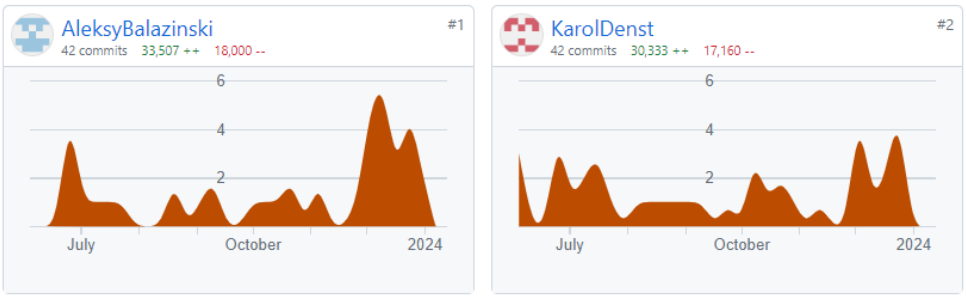


Figure 1.1: GitHub contributions (`https://github.com/Non-Euclidean-World/Hyper/graphs/contributors`)

Task	Author
Implementation of the game logic	Both
Implementation of the graphics engine	Both
Implementation of terrain editing	Both
Implementation of the spherical geometry	Aleksy Bałaziński
Implementation of the hyperbolic geometry	Aleksy Bałaziński
Integration of the physics engine with the rest of the system	Aleksy Bałaziński
Implementation of the terrain generation	Karol Denst
Implementation of the user interface (HUD & Menu)	Karol Denst
Implementation of character models and animations	Karol Denst

Table 1.1: Work distribution for the application

Chapter	Section	Author
Introduction		Both
Work distribution		Both
Functional Specification		Both
Theoretical foundations	Non-Euclidean geometry	Aleksy Bałaziński
Theoretical foundations	Marching Cubes	Karol Denst
Theoretical foundations	Models	Karol Denst
Theoretical foundations	Day night cycle	Aleksy Bałaziński
Theoretical foundations	Lighting	Aleksy Bałaziński
Implementation	Technologies selection	Aleksy Bałaziński
Implementation	Game objects management	Aleksy Bałaziński
Implementation	Procedural world generation	Karol Denst
Implementation	Chunk management system	Aleksy Bałaziński
Implementation	Rendering	Aleksy Bałaziński
Implementation	User interface	Karol Denst
Testing		Karol Denst
User Manual		Karol Denst
Results		Aleksy Bałaziński
Problems		Karol Denst
Improvements		Karol Denst

Table 1.2: Work distribution for the document

2. Functional Specification

The functional requirements are described in the form of user stories and listed in Table 2.1.

As a player I want to	**So that**	**Flags**
Start a game in Euclidean space	I can explore Euclidean spaces	Must
Start a game in hyperbolic space	I can explore hyperbolic spaces	Must
Start a game in spherical space	I can explore spherical space	Must
Have gravity in the game	I have Earth-like experience	Must
Be able to collide with other objects in the game	I can interact with other objects	Must
Be able to move around	I can explore the world	Must
Be able to jump	Moving around is easier	Must
Be able to shoot projectiles	I can see how things move in non-Euclidean spaces	Must
Have NPCs move around	The world looks richer	Must
Have NPCs shoot projectiles	The game is more interesting	Must
Edit the terrain	I can view any structure in different spaces	Must
See the marker of the terrain editor	I can decide where the terrain modification is going to take place	Must
Have different terrains generated every time I load a game	I can explore different terrains	Must
Have light sources that illuminate other objects	I can see what lighting looks like in different spaces	Must
Have a vehicle I can enter	I can move explore the world faster	Must
Have an inventory	I can manage different items	Must
Be able to save a game using a GUI	I can save interesting worlds	Must
Be able to delete a save using a GUI	I can remove old saves	Must
Be able to load a game using a GUI	I can revisit interesting worlds	Must

As a player I want to	**So that**	**Flags**
Be able to interact with NPCs	So that there is more to do in the game	Should
See a night/day cycle	I can compare the world during the day and the night	Should

Table 2.1: Functional requirements

3. Theoretical foundations

3.1. Non-Euclidean geometry

The *Euclidean* geometry is based on a set of five postulates originally given by Euclid. The postulates read as follows [20]:

1. A straight line segment can be drawn joining any two points.
2. Any straight line segment can be extended indefinitely in a straight line.
3. Given any straight line segment, a circle can be drawn having the segment as the radius and one endpoint as the center.
4. All right angles are congruent.
5. Given any straight line and a point not on it, there exists one and only one straight line that passes through that point and never intersects the first line, no matter how far they are extended. [16]

The fifth postulate, also called the *parallel postulate*, has been for hundreds of years a subject of debate if it can be proven from the former four postulates. It was discovered, however, that the negation of the parallel postulate doesn't lead to a contradiction [5]. The postulate can be negated in one of two ways.

- Given any straight line and a point not on it, there exist **at least two straight lines** that pass through that point and never intersect the first line, no matter how far they are extended.
- Given any straight line and a point not on it, there exists **no straight line** that passes through that point and never intersects the first line; in other words, there are no parallel lines, since any two lines must intersect.

When the parallel postulate is replaced with the first statement, we obtain a new geometry, called the *hyperbolic geometry*. Similarly, replacing it with the second statement yields the *spherical geometry*. These geometries are collectively referred to as *non-Euclidean geometries*.

3.1.1. Analytic description

To describe the non-Euclidean geometries analytically, we will follow the approach given by [15]. This approach allows us to view the points of a 3-dimensional non-Euclidean space as a subset of the 4-dimensional *embedding space*. Since imagining the fourth dimension is not something particularly easy, we will be decreasing the dimensionality whenever we give examples.

The elliptic space can be modeled as a unit 3-sphere, *embedded* in a 4-dimensional Euclidean space. By saying that a space is embedded in another, we mean that the embedded space inherits the distance from the embedding space. In this case, the spherical distance, given by $ds^2 = dx^2 + dy^2 + dz^2 + dw^2$ is derived from the Euclidean distance. This is similar to how we may model the 2-dimensional elliptic space as a sphere, where lines are identified with great circles. The inner product of two vectors u and v in the Euclidean space is given by

$$\langle u, v \rangle_E = u_x v_x + u_y v_y + u_z v_z + u_w v_w.$$

Thus, we can define that a point p belongs to the elliptic geometry if

$$\langle p, p \rangle_E = 1.$$

The model that we use for the hyperbolic geometry is the so-called *hyperboloid model*. In this model, points p of the hyperbolic space satisfy the equation

$$p_x^2 + p_y^2 + p_z^2 - p_w^2 = -1,$$

with $p_w > 0$. The set of these points creates the upper sheet of a hyperboloid, which could be visualized as shown in Figure 3.1 if the embedding space was Euclidean. However, the

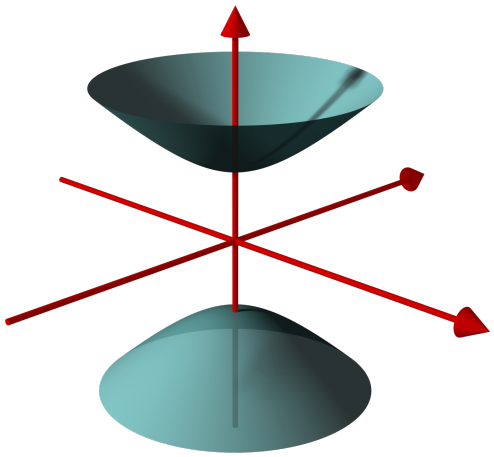


Figure 3.1: 2-dimensional hyperboloid embedded in Euclidean space (source: `https://commons.wikimedia.org/wiki/File:Hyperboloid2.png`)

hyperbolic space is not embedded in the Euclidean space, but in the *Minkowski space* instead. In the Minkowski space, the inner product of vectors u, v is given by the Lorentzian inner product:

$$\langle u, v\rangle_L = u_x v_x + u_y v_y + u_z v_z - u_w v_w.$$

Thus, the points p belonging to the hyperbolic geometry satisfy the equation

$$\langle p, p\rangle_L = -1.$$

It could be interpreted that they are located on a sphere with a radius of imaginary length $\sqrt{-1}$ (and hence are equidistant from the origin).

To build a unified framework for discussing both types of geometries, we introduce the notion of *sign of curvature*, $\mathcal{L}$, that attains the value $+1$ for spherical, and -1 for hyperbolic space. We also define the generalized inner product

$$\langle u, v\rangle = u_x v_x + u_y v_y + u_z v_z + \mathcal{L} u_w v_w. \tag{3.1}$$

3.1.2. Transformations

We will now define transformations that can be used in non-Euclidean geometries.

Reflection

A vector v_R obtained from reflecting vector v on vector m, see Figure 3.2, can be defined as

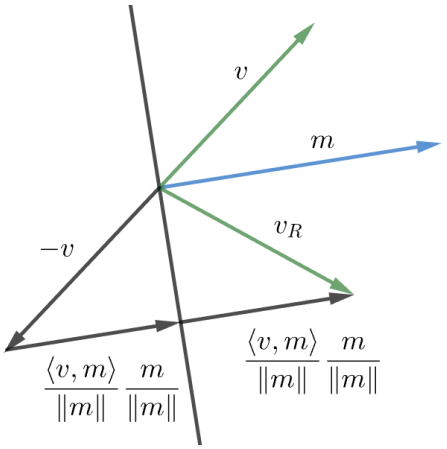


Figure 3.2: Reflection of v on vector m

$$v_R = 2\frac{\langle v, m\rangle}{\|m\|}\frac{m}{\|m\|} - v = 2\frac{\langle v, m\rangle}{\langle m, m\rangle}m - v.$$

It can be verified that this definition satisfies the intuitive conditions of a reflection:

- The reflected vector v_R lies in the plane spanned by v and m,

- The transformation is an isometry, i.e. $\|u - v\| = \|u_R - v_R\|$.

We should also note that given a point p in the geometry, i.e. satisfying $\langle p, p\rangle = \mathcal{L}$, its reflection, p', is also in the geometry.

Translation

Just like in Euclidean space, we can define translation in terms of an even number of reflections. More specifically, the translation will be defined by specifying two points: *geometry origin*, $g = (0, 0, 0, 1)$ and *translation target*, q, which is the point that the geometry origin is translated to. Now we can define that the translation is the composition of two reflections: one on the vector $m_1 = g$ and the second one on the vector $m_2 = g + q$, which is halfway between g and q. Applying the first reflection to an arbitrary point p gives a point

$$p' = 2\frac{\langle p, g\rangle}{\langle g, g\rangle} g - p = 2p_w g - p,$$

and the second reflection applied to p' yields a point

$$p'' = 2\frac{\langle p', g + q\rangle}{\langle g + q, g + q\rangle}(g + q) - p' = 2p_w q + p - \frac{p_w + \mathcal{L}\langle p, q\rangle}{1 + q_w}(g + q). \tag{3.2}$$

The effect of applying translation to an arbitrary point a is shown in Figure 3.3.

It can be verified that the geometry origin g is indeed translated to point $g'' = q$. We can

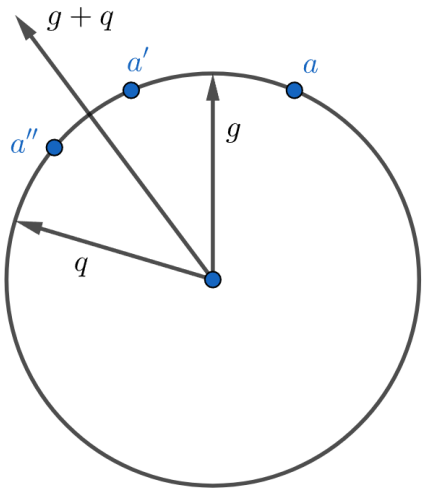


Figure 3.3: Translation of a point a

evaluate the formula for the basis vectors $i = (1, 0, 0, 0)$, $j = (0, 1, 0, 0)$, $k = (0, 0, 1, 0)$, and

$l = (0, 0, 0, 1)$ obtaining the translation matrix

$$T(q) = \begin{bmatrix} 1 - \mathcal{L}\frac{q_x^2}{1+q_w} & -\mathcal{L}\frac{q_x q_y}{1+q_w} & -\mathcal{L}\frac{q_x q_z}{1+q_w} & -\mathcal{L}q_x \\ -\mathcal{L}\frac{q_y q_x}{1+q_w} & 1 - \mathcal{L}\frac{q_y^2}{1+q_w} & -\mathcal{L}\frac{q_y q_z}{1+q_w} & -\mathcal{L}q_y \\ -\mathcal{L}\frac{q_z q_x}{1+q_w} & -\mathcal{L}\frac{q_z q_y}{1+q_w} & 1 - \mathcal{L}\frac{q_z^2}{1+q_w} & -\mathcal{L}q_z \\ q_x & q_y & q_z & q_w \end{bmatrix} \tag{3.3}$$

It can be seen that the translation is an isometry since the row vectors of the matrix are orthonormal.

Rotation

It can be shown that a rotation about an axis through the origin is the same as the Euclidean rotation about the same axis [13].

Camera transformation

The camera transformation allows us to describe the scene from the viewer's perspective. The transformation is defined in terms of the *eye position* e, and three orthonormal vectors in the tangent space of the eye:

1. the right direction i',
2. the up direction j', and
3. the negative view direction k'.

An example in Figure 3.4 shows the tangent space of the eye, with the e vector marked green, $-k'$ marked blue, and i' marked orange.

The transformation can be described by the matrix

$$V = \begin{bmatrix} i'_x & j'_x & k'_x & \mathcal{L}e_x \\ i'_y & j'_y & k'_y & \mathcal{L}e_y \\ i'_z & j'_z & k'_z & \mathcal{L}e_z \\ \mathcal{L}i'_w & \mathcal{L}j'_w & \mathcal{L}l'_w & e_w \end{bmatrix} \tag{3.4}$$

As the result of the transformation, the eye position is mapped to the geometry origin g. Furthermore, the vectors i', j', and k' are mapped to i, j, and k, respectively.

Perspective transformation

The perspective transformation is described using a projection matrix P. The projection matrix we use in spherical geometry is identical to the one used in the *Unity* implementation of

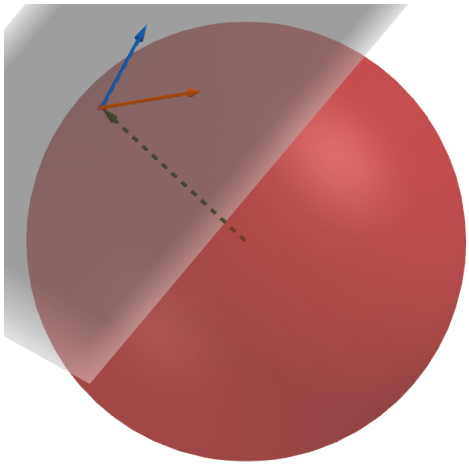


Figure 3.4: Tangent space of the camera

[15] (see `https://github.com/mmagdics/noneuclideanunity`). It is parameterized by the *near plane distance* n, *far plane distance* f, *aspect ratio* ASP, and *field of view* FOV:

$$P = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & -n & 0 \end{bmatrix},$$

where $s_x = 2n/(r-l)$, $s_y = 2n/(t-b)$, and r, l, t, b are defined in terms of $u = f\tan(\text{FOV})$:

$$r = u \cdot \text{ASP},\ l = -u \cdot \text{ASP},\ t = u,\ b = -u.$$

For hyperbolic and Euclidean geometries, the standard projection matrix is used.

Porting objects

The positions of objects in the scene are specified in a 3-dimensional Euclidean space. They are then "transported" or *ported* to a non-Euclidean space of choice. One possible mapping that could be used for this purpose is called the exponential map. For a given point p in the 3-dimensional Euclidean space with coordinates (x, y, z) the mapping to elliptic geometry is given by

$$\mathcal{P}_E(p) = (p/d\sin(d), \cos(d)), \tag{3.5}$$

and for hyperbolic space, it is given by

$$\mathcal{P}_H(p) = (p/d\sinh(d), \cosh(d)),$$

where $d = \|p\|$. The effect of porting a 1-dimensional point p onto a 1-dimensional elliptic space can be seen in Figure 3.5.

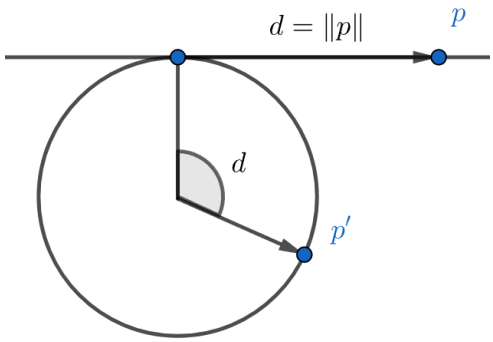


Figure 3.5: Exponential map

Porting vectors

A vector v starting at a point p can be ported to non-Euclidean space by translating it to a point $\mathcal{P}(p)$. Hence the ported vector v' is given by

$$v' = (v, 0)T(\mathcal{P}(p)), \tag{3.6}$$

where T is the translation matrix 3.3.

3.1.3. Practical considerations

There are two ways we could implement placing objects in the scene:

1. Port the object to non-Euclidean geometry and then use the translation given by Equation 3.2,

2. Translate the object using ordinary Euclidean translation and then port it to non-Euclidean geometry.

We will now shortly discuss both options[1].

Port, then translate

The first option is undesirable, as it may significantly change the relative positions of objects in the scene. To see why, let's consider two copies of a 2-dimensional rectangle that we will first port to the spherical geometry, and then translate using the non-Euclidean translation. The rectangle with vertices $a = (-0.5, -0.7)$, $b = (0.5, -0.7)$, $c = (0.5, 0.5)$, $d = (-0.5, 0.5)$ is ported

[1]The discussion is based on the results obtained using the Python script `https://github.com/Non-Euclidean-World/Thesis/blob/main/scripts/distortions_calc.py`

to spherical geometry using Equation 3.5. As a result, we obtain points on a unit sphere:

$$\mathcal{P}(a) = (-0.441, -0.617, 0.652),$$
$$\mathcal{P}(b) = (0.441, -0.617, 0.652),$$
$$\mathcal{P}(c) = (0.459, 0.459, 0.760),$$
$$\mathcal{P}(d) = (-0.459, 0.459, 0.760).$$

If we were to translate the first copy of the rectangle to point $t_1 = (0.5, 0.5)$ and the second copy to $t_2 = (1.5, 1.7)$ in Euclidean geometry, the two copies should meet at the point $(1, 1)$.

When we perform the translation to point t_1 (the corresponding translation target is obtained by porting t_1 using Equation 3.5, i.e. the translation target is $q_1 = \mathcal{P}(t_1)$), we get the following vertices:

$$\mathcal{P}(a)T_{q_1} = (-0.014, -0.190, 0.982),$$
$$\mathcal{P}(b)T_{q_1} = (0.761, -0.296, 0.577),$$
$$\mathcal{P}(c)T_{q_1} = (0.698, 0.698, 0.156),$$
$$\mathcal{P}(d)T_{q_1} = (-0.110, 0.809, 0.578).$$

The translation to t_2 (with $q_2 = \mathcal{P}(t_2)$) gives

$$\mathcal{P}(a)T_{q_2} = (0.709, 0.686, 0.160),$$
$$\mathcal{P}(b)T_{q_2} = (0.957, -0.031, -0.287),$$
$$\mathcal{P}(c)T_{q_2} = (0.141, 0.099, -0.985),$$
$$\mathcal{P}(d)T_{q_2} = (-0.117, 0.847, -0.519).$$

Even though we would expect the third vertex of the first copy of the rectangle to be identical to the first vertex of the second copy, there is a difference between the two. This effect can be seen in Figure 3.6. The effect is even more visible in the implementation. Figure 3.7 shows how the wheels of the car get misaligned from their wheel arches.

Translate, then port

The second option isn't unfortunately free of distortions either. For example, consider two identical squares of side length 0.5. The first one with the bottom-left corner at the point $(0, 0)$ and the second one with the corresponding corner at $(0.5, 0.5)$. After porting to spherical geometry using the Equation 3.5, we get squares with side lengths (listed counter-clockwise starting at the bottom edge):

0.500, 0.479, 0.479, 0.500

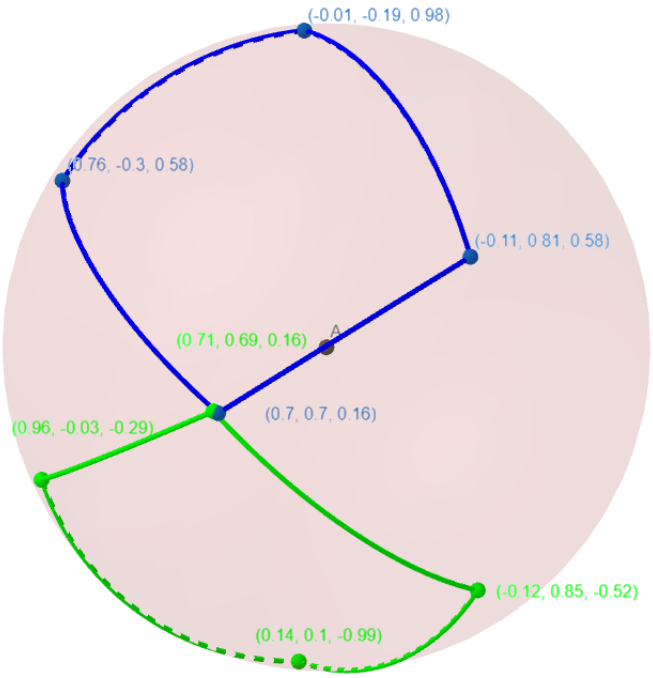


Figure 3.6: Rectangles ported onto a sphere and then translated

Figure 3.7: Non-Euclidean translation causing a misalignment of objects

for the first square and with side lengths

$$0.480,\ 0.425,\ 0.425,\ 0.480$$

for the second square. The side lengths of the square have been calculated as the lengths of geodesics[2] between the square's vertices. As we can see, the side lengths of the ported square are no longer equal to each other, and the distortion increases as the square is farther away from the origin. This effect can be seen in Figure 3.8.

[2]This is the "great-circle distance" equal to $2 \arcsin(c/2)$, where c is the chord length.

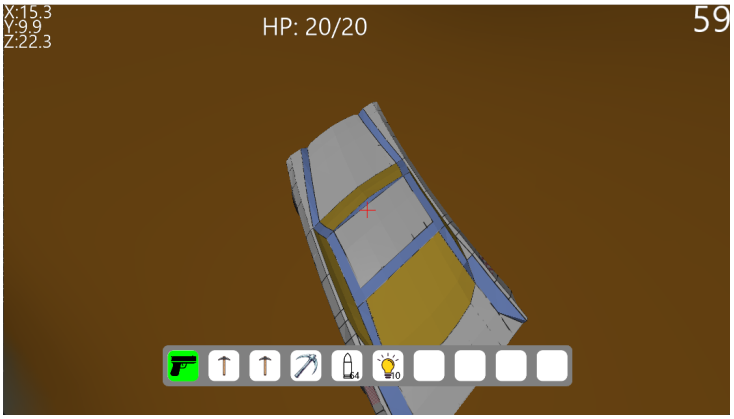


Figure 3.8: Distrotions caused by porting to spherical space

Spherical space

To minimize the distortions in spherical space we employed the following method. Due to the periodic nature of the porting given by Equation 3.5, the scene has to be confined inside a 3-dimensional ball of radius π. Since the distortions increase as an object is farther from the origin, we decided to split the scene into two physical regions – balls of radius $\pi/2$. Points p in the first ball (centered at the origin) are mapped to spherical geometry using the mapping

$$\mathcal{P}_{E,1}(p) = (p/\|p\| \sin(\|p\|), \cos(\|p\|)) \tag{3.7}$$

and points p in the second ball (which is centered at a point c) using the formula

$$\mathcal{P}_{E,2}(p) = (p'/\|p'\| \sin(\|p'\|), -\cos(\|p'\|)), \tag{3.8}$$

where $p' = p - c$. In the 2-dimensional case, the regions become disks with radii of length π, and the effect of using Equation 3.7 and Equation 3.8 can be visualized as "wrapping" the first disk on the upper half of a unit sphere, and "wrapping" the second one on the lower half of the sphere. We should note, that the implementation differs slightly from this description. More details will be covered in subsection 3.1.4.

Hyperbolic space

Dealing with distortions in hyperbolic space requires a more drastic approach because the scene we wished to port was potentially infinite. The main goal was to keep the distortions as small as possible near the camera and still show the visual aspects indicative of setting the scene in non-Euclidean geometry. To achieve this, the camera position is fixed at some point close to the origin, e.g. $(0, 1, 0)$. The movement of the camera is then simulated by moving all of the objects in the scene in the direction opposite to the camera's movement direction.

3.1.4. Teleporation in spherical space

Even though splitting the scene into two regions in spherical geometry allows us to minimize distortions significantly, it introduces a wide range of other problems. The most important one has to do with moving objects and the camera from one region to the other; we call this process *teleportation*.

Teleportation is schematically shown in Figure 3.9. In this 2-dimensional example, an object

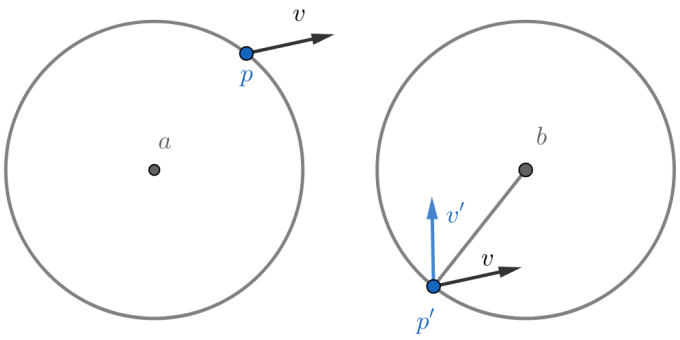


Figure 3.9: Teleportation between two regions

leaves the first region (centered at a) at the point p and is teleported to the second region (centered at b), appearing at a location given by the point p':

$$p' = b + R_{xz}(p - a),$$

where $R_{xz}(p)$ denotes reflection of a point p across the origin. The velocity vector v of the object is mapped to vector v' which is the reflection of v on a vector $p - a$.

The teleportation in the 3-dimensional case can be defined in a very similar manner. There are only two differences. The first one is that $R_{xz}(p)$ now denotes a reflection on the unit vector $\hat{y}$ (assuming positive y direction coincides with the "up" direction for the scene). The second difference is that v' is now the reflection of v through a plane through the origin orthogonal to $(p - a) \times \hat{y}$.

To account for the point reflection R_{xz}, the porting given by Equation 3.8 has to be modified by replacing p' with $R_{xz}(p')$.

Setting up the view transformation in the second region also comes with its own set of challenges. The standard way of obtaining the vectors i', j', and k' for the view matrix 3.4 is as follows. We first port the camera position to non-Euclidean space (in the case of the second region, we use Equation 3.8), and then use the Equation 3.6 to port its right, up, and front vectors. The problem with this approach is that the translation matrix 3.3 is defined in terms of translating the geometry origin $g = (0, 0, 0, 1)$ and not $(0, 0, 0, -1)$. This means that if the

translation target q is close to $(0,0,0,-1)$, $T(q)$ can map a point p with $p_w < 0$ to a new point p' with $p'_w > 0$ because

$$p'_w = -q_x p_x - q_y p_y - q_z p_z + q_w p_w \approx q_w p_w > 0.$$

The matrix isn't even defined for translation targets with $q_w = -1$.

To solve this problem, we derived a translation matrix $T_2(q)$ which we use for translations in the second region. It is defined analogously to $T(q)$ given by Equation 3.3, but in the context of T_2, the translation target q is redefined as the point that the point $(0,0,0,-1)$ is translated to. The matrix is given by

$$T_2(q) = \begin{bmatrix} 1 - \frac{q_x^2}{1-q_w} & -\frac{q_x q_y}{1-q_w} & -\frac{q_x q_z}{1-q_w} & q_x \\ -\frac{q_y q_x}{1-q_w} & 1 - \frac{q_y^2}{1-q_w} & -\frac{q_y q_z}{1-q_w} & q_y \\ -\frac{q_z q_x}{1-q_w} & -\frac{q_z q_y}{1-q_w} & 1 - \frac{q_z^2}{1-q_w} & q_z \\ -q_x & -q_y & -q_z & -q_w \end{bmatrix}. \tag{3.9}$$

Using the fact that q is in the spherical geometry, i.e. $\langle q, q \rangle_E = 1$, it can be verified that the row vectors of the matrix are orthonormal, thus the matrix describes an isometry. Moreover, $T_2((0,0,0,-1))$ is the identity matrix as expected.

3.2. Marching Cubes

One of the requirements for our project was to incorporate terrain generation. Numerous algorithms exist for this purpose, and the chosen algorithm for our project is known as *marching cubes*. This algorithm was selected due to its simplicity, versatility, and standardization. Marching cubes is an algorithm that can be easily modified to suit different requirements, as explained in detail in section 4.3. Furthermore, it is widely used in various applications and is well-documented, making its understanding and implementation easier.

The concept of marching cubes was first introduced by William E. Lorensen and Harvey E. Cline in 1987 [9]. The fundamental idea behind this algorithm is to generate a mesh from a scalar field. Then, the *isolevel* is chosen, 0 being the most common choice and the one we used. Points of the scalar field with values greater than the isolevel are considered to be "above" the surface, while points with values lower than the isolevel are considered to be "below" the surface. The world is divided into cubes, and for each cube, the algorithm determines which of its vertices are above and below the surface. Based on this information, a mesh is created to separate the vertices above the surface from those below it. An example of this process is illustrated in Figure 3.10, where $v0$ is below the surface, while the remaining vertices are above it. It is important to note

that the same effect would be achieved if $v0$ were above the surface and the other vertices were below it.

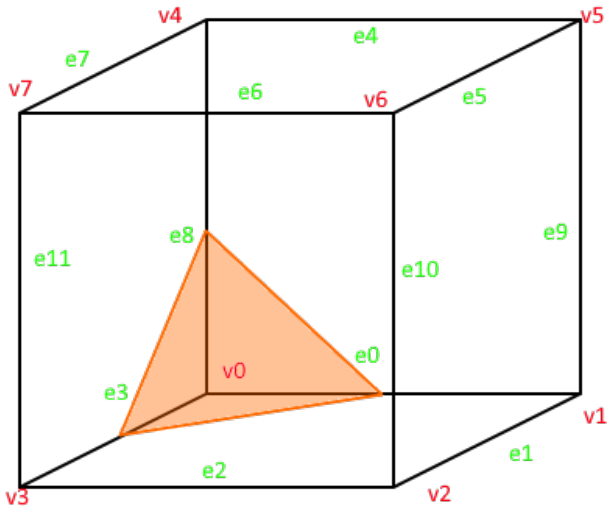


Figure 3.10: "Marching" cubes with only $v0$ below the surface. Source: `https://polycoding.net/marching-cubes/`

Each cube consists of 8 vertices, and each vertex is classified as either above or below the surface, resulting in a total of 256 possible combinations. However, there are only 15 unique combinations, all of them depicted in Figure 3.11, with the remaining combinations being rotations and reflections of these 15 cases. For each of these unique combinations, a precomputed table is utilized to generate the corresponding mesh. This table provides information about which edges of the cube are intersected by the surface and how to connect them.

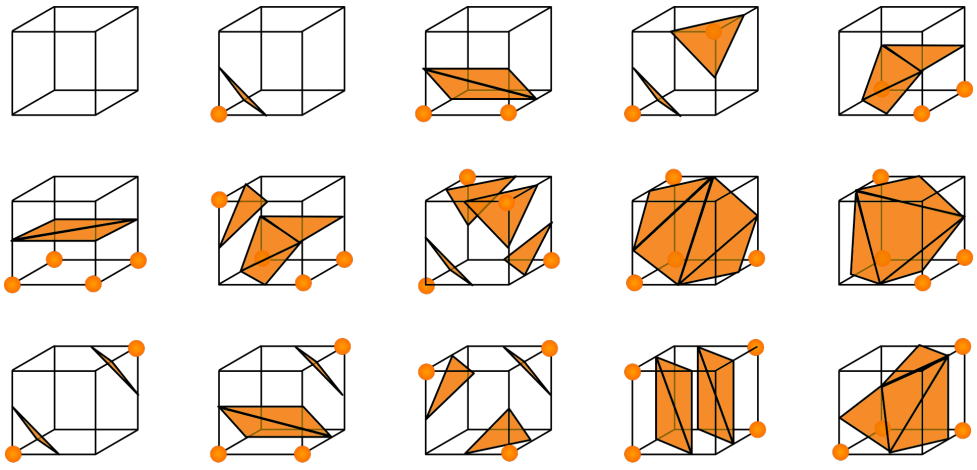


Figure 3.11: Unique marching cubes configurations. Source: `https://polycoding.net/marching-cubes/`

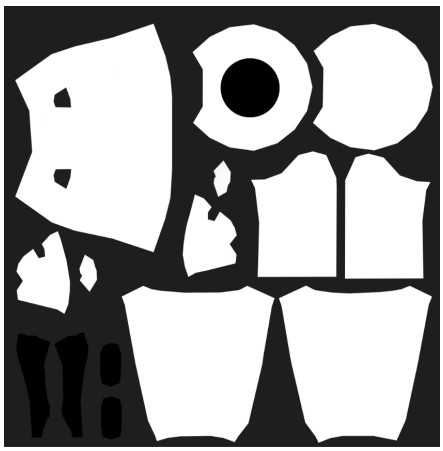
3.3. Models

There are two models used in the game: a character model and a car model.

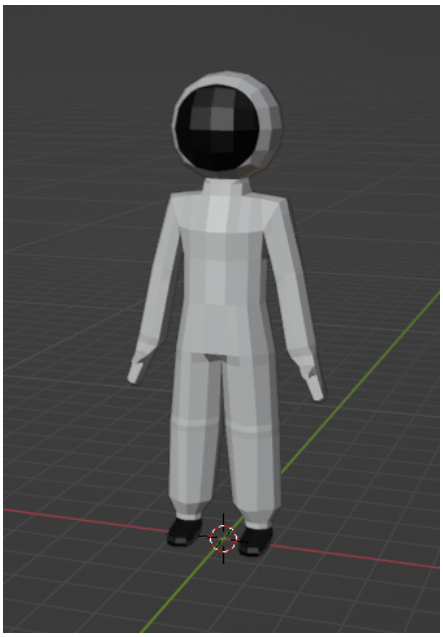
The character model consists of two parts: the mesh part and the animation part. What the two parts have in common is that they are both created in Blender, exported to a COLLADA file and loaded into the game using AssimpNet. Both of these parts are described in this section.

3.3.1. Model mesh

A model mesh contains vertices, normals, texture coordinates and indices of a model. Vertices are the points that make up the model and normals are automatically assigned by Blender. Texture coordinates are the coordinates of the texture that is mapped onto the model. The texture however is a two-dimensional image whereas the model is three-dimensional so the mapping is not trivial. The model mesh has seams that define how a mesh is unwrapped. These are selected edges that are marked as seams in Blender. Cutting along these seams will result in a set of faces that can be laid out flat. As a result, we get the so-called *UV map*, which defines how the texture is mapped onto the model. The UV map for the astronaut model is shown in Figure 3.12a. For example, the helmet is cut into two parts (the front and the back) and laid out flat in the top right corner of the texture. This texture describes the color of each pixel of the model. The whole model can be seen in Figure 3.12b.



(a) UV map



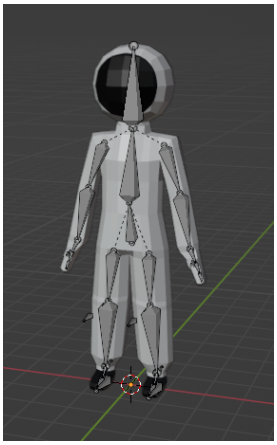
(b) Player model

Figure 3.12: Player model and the texture.

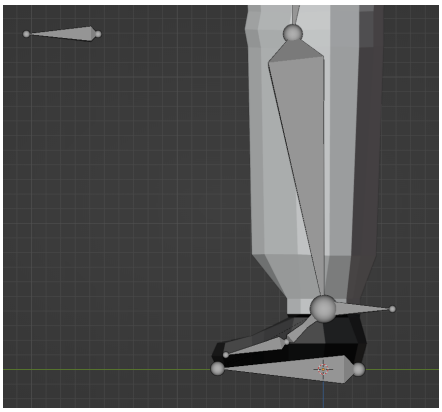
3.3.2. Model Animation

Model animations make use of a *model armature* (also called *skeleton*) which can be seen in Figure 3.13a. The armature is a set of bones that have different relations to each other and the mesh. The bones create a tree structure with the root being the torso bone. Most other bones are children of the torso bone or children of children of the torso bone. This relation allows the bones to inherit transformations from their parents. For example, if a leg bone moves the foot bone will follow.

It is worth noting that bone structure does not have to resemble the human bone structure. For example in Figure 3.13b we can see a set of bones that are outside of the body. These bones are not used to deform the mesh but to interact with other bones. Among other things, these guarantee that the knee does not bend in the wrong direction.



(a) Armature.



(b) Abnormal bones.

Figure 3.13: Armature.

The bones need to be connected to the mesh. Each vertex is assigned a set of bones that influence it with different weights. When bones move the vertices move with them according to how much they move and how much a certain bone influences a certain vertex. Assigning weights to vertices in Blender is shown in Figure 3.14. As can be seen in the figure, it is done by selecting a bone and then painting the vertices with a certain weight.

To define an animation a set of *keyframes* is used. The process of creating the keyframes can be seen in Figure 3.15. A keyframe is a set of transformations for each bone at a certain time. Once the keyframes are defined the animation is interpolated between them to produce a smooth animation. In the game, there are two animations: walking and running, both of which are looped to create a continuous animation when the player or an NPC moves.

The whole process of rendering an animation is as follows:

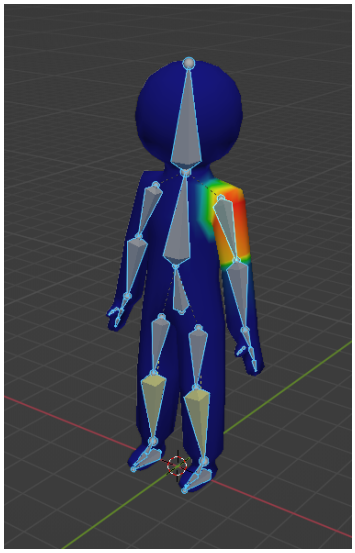


Figure 3.14: Vertex weights.

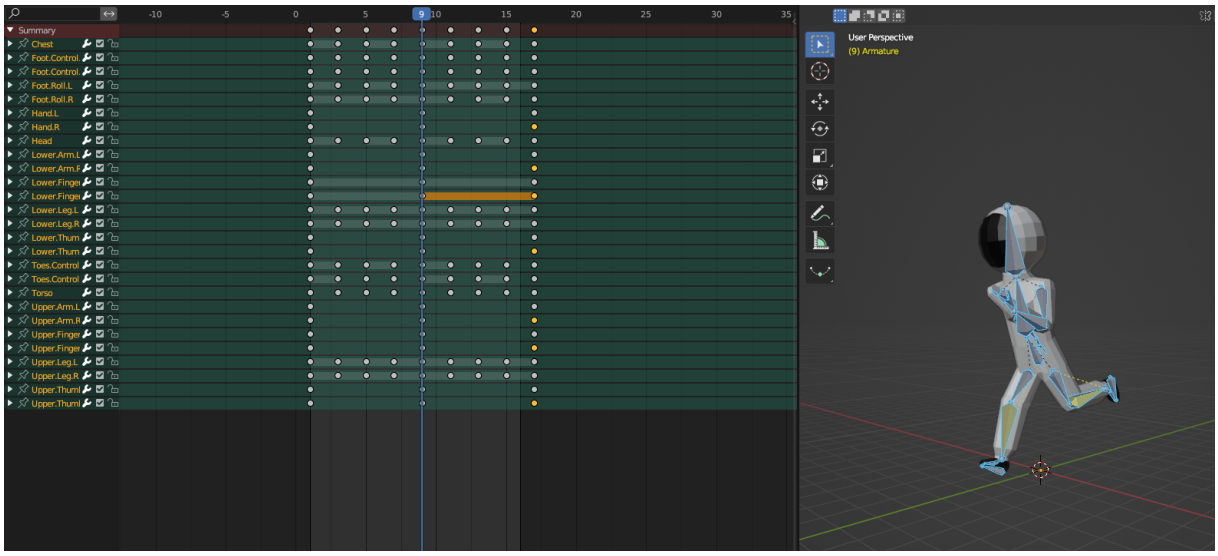


Figure 3.15: Keyframes.

1. The model along with the armature and keyframes are loaded into the game.
2. The timer is started.
3. The bone transformations for the current frame are calculated based on the keyframes and the timer, by interpolating between the keyframes.
4. The bone transformations are passed into the shader.
5. For each vertex the shader iterates over the bones that influence it and calculates the final position of the vertex based on the bone transformations and their weights.
6. The animation is rendered.

3.3.3. Car model

The car model consists of two sub-models: the wheel model and the body model. The wheel model is just a slightly modified cylinder.

The main difficulty in modeling the car's body is getting the proportions right. Otherwise, the model looks very unnatural. After some unsuccessful attempts to create the model free-hand, we decided to use blueprints of an existing car (1965 Ford Mustang) for modeling. The process of modeling the car based on blueprints (side view) is depicted in Figure 3.16.

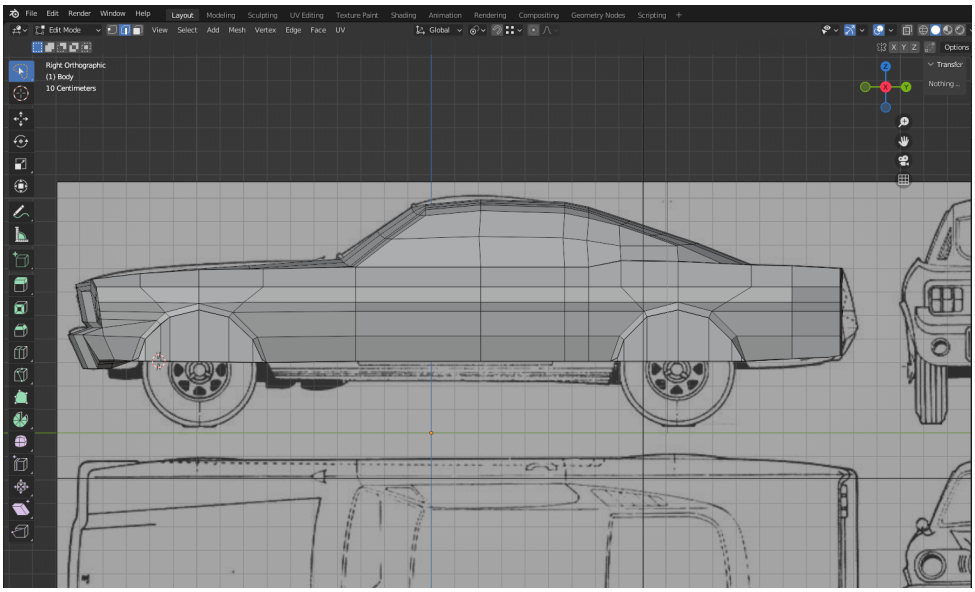
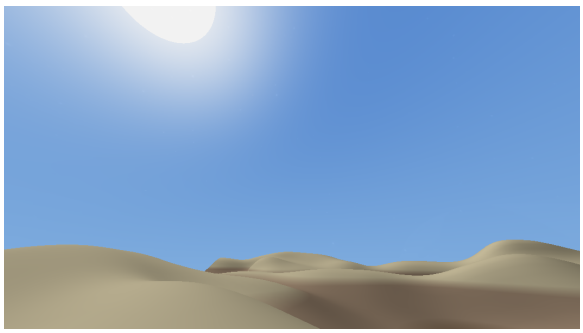


Figure 3.16: Modeling in Blender based on blueprints

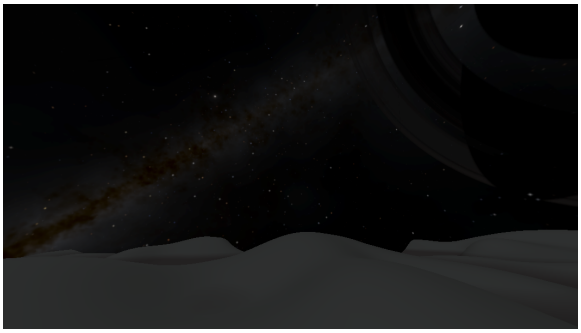
3.4. Day night cycle

Even though the game can be played in non-Euclidean spaces which makes it inherently unrealistic, we decided to add some elements that would make the scenes portrayed in the game feel familiar. In particular, one property of the real world that we wanted to capture in the game was the daytime cycle. In the game, the full cycle is 10 minutes long, with 5 minutes long daytime and 5 minutes long nighttime. We also added transitions between day and night to give the Earth-like experience of sunrise and sunset. The scene during various times of the day is shown in Figure 3.17.

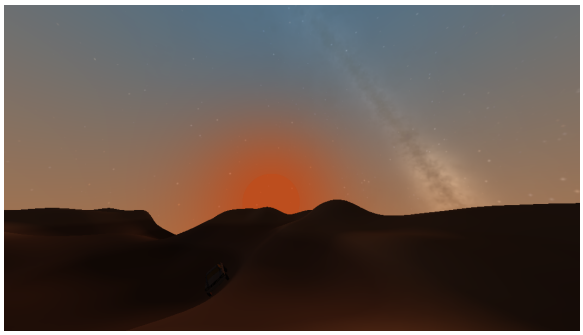
The implementation of the day-night cycle relies on two components: directional lighting (described in subsection 3.5.1) which corresponds to the light coming from the sun and a *skybox* representing the sky. Conceptually, a skybox is a cube made out of six images, one per each



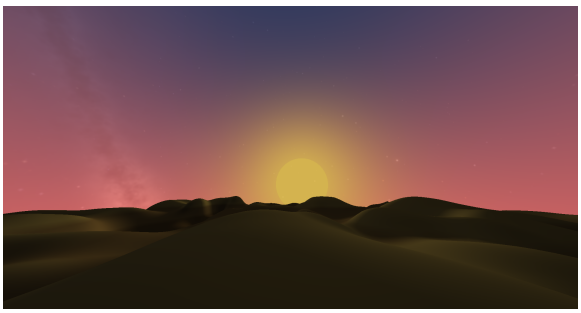
(a) Day



(b) Night



(c) Sunrise



(d) Sunset

Figure 3.17: Day night cycle in the game

side, that encompasses the scene thus creating an illusion that the world is much bigger than it is in reality. A skybox can be implemented in OpenGL using a special type of texture, a *cubemap*, i.e. a texture that contains six individual 2D textures. In the game, we're using images of the night sky obtained from an HDR file `https://www.reddit.com/r/blender/comments/3ebzwz/free_space_hdrs_1/` using an online utility program `https://matheowis.github.io/HDRI-to-CubeMap/`. The images were then slightly edited by applying Gaussian blur to make the stars appear larger. The vertices of the cube passed to the vertex shader are transformed using the model, view, and projection matrices. The model matrix is responsible for rotating the skybox (similarily to how stars appear to move across the night sky as the Earth is rotating). The view matrix has to be modified so that the skybox doesn't move along with the camera to create the illusion that it is very far away from the player. The part responsible for translation can be removed from the view matrix by replacing the last row the the view matrix with the vector $\begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}$ [18].

The day is split into *phases*, each characterized by the color of the sky at the zenith, the sky's color at the horizon, the color of the sun, and *stars' visibility factor*. In the fragment shader, we determine the color of each fragment. This is done by first obtaining the zenith and horizon colors by interpolating between the respective colors for the previous and the next phase based

on the current time. In the same manner, we obtain the current stars' visibility factor. Then, we obtain a sky color for a given fragment by interpolating between the zenith and horizon colors based on the height of the fragment[3]. The sun's position is given by a vector s that rotates at the same rate as the skybox. Calculating a dot product d of s with the current fragment's position allows us to easily draw the sun and the sun glare by mixing the sky's color with the sun's color in proportions depending on d. As the last step, we mix the pure-day-time color of the sky with the pure-night-time texture of the stars in proportions given by the stars' visibility factor.

The day-night cycle hasn't been implemented for the spherical space, as the terrain in the spherical space fully "encloses" the scene leaving no way of seeing anything "outside".

3.5. Lighting

Lighting is an important aspect of the game, making it more immersive and has a major impact on how it is perceived by the player. Artificial light sources present in the game are also indispensable when exploring the world during the in-game night. In the game, we use three types of light casters: *directional lights*, *point lights*, and *spotlights*. As the lighting model, we used the *Phong lighting model*. In this model, light is considered to have 3 components:

- ambient lighting I_a (with coefficient k_a),
- diffuse lighting I_d (with coefficient k_d),
- specular lighting I_s (with coefficient k_s and material shiness constant α).

In the case of N light sources in the scene, the total illumination is calculated using the formula

$$L = k_a I_a + \sum_{i=1}^{N} k_d I_{i,d} \max(0, \langle n, l_i, \rangle) + k_s I_{i,s} \max(0, \langle r_i, v \rangle^{\alpha}), \tag{3.10}$$

where n is the normal vector of the fragment, l is the vector pointing from the fragment to the light source, r is the reflection of l on n, and v is the vector pointing towards the viewer (all of the aforementioned vectors are assumed to be normalized). It's important to note that $\langle \cdot, \cdot \rangle$ is the inner product as defined by Equation 3.1. The reflected light vector r is calculated using the usual formula

$$r = 2\langle l, n \rangle n - l.$$

[3]The "position" of a fragment in this context is given by world space coordinates, normalized so that we're treating the points as located on a sphere (*skydome*). The height is then simply the y coordinate of the fragment's position.

3.5.1. Directional lighting

During the in-game daytime, the directional light is used to represent the light cast by the sun. The light direction l is the vector pointing toward the sun. During the night the directional light is much dimmer but still present. In this case, the light direction l is pointing toward an imaginary light source ("the stars") which is rotating along with the sky. Figure 3.18 shows the terrain and other game objects illuminated by the directional light of orange color.

Figure 3.18: Directional light

3.5.2. Point lights

In the game, spotlights are represented by white spherical lamps.

In Euclidean geometry, assuming that a point light is placed at a point p and that the current fragment's position is given by f, we can calculate the light direction vector l simply as

$$l = \frac{p - f}{d_E(p, f)},$$

where d_E is the usual Euclidean distance, i.e.

$$d_E(a, b) = \sqrt{\langle a - b, a - b \rangle_E}. \tag{3.11}$$

For non-Euclidean geometries, we use modified formulas given by [15], namely

$$l = \frac{p - f\cos(d_S(p, f))}{\sin(d_S(p, f))} \tag{3.12}$$

for spherical geometry and

$$l = \frac{p - f\cosh(d_H(p, f))}{\sinh(d_H(p, f))} \tag{3.13}$$

for hyperbolic geometry. The spherical and hyperbolic distances d_S and d_H are given by

$$d_S(a,b) = \cos^{-1}(|\langle a,b\rangle_E|) \tag{3.14}$$

and

$$d_H(a,b) = \cosh^{-1}(-\langle a,b\rangle_L) \tag{3.15}$$

respectively.

The important difference between a point light and a directional light is the *attenuation factor* a. Attenuation represents how the light's strength diminishes over distance. It can be expressed as a reciprocal of a quadratic function:

$$a = \frac{1}{K_c + K_l d + K_q d^2}, \tag{3.16}$$

where d is the distance of the fragment from the source that can be calculated using one of the formulas 3.11, 3.14, or 3.15 depending on the geometry and K_c, K_l, and K_q are constants specific to the light source's strength. Multiplying the light by the attenuation factor gives the desired effect of a realistic point light source such as a lamp, see Figure 3.19. Point lights in hyperbolic

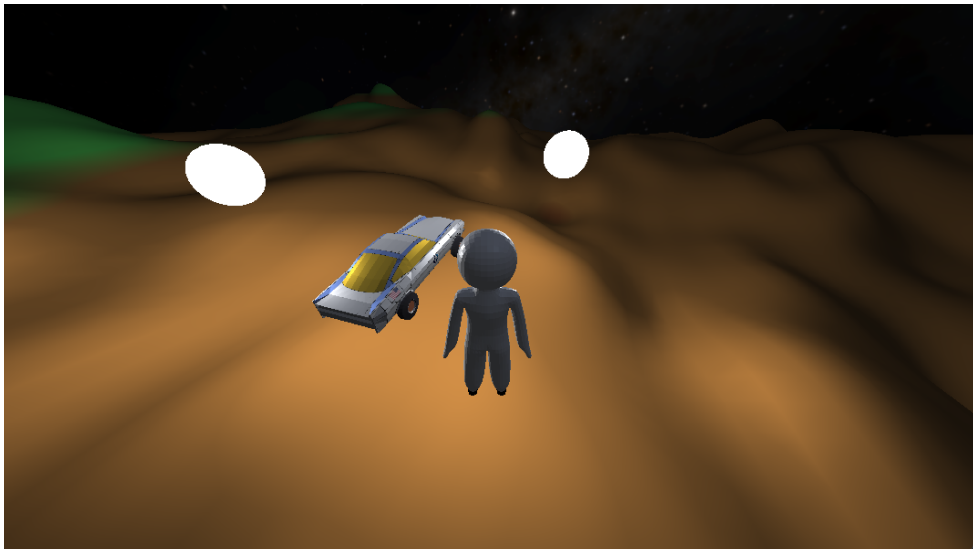


Figure 3.19: Point lights

and spherical geometry are shown in Figure 3.20.

3.5.3. Spotlights

In the game, spotlights are used to represent the player's flashlight and the car's head- and tail lights.

Spotlights are modeled in the same way as point lights, with only one exception. In the case of spotlights, we want to capture the fact that the light forms a cone. To do that we calculate

(a) Hyperbolic space

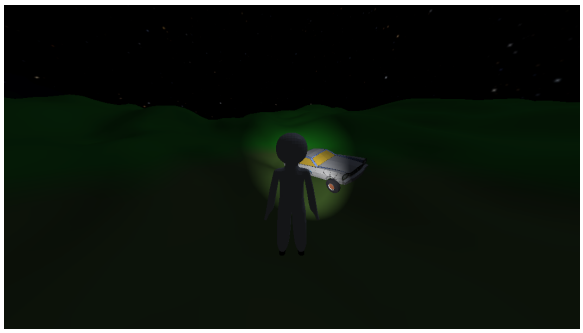
(b) Spherical space

Figure 3.20: Point lights in non-Euclidean spaces

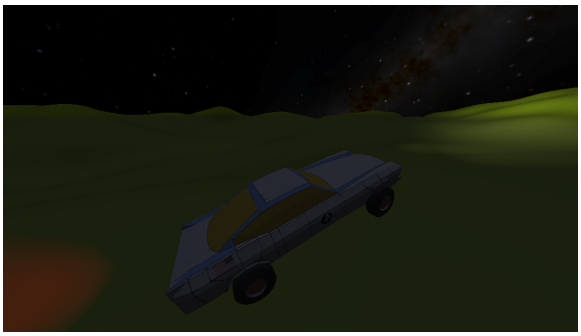
the *intensity coefficient* given by

$$\mathrm{IC} = \frac{\langle l, -d \rangle - R}{r - R},$$

where d is the vector along which the spotlight is directed, and R and r are the parameters defining the light cone [19]. The intensity coefficient is then used to multiply each component of the light, giving the result shown in Figure 3.21.



(a) Player's flashlight



(b) Car illumination

Figure 3.21: Spotlights used in the game

4. Implementation

The project that was created as the practical part of the thesis is the *Hyper* video game available at `https://github.com/Non-Euclidean-World/Hyper`. The *Hyper* video game is a relatively big project (it comprises around 260 classes), so naturally we have to contain the discussion to only the most important parts of the system. Nevertheless, in case in-depth information is needed, the detailed technical documentation can be accessed via `https://non-euclidean-world.github.io/Hyper/api/index.html`. Conceptually, there are four different "areas" spanned by the project:

1. game objects management,
2. procedural world generation,
3. rendering,
4. user interface.

In what follows, we will discuss these areas in more detail. We will also provide information on the technologies used in the project and present its core components.

4.1. Technologies selection

The video game is written in C# and uses OpenGL Shading Language (GLSL) for the shaders. The targeted platform is .NET 7 on Windows 10 and above.

The game relies on the following third-party libraries:

- OpenTK[1] – a set of low-level C# bindings for OpenGL; used for 3D rendering,
- BepuPhysics2[2] – a low-level, highly performant physics simulation library; used as the physics engine,

[1] `https://github.com/opentk/opentk`
[2] `https://github.com/bepu/bepuphysics2`

- SkiaSharp[3] – API for rendering 2D images; used for 2D rendering (menus, controls, etc.),
- AssimpNet[4] – a .NET wrapper for the Open Asset Import (Assimp) library[5]; used for loading models,
- StbImageSharp[6] – C# port of the `stb_image.h` header file; used for loading images.

4.2. Game objects management

A game object, such as a humanoid-looking bot or a car most often can be understood as existing in three different planes simultaneously:

1. the visual layer,
2. the physical layer,
3. the logical layer.

The visual layer of an object describes how the object looks like during the gameplay. This side of a game object we will call a *model*. The information about models (vertices, normal vectors, UV mappings, textures, and animations) is read from COLLADA and PNG files (see subsection 4.2.2) and stored in the `Model` class. It's important to note that if multiple game objects look the same, the model information is shared between them in the form of a single `ModelResource` class instance. More specifically, the `ModelResource` class is abstract, and its derived classes are singletons. Animated models additionally use the `Aminator` class (see subsection 4.2.1) which provides a way to calculate the bone transforms based on the elapsed time. Some models are *compound*. The car model, for example, stores information about the look of the wheels and body separately.

The physical layer defines the physical properties of the game object at any given moment. Some basic ones include inertia, angular and linear velocity, and friction; in the case of more complex objects such as a car, the description contains information about *constraints*. The constraints define the relative positions of different elements of the object and their velocities. An example of a constraint would be the wheel axes of the car. It's worth noting that the position constraints don't have to be "stiff" (they are modeled as springs). Technically, once a game object is added to the simulation, it is represented by one (or more in the case of compound bodies)

[3] `https://github.com/mono/SkiaSharp`
[4] `https://github.com/MonoGame/AssimpNet`
[5] `https://assimp.org/`
[6] `https://github.com/StbSharp/StbImageSharp`

body handle. Therefore, each game object has to implement the `ISimulationMember` interface which has a property that returns the list of all body handles associated with the given game object. More details on coupling the physics engine with the rest of the game can be found in subsection 4.2.3

The logical layer is the soul of each object. NPCs move in a way given by a simple algorithm; the player can inflict damage to NPCs by shooting projectiles, and so on. Game objects can react to collisions with other objects by registering for contact callbacks, i.e. implementing the `IContactEventListener` interface. They can also move (either on their own, like NPCs, or due to the player's input) by registering input callbacks, i.e. implementing the `IInputSubscriber` interface (see subsection 4.2.7 and subsection 4.2.6). The logic related to game objects of a given kind is usually contained in a respective Controller class, described in more detail in subsection 4.2.5.

All game objects are members of a scene (see subsection 4.2.4).

In the remainder of this section, we will cover some of the most important components related to managing game objects.

4.2.1. `Animator` class

The `Animator` class is responsible for animating the models loaded by `ModelLoader`. It takes animation keyframes and interpolates them using quaternion slerp and vector lerp which are provided by AssimpNet. It then uses the interpolated keyframes to calculate the transformation matrices for each bone in the model which are later passed to the shader. The logic is described in more detail in section 3.3.

4.2.2. `ModelLoader` class

The model loader is responsible for loading models from the file system. It loads COLLADA files using AssimpNet to convert them to model class objects. Each vertex in the mesh of the model must be dependent on at most four bones. It creates VAOs[7] for each mesh of the model, stores them in the Model class object, loads a PNG file, and stores it in a `Texture` class object.

4.2.3. `Physics` project

The Physics component is responsible for collision detection, ray casting, and physical modeling. The main purpose of this component is to expose the various functionalities of the BepuPhysics2 library to the rest of the system. The component also makes extensive use of the code

[7] VAO stands for *vertex array object* (an OpenGL Object).

from the BepuPhysics2 Demos project.

Each game object is represented by a *body* (each body has an associated *body handle*) in the physics engine. This one-to-one mapping is stored in the `Scene` class in an object of `SimulationMembers` type. `SimulationMembers` allows for adding and removing simulation members as well as accessing handles to bodies in the physics engine associated with a given game object. Game objects are represented by either simple or composite shapes in the physical simulation. Simple shapes used in the game include cylinders, capsules, and boxes. Composite shapes arise by composing simple shapes.

To facilitate debugging, the Physics component also provides a way to extract all shapes from the simulation (and decompose composite shapes) which then can be shown in debug mode, see Figure 4.1. When shown, the bounding shapes can be either green or red depending on whether the corresponding bodies are in the active or inactive state in the physics engine. In Figure 4.1 it can be seen that the player has been hit by a projectile which caused the player's body to become active (or *awakend* using the BepuPhysics parlance) as indicated by the green bounding shape. The projectile that hit the player can also be seen to be active. On the other hand, the projectiles that fell to the ground and had been still for some time became inactive as marked by the red bounding shapes.

Figure 4.1: Game objects enclosed in bounding shapes for collision detection

The body that represents the terrain was created as a separate shape from the terrain mesh triangle-by-triangle.

Game objects can listen and respond to collisions by registering their contact callbacks via the API in `SimulationManager`. Such objects implement the `IContactEventListener` interface whose implementations define the contact callbacks. The only information about the collision is

which two bodies collided and where.

Some game objects can cast rays. An example of such an object is the player who uses a ray to define a place where terrain modification should take place. Each ray in the physics engine has an ID number, direction, etc. An object that wishes to cast rays has to therefore implement an interface (`IRayCaster`) that exposes all the necessary information.

4.2.4. `Scene` class

The `Scene` class is responsible for storing all the objects in the game the player can interact with. It is also responsible for releasing the objects' resources. Some of the objects stored in the `Scene`'s instance are the following: car, chunks, player, camera, etc.

4.2.5. `*Controller` classes

There are nine controller classes in the project: `PlayerController`, `BotsController`, `ChunksController`, `ProjectilesController`, `VehiclesController`, `LightSourcesController`, `HudController`, `BoundingShapesController`, and `SkyboxController`.

All the controller classes serve the same purpose. They are responsible for rendering objects and dealing with object callbacks. For example, `ProjectilesController` renders all existing projectiles and also removes dead projectiles from the scene. Each projectile has an `IsAlive` property which is read every time a render frame callback is called by the controller to check if the projectile is still alive.

4.2.6. `*Transporter` classes

The Transporter component is vital for the spherical geometry mode of the game. It is responsible for transporting game objects between spheres. All transporters implement the `ITransporter` interface.

Each game object has a property `CurrentSphereId` that attains one of two values: 0 or 1 depending on which sphere the object is currently in (this property is a part of `ISimulationMember` interface). Whenever an object moves, the Transporter determines whether it should be transported to the opposing sphere by calculating the object's distance from its current sphere's center.

The transportation process changes the object's position and velocity. In the case of objects modeled as compound bodies, it is necessary to alter the positions and velocities of each of the components. When a camera is attached to an object that gets transported, it also has to be updated accordingly. More specifically, the Transporter transforms the camera's front vector and

updates the information about the camera's current sphere.

The functionalities mentioned above are only relevant in the spherical geometry mode. To make the system design consistent across all modes, there is also a `NullTransporter` implementation of the `ITransporter` interface which is used in hyperbolic and Euclidean geometry modes. This is a dummy class with methods that don't do anything.

4.2.7. `Context` class

The Context component is responsible for input handling and performing actions on each frame. There are two sources of user input considered in the video game: keyboard and mouse. Keys and mouse buttons can be in one of three states: *pressed*, *down*, and *up*. Additionally, the mouse can also generate events when it's moved.

The `Context` class stores mappings between different types of input and actions that should be triggered by the given input. Changes in the input state are tracked by OpenTK and the `Context` activates relevant actions as a response. It is a convention that every class that registers new actions in the `Context` implements the `IInputSubscriber` interface. Also, it is worth noting that, for better performance, the Context will only trigger actions associated with the input that has itself been previously registered.

4.3. Procedural world generation

The terrain was designed to be randomly generated so that the player can have a new, unique map every time they play. Additionally, in the case of Euclidean and hyperbolic geometries, the terrain is infinite in the sense that it is generated on the fly as the player moves around the scene. Another important part of the design was making sure that the player could edit the terrain in any way they wanted. To fulfill these requirements, the terrain was split into smaller units, called *chunks*, which are created and/or modified individually.

In what follows, we will provide more details on both terrain generation and modification.

4.3.1. Scalar Field

The first step in the terrain generation is to generate a scalar field which is a function that assigns a value to each point in 3-dimensional space. What is important is that this function always returns the same value for the same point. Another important property is that the function should return similar values for points located close to each other. Our function returns values for points that have integer coordinates.

Having these properties in mind we decided to use the Perlin noise function. Perlin noise first introduced by Ken Perlin in 1983 [11] is often used in computer graphics and in particular in procedural terrain generation. It is a pseudo-random function that returns values for any point in 3D space. However, unlike most random functions, it returns similar values for similar points. This makes it ideal for this game.

The Perlin noise function is used to generate a value for each point in the scalar field. This value is then modified based on five parameters: number of octaves, initial frequency, frequency multiplier, initial amplitude, and amplitude multiplier. How these parameters affect the terrain can be seen in Figure 4.2. The values of these parameters depend on the *seed* of the world (there are five sets of options corresponding to five different "terrain styles" or *biomes*). Each point is also assigned a *type* based on its position that is later used to determine the color of the mesh vertex.

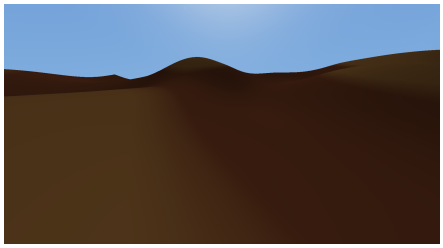
4.3.2. Chunks

As mentioned before one of the most important things for the terrain was a way to edit it. Editing the whole terrain at once would be very slow and not very efficient. Thus the terrain is split into chunks – cubically shaped, distinct sections of the world. Each chunk is a separate object and can be edited independently. This solution is much more efficient but it also causes some problems.

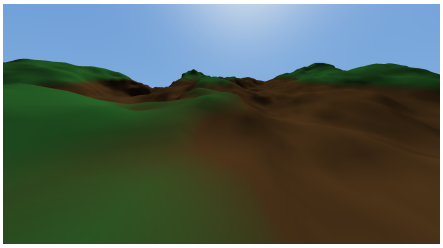
One problem is that the terrain is not continuous. Every time we edit a chunk we need to make sure that the edges of the chunk are behaving in the same way as the edges of the neighboring chunks. This is done by making sure that when a function that updates one chunk is called it is also called with the same parameters for other affected chunks. Without this, the terrain would have holes in it between chunks which is shown in a screenshot from an early version of the game in Figure 4.3.

Another problem is that the algorithm we used for generating the terrain, described in subsection 4.3.3, calculates normal vectors based on the values of the scalar field around the point at which the normal is calculated. This means that the normal vectors at the edges of the chunks have to be calculated differently. This is a common problem with the algorithm and it is visualized in Figure 4.4. The most common solution and the one we used is extending the scalar field by one layer of points around the chunk. This means that the chunk contains information about the scalar field outside of the chunk itself. That way the normal vectors can be calculated the same way for all points in the chunk.

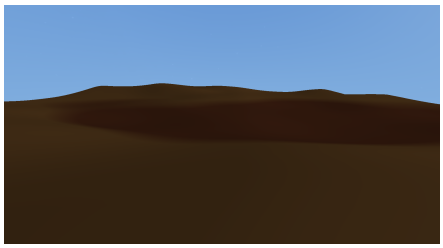
There are potentially infinitely many chunks in the world, which is why chunks are only



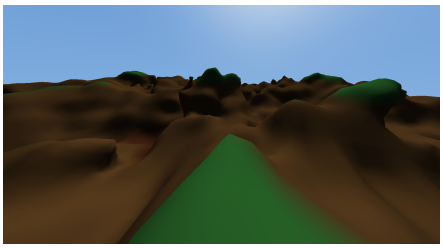
(a) Small number of octaves (1).



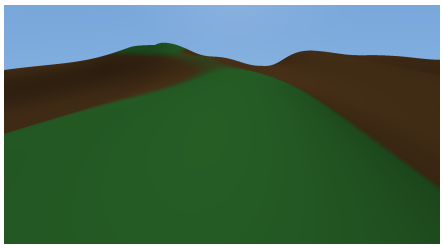
(b) Large number of octaves (5).



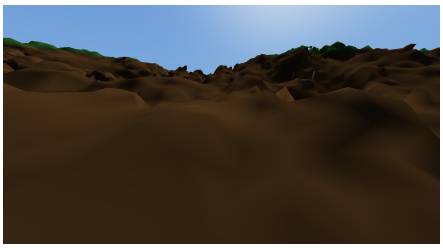
(c) Small initial frequency (1).



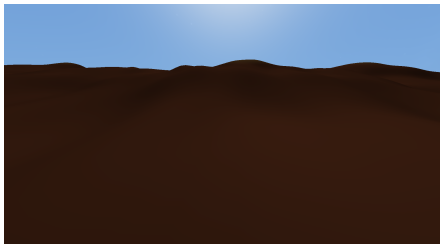
(d) Big initial frequency (0.5).



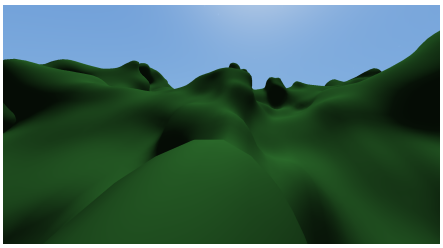
(e) Small frequency multiplier (0.5).



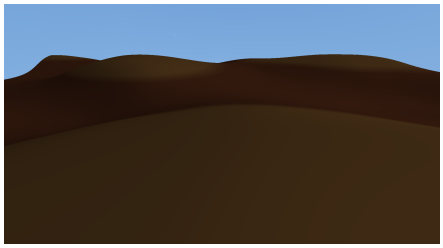
(f) Big frequency multiplier (5).



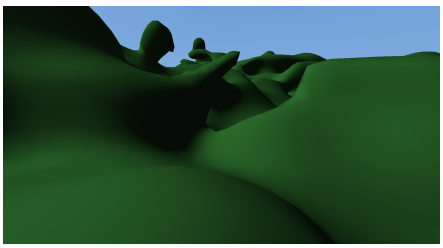
(g) Small initial amplitude (8).



(h) Big initial amplitude (32).



(i) Small amplitude multiplier (0.1).



(j) Big amplitude multiplier (1).

Figure 4.2: Scalar field parameter comparison.

loaded/created when a player is close to them. They are also unloaded when the player moves far enough away from them. When that happens they are saved to disk and removed from RAM. The same thing happens when the game is closed.

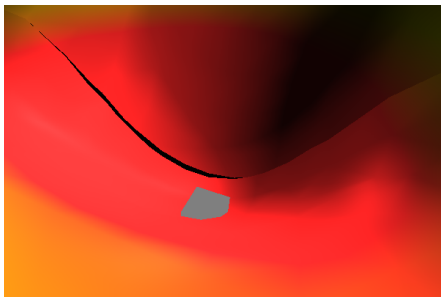


Figure 4.3: Gaps between chunks.

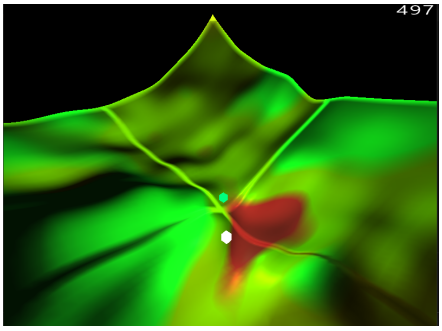


Figure 4.4: Problem with normals at chunk edges.

4.3.3. Marching Cubes

The idea of the algorithm is described in section 3.2. This section will describe the way the algorithm was implemented in our project.

To make the mesh look smoother we interpolate the position of the vertices along the edges based on the values of the scalar field at the cube's vertices. This is done by using the linear interpolation given by equation Equation 4.1.

$$P = V_1 + \frac{\text{IsoLevel} - v_1}{v_2 - v_1}(V_2 - V_1) \tag{4.1}$$

where P is the resulting position of the vertex, V_1 and V_2 are the positions of the vertices of the cube, v_1 and v_2 are the values of the scalar field at the vertices and IsoLevel is the isolevel of the mesh.

This gives us a mesh. To make the impression of a light reflecting off a smooth surface we also need to calculate the normal vectors for each vertex. The normal vectors in each point of the scalar field are calculated using Equation 4.2

$$n(x,y,z) = \begin{bmatrix} s(x+1,y,z) - s(x-1,y,z) \\ s(x,y+1,z) - s(x,y-1,z) \\ s(x,y,z+1) - s(x,y,z-1) \end{bmatrix} \tag{4.2}$$

where $s(x,y,z)$ is the value scalar field function at (x,y,z) and $n(x,y,z)$ is the normal vector at that point. These vectors are used to calculate the mesh normals using the same interpolation used for the mesh Equation 4.1.

The last part of creating the mesh is assigning colors to each vertex. Each point of the scalar field is assigned a type which is described in subsection 4.3.1 and each type has a corresponding color. The color of each vertex of the mesh is calculated by interpolating the colors of the points of the scalar field using Equation 4.1.

4.3.4. Editing terrain

Editing terrain is done by changing the values of the scalar field. When a chunk is first created the values of the scalar field are calculated for each point in the chunk and then saved. This allows for the scalar field values to be edited. When a chunk is edited the values of the scalar field are recalculated for the edited points and the points around them. The player can choose a point to build or mine at (both of these types of modification are performed using a *pickaxe*). In the case of mining, the values of the scalar field are increased by a certain amount and in the case of building, they are decreased. Modification of the scalar field also depends on the pickaxe the player uses and how long they mine for. Points located closer to the chosen point c are influenced more by the modification whereas the points lying outside of the pickaxe's range are left untouched. We propose to use a 3D Gaussian function to calculate exactly how much each point is affected to make the terrain look smooth after the modification. The function is shown in Equation 4.3.

$$f(x, y, z) = e^{-a\left((x-c_x)^2+(y-c_y)^2+(z-c_z)^2\right)} \tag{4.3}$$

In Equation 4.3 x, y and z are the coordinates of a point being edited, c_x, c_y and c_z are the coordinates of the chosen point (center of the modification) and a is a constant. The function gives the desired result of gradually decreasing the effect of modification the further a point is from c. The results of the function are added or subtracted from the values of the scalar field depending on the type of modification.

Once the values of the scalar field are updated the chunk's mesh is regenerated. This is a very time-consuming process which is why it was moved to a second thread. The communication between that thread and the main one is described in section 4.4.

4.4. Chunk management system

The majority of the game logic is handled by a single thread. The only two exceptions are the physics engine (external library) and the chunk management system. The chunk management system is split between two threads: the main thread (the thread that the rest of the application is running on) and the *chunk worker's* thread. The worker's thread is concerned with operations that are either CPU-intensive or could take a long time to execute. More specifically, the chunk worker is performing the following operations:

1. loading saved chunks from disk and generating new chunks,

2. saving chunks to disk,

3. updating chunks.

The system is built around the producer-consumer pattern, i.e. the main thread communicates with the worker's thread (and *vice versa*) using queues. It's important to note that depending on the stage of an operation either thread can be the *producer* or the *consumer*. We will now describe the workflow for each of the operations mentioned before.

4.4.1. Loading and generating chunks

On each render frame, the main thread calls the `UpdateCurrentPosition` method. This method, based on the camera's current position and the render distance, determines which chunks should be loaded into the game. If a chunk isn't already loaded or isn't queued for loading, the position of the chunk is enqueued into the `chunksToLoad` queue.

The worker thread in the `LoadChunks` function dequeues the positions of the chunks to be loaded from the disk. If a chunk is not saved on the disk, it is generated. The loaded/generated chunk is then enqueued into the `loaded` queue.

The main thread through the `ResolveLoaded` function dequeues a chunk and performs the following actions:

1. creates a vertex array object for the chunk's mesh,

2. creates a collision surface for the chunk,

3. adds the chunk to the list of scene's chunks for rendering.

It's worth noting that the operations in `ResolveLoaded` function have to be performed on the main thread because they interact with OpenGL and the physics engine APIs.

4.4.2. Saving chunks

Saving chunks is handled by a process similar to the one discussed in subsection 4.4.1. On each frame, the main thread in the `DeleteChunks` function determines which chunks are too far from the player and thus should be removed from the game and saved to disk. Positions of these chunks are enqueued into the `chunksToSave` queue, and their resources are freed. More precisely, the removal takes place only if the number of chunks in the game exceeds a certain limit. This is so that chunks are not deleted and loaded again constantly when the player moves back and forth between chunks.

The worker thread dequeues chunks' positions from the `chunksToSave` queue and saves the scalar field associated with a given chunk on the disk.

4.4.3. Updating chunks

Terrain modification consists of two main steps:

1. the scalar field for the modified chunk has to be altered, which involves iterating over a 3-dimensional array of numbers and modifying the values stored in that array using some function,
2. a new mesh has to be generated based on the new values of the scalar field.

Since the modifications happen 60 times a second and the number of operations they require is rather big (of the order of the chunk size, i.e. 16^3) it's unfeasible to do them on the main thread without severe lags. Thus most of the work related to terrain modification is delegated to the worker thread.

The workflow for terrain modification can be described as follows. The main thread in the `Pickaxe.ModifyTerrain` method determines which chunks are going to be affected by the terrain modification, and adds them to a buffer `buffer`. Once all the chunks are in `buffer`, we set the `IsProcessingBatch` flag to `true` (while set to true, no further terrain modifications will be registered) and enqueue each of them into the `modificationsToPerform` queue along with some additional information (passed in the form of an instance of `ModificationArgs` struct) necessary to perform the modification. A very important piece of information is the `batchSize` which is the size of `buffer` (its importance will become apparent later).

The worker thread dequeues chunks from the `modificationsToPerform` queue. It modifies the scalar field using the information passed in `ModificationArgs` and generates a new mesh based on the scalar field. Once new vertices for the mesh are calculated, it enqueues the chunk together with `batchSize` (retrieved from `modificationArgs`) into the `updatedChunks` queue.

In the `ResolveUpdated` function the main thread dequeues the `(chunk, batchSize)` pair from the `updatedChunks` queue and adds `chunk` to the `currentBatch` list. Once the number of chunks in `currentBatch` is equal to `batchSize` of the dequeued chunk, it means that the main thread has "received back" all of the chunks from a single modification call. The main thread then updates the GPU buffers with the new vertices of the chunk's mesh and updates the shape of the collision surface in the physics engine for each chunk from `currentBatch`. Once the whole `currentBatch` is updated, the `IsProcessingBatch` flag is set back to false.

The whole process is depicted in Figure 4.5.

The reason for processing chunks in batches rather than individually is simple. If we modify chunks one by one it may be the case that when the terrain is rendered, one chunk has already been modified, while its neighbor has not, resulting in a visible gap between the two. This

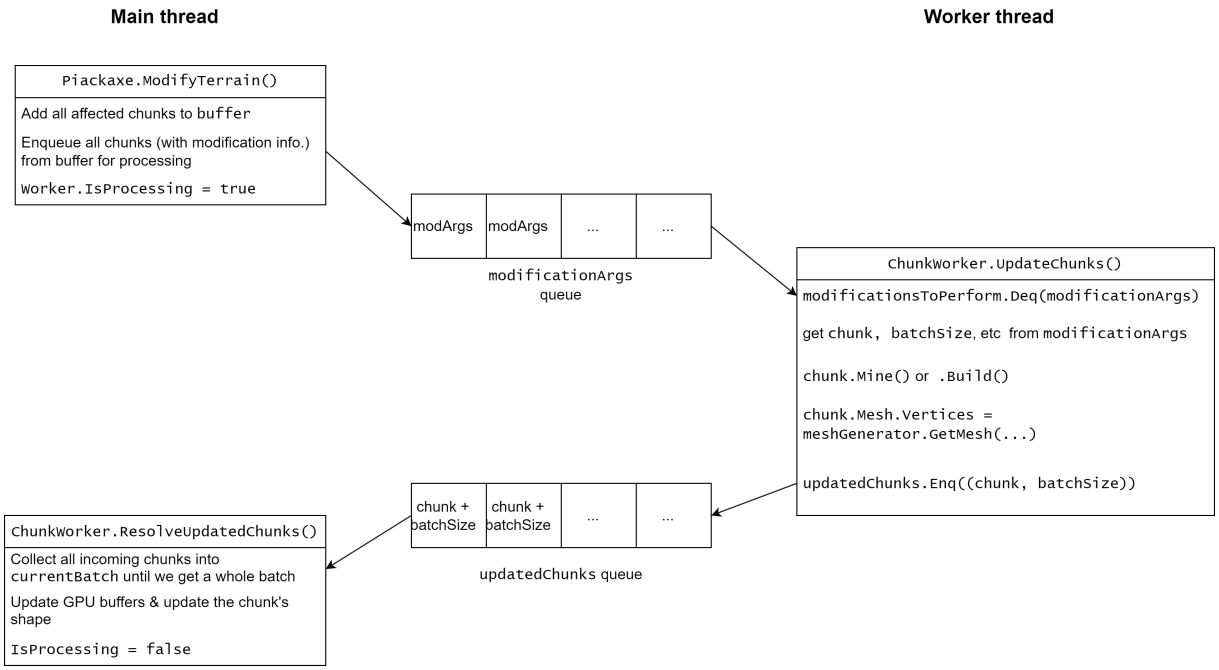


Figure 4.5: Terrain modification in the chunk management system

problem can be seen in Figure 4.6 which comes from an early stage of the game's development.

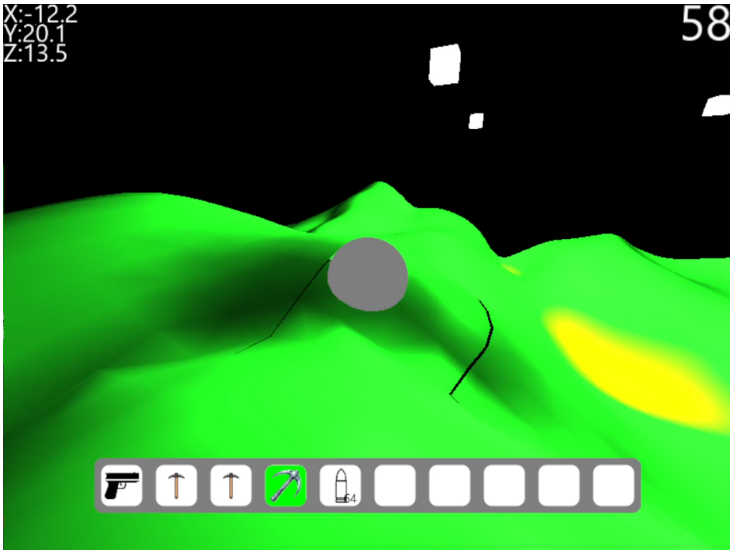


Figure 4.6: Gaps between chunks appearing during terrain modification

One drawback of this approach is that the main thread doesn't register new chunks for modification while the whole previously enqueued batch hasn't been processed. This could cause the modification rate to be irregular should the main thread "drop" a lot of modifications. To deal with this problem, the modification depends on the time elapsed between two modifications.

4.5. Rendering

Every game object exposes a `Render()` method. The `Render` methods' signatures differ slightly across different game objects, but they usually take camera position, shader and space curvature as arguments. At the time of writing, there are four shaders available for 3D rendering:

1. light source shader,
2. model shader,
3. object shader,
4. skybox shader.

Model shader is used for rendering animated models, whereas light source and object shaders are used for rendering "static" bodies. The light source shader is extremely basic: it doesn't take into account other light sources and colors the body uniformly. The `Render` method typically interacts directly with the OpenGL interface, i.e. sets up the uniforms, binds VAOs and makes a draw call. In the case of 2D rendering, `HudShader` class is used. The skybox shader is used for rendering the *skybox* i.e. a background used for the scene.

4.6. User interface

OpenGL is a low-level graphics API. It offers a set of functions to draw points, lines, and triangles. It does not offer any functions to draw circles, ellipses, or other shapes. That includes drawing text. Because of that adding the heads-up display (HUD) and the Menu to the game is not as simple as we would like it to be. In this chapter, we describe how we draw 2D elements in our application.

4.6.1. Textures

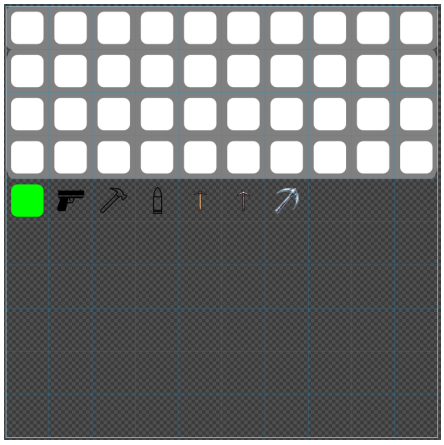
OpenGL does offer a way to draw images. To draw an image you have to create a texture. This texture is then passed to a shader. You also have to create a VAO which contains vertices for a rectangle. Each vertex has a position and a texture coordinate. These texture coordinates are used to sample the texture using built-in shader functions.

Each image we want to draw can be represented by a texture. For each image we want to draw, we can create a texture and pass it to the shader. While this approach works, it is not very efficient. Passing textures to the shader is a slow operation. Because of that, we want to use as

few textures as possible. We can do that by combining multiple images into one. We create *sprite sheets* which contain multiple images. Each image in a sprite sheet (a *sprite*) has a position and a size. We can use this information to calculate the texture coordinates for each sprite. We pass this information to the shader and use it to sample the correct part of the texture. This way, we can pass a single texture to the shader and draw multiple images with it.

In our application, we have two sprite sheets. The first one contains the images for the HUD. This includes the inventory and all the items in it. This sprite sheet is stored as a PNG file. Sprite sheet with the inventory can be seen in Figure 6.6. It also has a JSON file that contains the position and size of each sprite which can be seen in Figure 4.7b. The second sprite sheet contains the images of all letters and symbols used in the game. This one is not stored in a file. Instead, it is generated at runtime right after the program launches. It uses the SkiaSharp library to create a bitmap with all the ASCII characters. This bitmap is then converted to a texture, which we call the *symbol texture*. The position and size of each symbol are calculated using the font metrics.

These techniques are used to draw the HUD and the Menu. Both of those are described in this chapter.



(a) PNG file

```
1   {
2       "width": 10,
3       "height": 10,
4       "items": [
5           {
6           "name": "hotbar",
7           "x": 0,
8           "y": 0,
9           "width": 10,
10          "height": 1
11          },
12      ]
13  }
```

(b) JSON file

Figure 4.7: Inventory Sprite Sheet

4.6.2. Heads Up Display

The HUD encapsulates all the 2D elements that are drawn on top of the 3D scene while the game is running. This includes:

- Crosshair
- FPS counter
- Player position
- Inventory

The Crosshair is a simple cross in the middle of the screen. It is used to help the player aim. It consists of two lines and does not use any textures.

The FPS counter is a simple text that shows the current frame rate. It is drawn in the top right corner of the screen. It uses the symbol's texture.

The player position is a simple text that shows the current position of the player. It is drawn in the top left corner of the screen. It uses the symbol's texture.

The inventory is a collection of images that represent the items the player has [8]. It is drawn at the bottom of the screen. It uses both the item's texture containing the HUD items and the texture containing the alphabet. The images of the items are drawn first and then the text is drawn on top of them.

All the elements of the HUD have their positions and sizes. Each element is either placed in some position on the screen or is placed relative to some other element.

4.6.3. Menu

The menu is a lot more complicated than the HUD which is described in subsection 4.6.2. Because of that, we decided to create a framework for creating menus. This framework was heavily inspired by Flutter [7]. It uses the same concepts and terminology. The overall idea is that everything is a widget. A widget is a class that has a `Render` method as well as a `GetSize` method. The `Render` method takes a context that includes information about the position and size on the screen that the widget can render to. Some widgets also have children so the overall structure of the menu is a tree of widgets.

[8] Icons for the items were downloaded from `https://www.flaticon.com/`; Idea icons created by Good Ware - Flaticon (`https://www.flaticon.com/free-icons/idea`), Pistol icons created by Vector Stall - Flaticon (`https://www.flaticon.com/free-icons/pistol`), Bullet icons created by Smashicons - Flaticon (`https://www.flaticon.com/free-icons/bullet`)

An example of a widget is shown in Figure 4.8b. This widget renders the main menu of the game. The rendering logic of this widget can be seen in Figure 4.8a. The idea is as follows. The root widget calls the render method of its child which is the `Background` widget. The `Background` widget renders the background color. Then the `Background` widget calls the render method of its child which is the `Column` widget. This widget renders its children in a column but to do that it first needs to know the size of each of its children. Based on that information it will call a render method of each child with the appropriate context. Each child asks its children for their size recursively until it reaches a leaf widget. The process stops at the leaf and the render method is called on the children of the column widget.

This is a simplified version of the rendering logic as each widget has multiple options and rules that change how it or its children are rendered. For example, the `Column` widget has a `alignment` property which changes how the children are aligned. The `Button` widget in the Figure 4.8a itself is a tree of widgets.

This approach to rendering the menu is very flexible and allows for a lot of customization. It improves on the method used to render the HUD described in subsection 4.6.2. This approach is not common in game development. Usually, menus are created by putting elements on the screen at specific positions. In this part, we believe that our approach is better than the traditional one and improves on approaches used in for example Godot [8].

(a) Widget rendering logic.

(b) Main menu.

Figure 4.8: Example widget.

5. Testing

Testing a video game is not an easy task and it differs substantially from testing other software. In most software the most important thing is the functionality, however, in games, the visuals are just as important. That means that testing if for example the player model was rendered is not enough; it also has to be tested if it was rendered correctly. That is made even more difficult by the fact that the game uses the GPU to render the graphics. All these difficulties make testing in code very difficult, which is why apart from unit testing we focused on manual testing which is a standard approach in the industry [14].

5.1. Unit testing

Unit testing was used to test specific methods and classes. These tests check if the methods behave as expected. However, even using mocking frameworks some things, in particular things connected to the graphics, cannot be unit tested. Because of that the most important part of testing was manual testing described in section 5.2.

5.2. Manual testing

Manual testing is responsible for testing the game as a whole. It tests the results and not particular methods or parts of the code. Manual test scenarios were created that describe in detail what should happen in the game after some actions are performed. These tests cover the entirety of the application and can be viewed at `https://github.com/Non-Euclidean-World/HyperTesting`. The tests are written as Markdown files that have references to image and video files that show the expected behavior of the game. The test cases are divided into four parts:

1. Title – The name of the test case.
2. Description – What the test case is supposed to verify.
3. Prerequisites – What has to be done before the test case can be performed.

4. Steps – A table that describes the actions that have to be performed and their expected results.

Below is an example of a test case. It is important to note that the test case has been modified to fit the paper format by changing links to the images into references.

Flashlight works

Description

Test case for checking if the player can use the flashlight. The test case should work for all geometries.

Prerequisites

The game is running.

The camera is in 1st person mode.

Steps

Step	Action	Expected Result
1	Press `Y`	The flashlight should turn on, see here. (Figure 5.1)
2	Press `Y` again	The flashlight should turn off

Figure 5.1: The flashlight is on.

6. User Manual

This section describes how to launch and later use the application. The user controls were made with industry standards in mind, so the user should not have any problems with using them. However, if the user does not have much experience with video games, this section will help them get started as it provides a detailed start-to-finish description of how to play the game.

6.1. Launching the game

The application starts in the main menu shown in Figure 6.1. From here a new game can be started, a previous save can be loaded, or a save can be deleted. The user can also view controls, and exit the game. To start a game for the first time, the user has to click on the "New Game" button which will show the "New Game Screen".

When the game is running, the user can press the Esc key to view the main menu or hide it. While the game is running the main menu will also show a "Resume" button which will resume the game and hide the menu.

Figure 6.1: Main menu

6.2. New Game Screen

The game can be started from the New Game screen shown in Figure 6.2. To start a game, the user has to click on the "Input Game Name" input text box. This will allow them to enter a name for the game. If there is no game save with the same name, the text box will light up green. If there is already a game save with the same name, the text box will light up red. If the box is green, the user can press one of the buttons to start a new game in the chosen geometry.

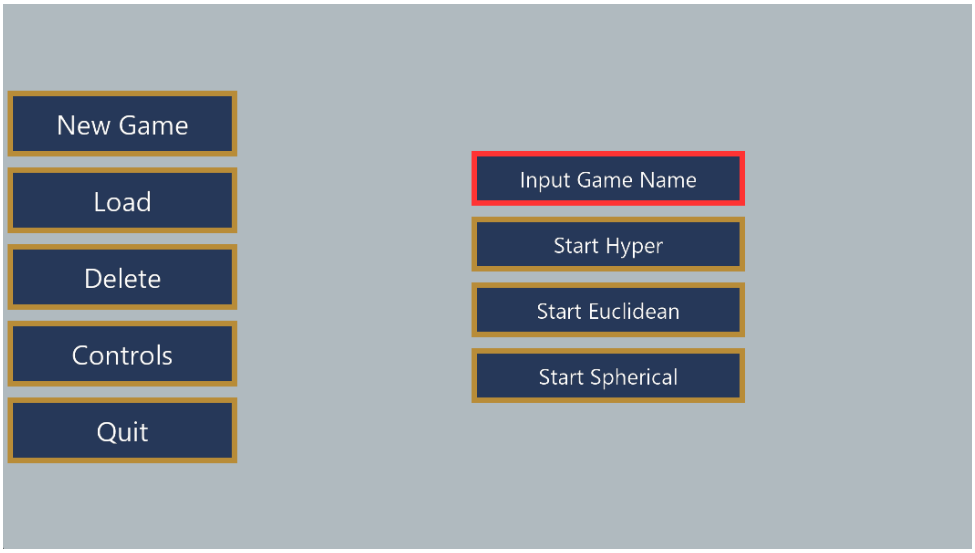


Figure 6.2: New Game screen

6.3. Load Game Screen

The game can be loaded from the Load Game screen shown in Figure 6.3. This screen displays the nine most recent saves. The user can load a save by clicking a tile corresponding to the save. To load a more recent save the user has to first delete more recent saves using the "Delete Game" described in section 6.4

6.4. Delete Save Screen

The "Delete Game" screen shown in Figure 6.4 allows you to delete your saves. It will display the nine most recent saves. To delete a save the user has to click a tile corresponding to the save. If a game is running, the user will be prevented from deleting its associated save.

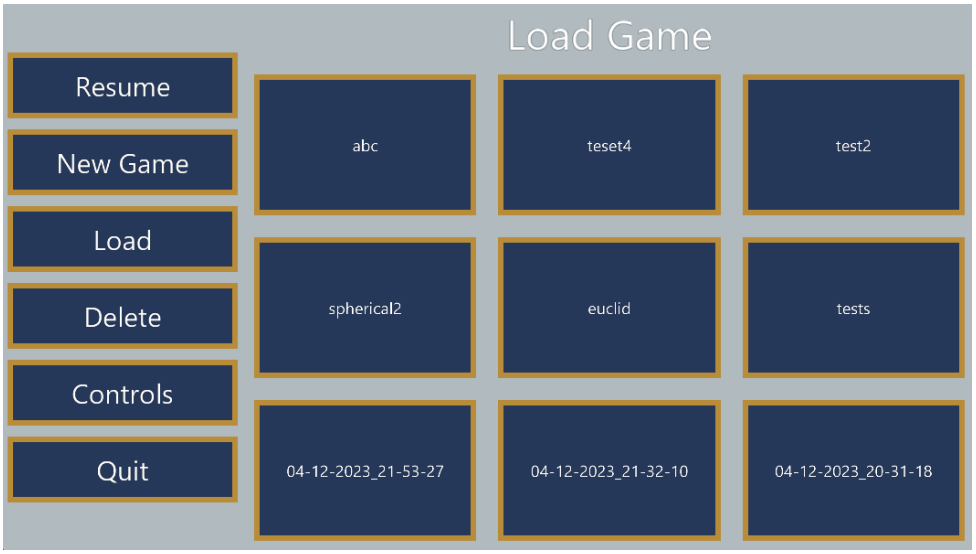


Figure 6.3: Load Game screen

Figure 6.4: Delete Save screen

6.5. Controls

The controls for the game are presented in Table 6.1. If the user forgets them, they can press the "Controls" button in the menu at any time to see them in the game, as shown in Figure 6.5.

6.6. Items

Items are a big part of the game. Different items in the game have different uses. Description of all the in-game items can be found in Table 6.2. To use the items the player has to first select

Key	Function
W	Move forward.
S	Move back.
A	Move left.
D	Move right.
⇧	Sprint.
Space	Jump.
`Left Mouse`	Use item.
`Right Mouse`	Use item's second ability.
Esc	Show/Hide menu.
⇥	Switch camera between 1st and 3rd person.
`Scroll`	Change curvature (works only in hyperbolic geometry).
0 through 9	Select item.
C	Enter car.
F	Flip car (works only outside the car).
L	Leave car.
Y	Toggle flashlight/Toggle car reflectors (when inside the car).
F1	Toggle cinematic mode.
F3	Toggle showing hitboxes.

Table 6.1: Keyboard Key Functions for Game Controls

the desired item using number keys 0 through 9. The selected item will be highlighted in the inventory as shown in Figure 6.6; in this example, the selected item is the gun. The player can use the item by pressing the left mouse button (`LMB`) or the right mouse button (`RMB`). The effect of the item depends on the item itself and the mouse button used.

6.7. Gameplay

After the game is started, the user can finally start playing. At the start, the screen will look like in Figure 6.7. The user can see their position in the top left corner, the current frames per second in the top right corner, and the inventory in the bottom of the screen. In the middle

Figure 6.5: Controls screen

Item	`LMB` Effect	`RMB` Effect
	No effect	No effect
	Fires a bullet if there is one in the inventory. Removes a bullet from the inventory.	No effect
	Mines slowly.	Builds slowly.
	Mines.	Builds.
	Mines quickly.	Builds quickly.

Table 6.2: Table of in-game items

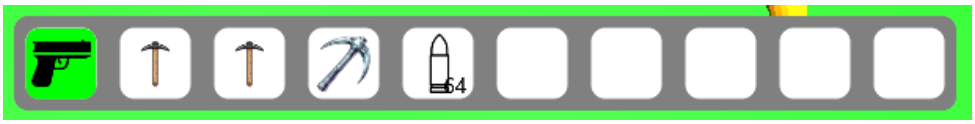


Figure 6.6: Inventory

of the screen, there is a crosshair that shows where the user is aiming. The user can also see their character model. By using the controls described in section 6.5, the user can start moving around and exploring the world.

Figure 6.7: Gameplay screenshot

7. Results

We find the results of our work satisfactory. All the functional requirements are satisfied and the application runs as we envisioned it. These sections will describe the results focusing on the performance, gameplay, and incorporation of non-euclidean geometry.

7.1. Performance

The performance and resource utilization of the game has been tested using Visual Studio's performance profiler[1]. The analysis was focused on three aspects:

1. File I/O,
2. CPU usage,
3. memory usage.

The data was collected on a machine running on x64-based Windows 11 with Intel(R) Core(TM) i7-9750H CPU, 32 GB of RAM, and NVIDIA GeForce GTX 1650 GPU. During the first minute of the data collection, the player was running constantly, approximately in one direction, while modifying the terrain at the same time. Starting at the 1-minute mark, the player turned around and started running in the opposite direction, still modifying the terrain, and also shooting at the bots from time to time. The results of the profiling are shown in Figure 7.1.

The "File Reads" graph shows the amount of data read in MB. The initial spike in the file reads corresponds to the game reading the shader files and PNG files with textures. It can be seen that starting at the 1-minute mark, there are frequent file reads. This is because the chunks that were generated (and subsequently removed from the game and saved) during the initial run in one direction are now being revisited by the player and read from the disk.

The "Process Memory" graph shows that the memory usage remains approximately constant at around 350 MB.

The "CPU" graph shows that the CPU utilization peaks at around 10%.

[1] `https://learn.microsoft.com/en-us/visualstudio/profiling/?view=vs-2022`

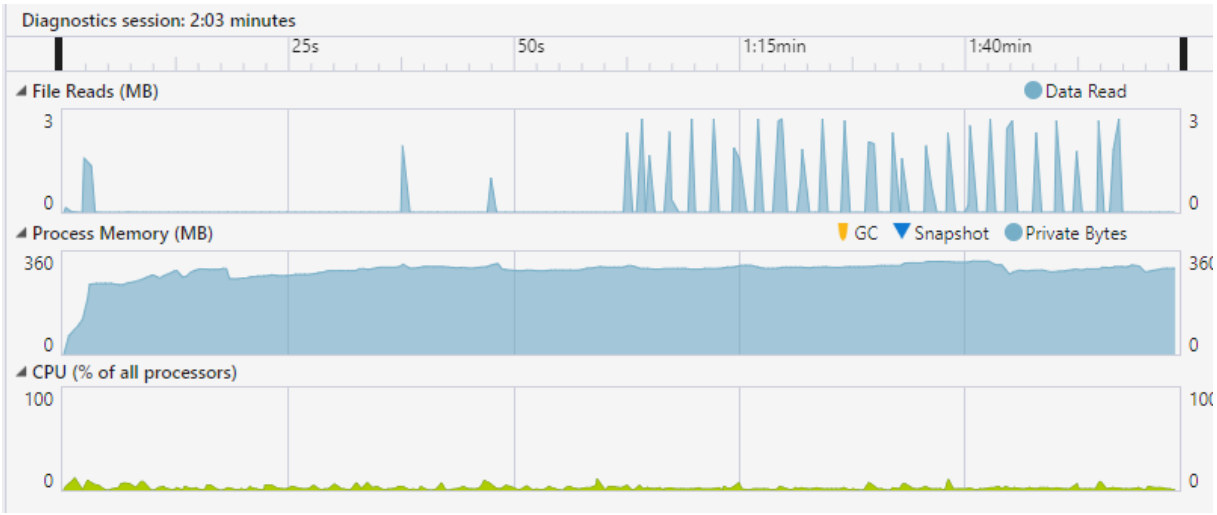


Figure 7.1: Resource utilization of the game

The total amount of disk space required to store the game saves for this session was 100 MB.

7.2. Gameplay

Hyper was designed as an *open world* game. In a game of this type, the player is not constrained to achieving a specific goal and has a large degree of freedom to explore, interact with, or modify the game environment [10]. In this section, we'll show the various aspects of this concept in our game.

7.2.1. Terrain editing

The player has three "terrain modifiers", represented with pickaxe symbols, at their disposal. Using the terrain modifier, the player can edit the game's landscape by building new structures or digging in the ground. As an example of the "creation capabilities", we show in Figure 7.2 how the letters making up the game's name could be created inside the game.

As mentioned before the terrain modifiers can also be used for carving in the terrain. To illustrate that, in Figure 7.3 we show a tunnel that was dug through a hill.

7.2.2. Exploring the game world

The game starts in one of the various "landscapes" such as a desert, a forest, etc. each characterized by different terrain generation parameters and colors.

To make exploring the game world easier, the player can get into a car that moves considerably faster than the player. Figure 7.4 shows the car driving in hyperbolic space. Even though the

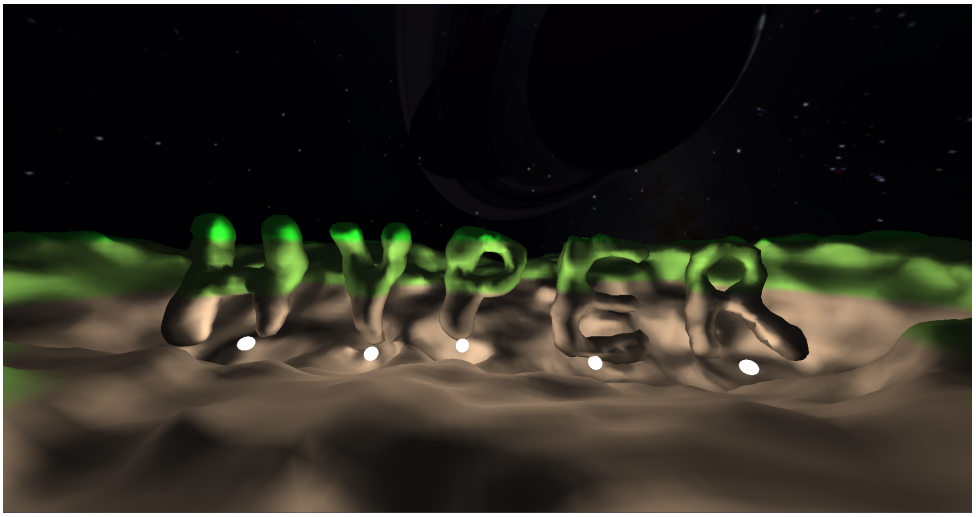


Figure 7.2: The letters of the word "Hyper" created in the game

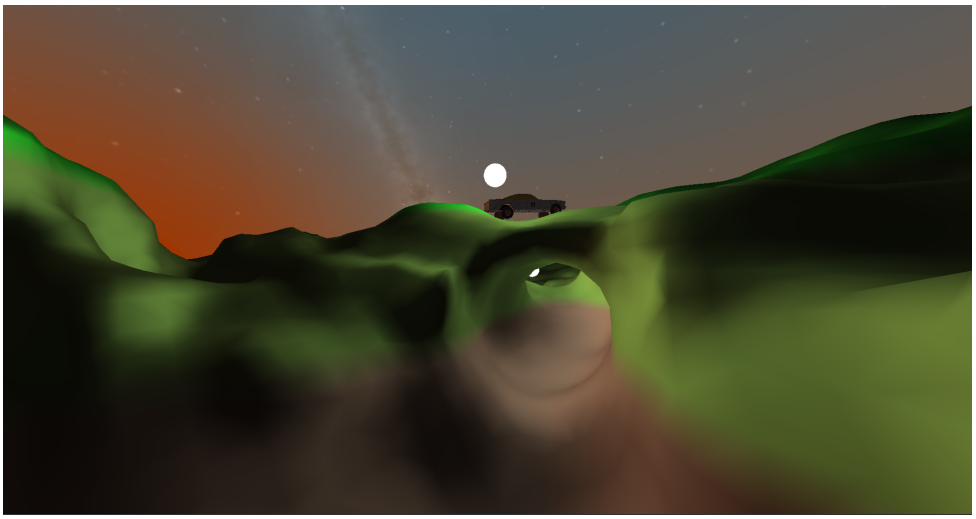


Figure 7.3: Tunnel dug under a hill

car is much faster than the player it can roll over when traversing a particularly bumpy terrain. For this reason, we included an option to flip the car back on its wheels when that happens.

7.2.3. Interacting with the world's inhabitants

To make the world more interactive we decided to populate it with NPCs also called bots. Bots can be either hostile toward the player or neutral. Hostile bots shoot projectiles and walk toward the player once they get into the player's proximity. A group of bots shooting projectiles at the player in spherical space is shown in Figure 7.5. The neutral bots are unbothered by the player's presence and just walk around. This can change, however. Once the player shoots at a neutral bot, it'll become hostile.

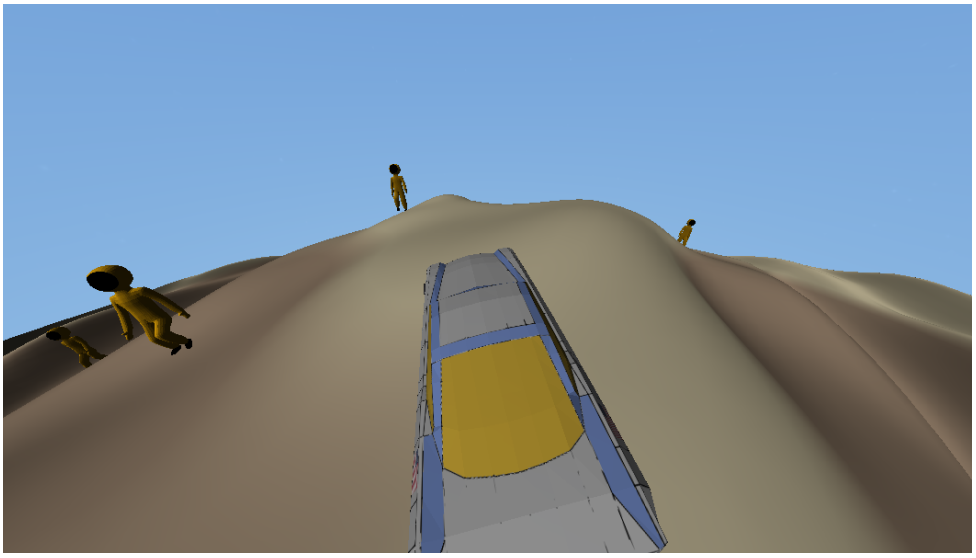


Figure 7.4: Riding a car in hyperbolic space

Figure 7.5: Fighting with bots

7.2.4. Physics

BepuPhysics2 proved to be an excellent choice for a video game physics engine. Our tests, such as the one depicted in Figure 7.6 show that it's able to handle large workloads. In this particular test, we spawned 100 bots, each firing projectiles at the player. Despite this workload on the physics engine, the application still managed to work without lags.

Figure 7.6: BepuPhysics library taken to the extreme

7.3. Depicting non-Euclidean spaces

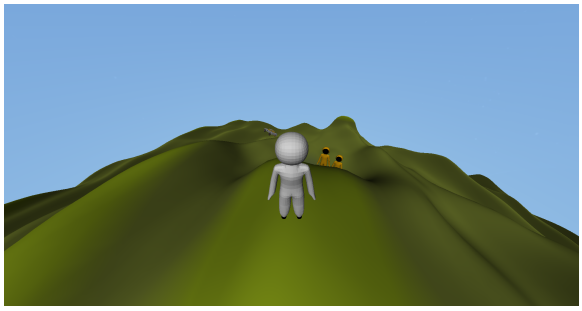
One of the main features of the game is the ability to explore non-Euclidean spaces. To assess our depictions of non-Euclidean spaces, we decided to compare them with the ones from the game *Hyperbolica*[2] by *CodeParade*. It should be noted that the approach used in our game, although relatively simple makes it impossible to capture some of the interesting properties of non-Euclidean geometry, like tiling the plane with right-angled regular pentagons in hyperbolic space.

7.3.1. Hyperbolic space

From the visual standpoint, the hyperbolic space can be identified by the fact that the otherwise flat terrain appears "curved downward". This may create an illusion that the terrain is wrapped around a giant sphere. However, by exploring the hyperbolic space, we can quickly notice that it is infinite, just like the Euclidean space. Figure 7.7 shows the comparison of the depictions of hyperbolic space between our game and *Hyperbolica*.

As mentioned in Equation 3.1.3 in our approach the camera's position (as passed to the view matrix) is fixed. Changing the point where it is located allows us to change the curvature of the terrain. Figure 7.8 shows the scene in hyperbolic space where the camera is positioned close to the origin (cf. Figure 7.8a) resulting in small curvature as compared to the scene when the camera is moved farther from the origin (cf. Figure 7.8b).

[2] `https://codeparade.itch.io/hyperbolica`



(a) *Hyper*

(b) *Hyperbolica [3]*

Figure 7.7: Hyperbolic space

(a) Camera close to the origin

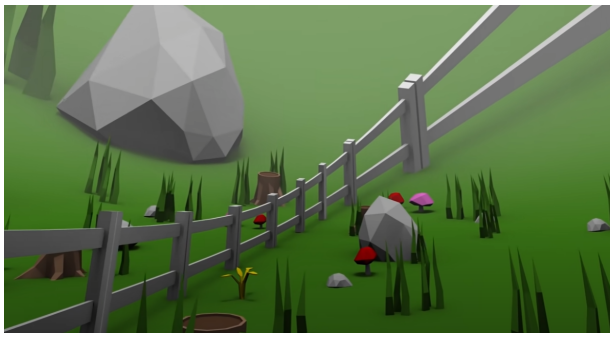
(b) Camera far from the origin

Figure 7.8: Hyperbolic space, different curvatures due to camera's position

7.3.2. Spherical space

Playing the video game in spherical space gives the impression that the terrain is wallpapered onto the inside of a giant sphere. Unlike the hyperbolic space, the spherical space is finite. Comparison in Figure 7.9 shows that our implementation provides visual effects to a certain degree similar to those in *Hyperbolica*.

(a) *Hyper*

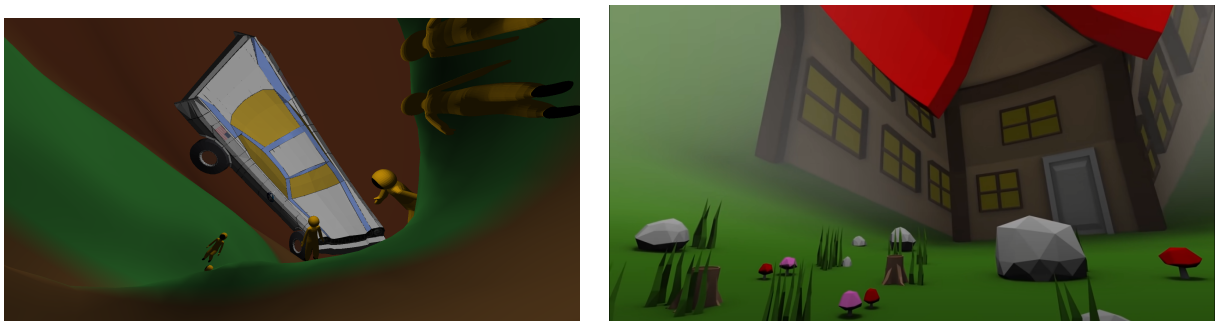


(b) *Hyperbolica [4]*

Figure 7.9: Spherical space

The effect of characters bending increasingly as the player gets further from them is another

example of an unusual visual effect in spherical space. Furthermore, unlike in Euclidean geometry, objects further from the player do not always appear smaller, sometimes they can even appear larger. This effect of "reversed perspective" can be seen in both games as shown in Figure 7.10. In Figure 7.10a, the car and the bot to the right of the car's hood are further from the camera than other objects in the scene. It's also important to note that bullets no longer fly in straight



(a) *Hyper* (b) *Hyperbolica*

Figure 7.10: Reversed perspective

lines and that their trajectory is bent instead. By playing the game it can be experienced that steering the car or moving the player's character appears unintuitive close to either of the spheres' boundaries.

8. Problems

While making the game many problems were encountered, fortunately, almost all of them were solved. This chapter is dedicated to both the ones that were solved and the ones that were not. It will focus mostly on the problems specific to this game, meaning the problems connected to non-Euclidean geometry and terrain generation will be the main focus.

8.1. Problems with hyperbolic space

As described in subsection 3.1.3, in hyperbolic geometry the world moves and the camera is stationary. This is done to limit the distortions since the distortions get bigger the further away from $(0, 0, 0)$ the object is.

To make the world appear curved in the hyperbolic geometry, the camera's position, as passed to the view matrix, has to be at some position other than $(0, 0, 0)$. If the camera is too close to the origin, the curvature of the scene would be very small, barely distinguishable from Euclidean geometry as can be seen in Figure 8.1.

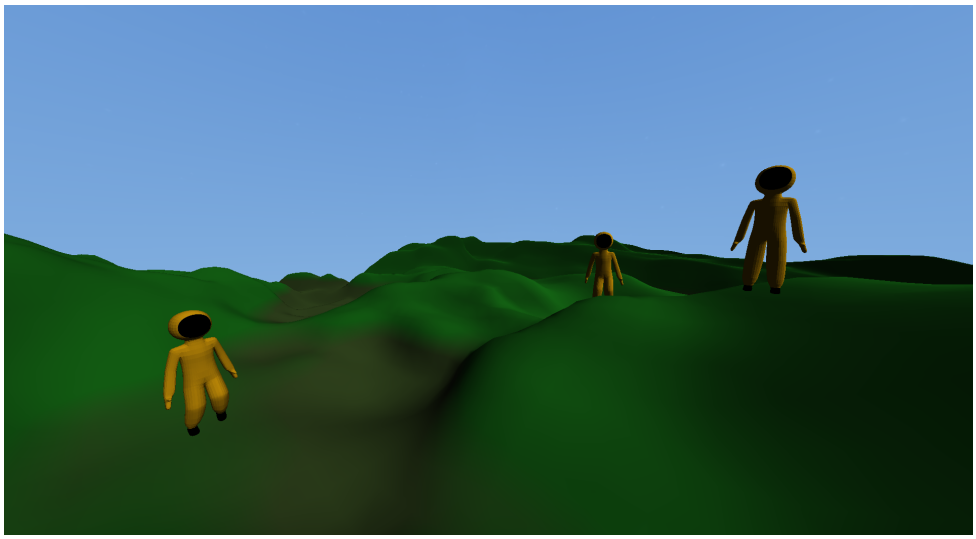


Figure 8.1: Hyperbolic geometry with the camera at the player's head.

To make the scene appear visibly curved, the camera position was changed to a point with a

relatively high value of the y coordinate. This solution, however, created a new problem, as the new camera's position made the gameplay look unnatural, as can be seen in Figure 8.2.

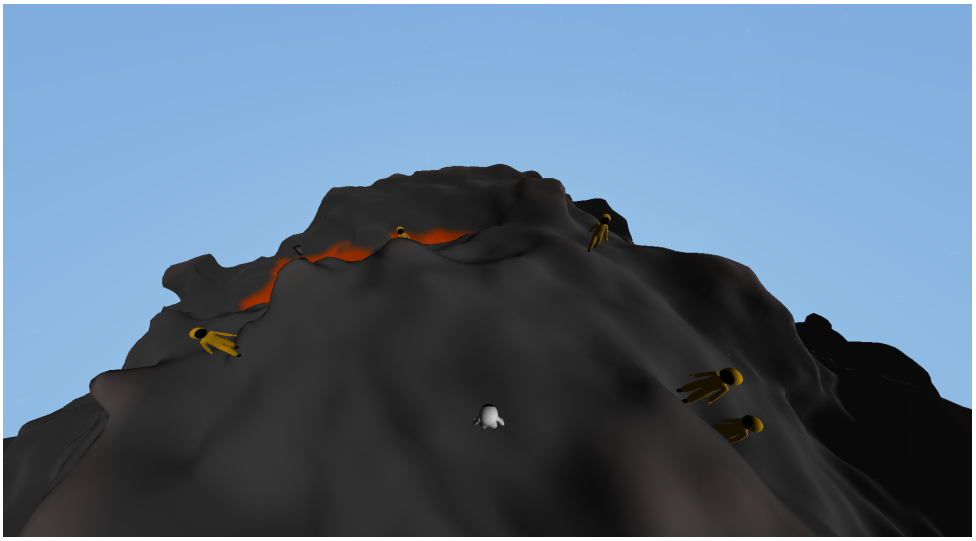


Figure 8.2: Hyperbolic camera placed high above the player.

The solution to this problem was to move the whole world up. The whole logic is as follows:

1. The player inputs a key combination that moves him.
2. The world is moved to simulate the effect of the player moving.
3. The world is moved up to adjust for the raised camera.
4. The frame is rendered.

It is worth noting that "moving the world" only involves passing an appropriate vector to the shader. The locations of the objects themselves are obviously not changed for performance reasons.

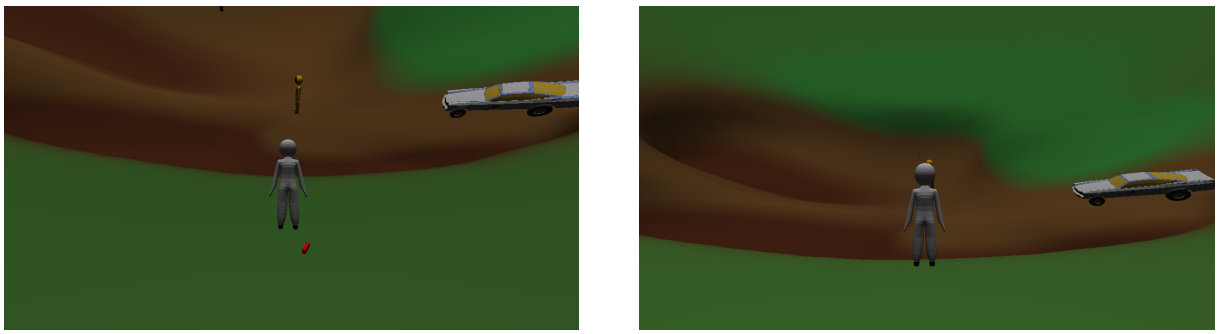
8.2. Problems with spherical space

The spherical geometry is the most problematic of the three geometries. This is because of the mechanism of dividing the world into two regions as described in detail in subsection 3.1.4.

8.2.1. Teleportation

The teleportation mechanism described in subsection 3.1.4 is far from a perfect solution. When a model crosses the boundary of one region it has to be physically moved to a completely different position and its velocity has to be altered as well. This causes a visible discontinuity in

the object's movement. The problem is even more apparent when the camera is being teleported. Positions of the objects in the scene appear to be slightly different after the teleportation which causes an unpleasant impression of a sudden "jump". This effect is depicted in Figure 8.3. Figure 8.3a shows the scene just before the player crosses the boundary of a region, while Figure 8.3b shows the scene just after the teleportation to the second region took place.



(a) Just before teleporation

(b) Just after teleportation

Figure 8.3: Camera jump resulting from teleporation

8.2.2. Terrain generation

The terrain generation mechanism has to be modified in spherical space. Firstly, we have to generate only a finite amount of terrain for each of the regions. Furthermore, each region has to have a disk-like shape with a diameter equal to π. In the implementation, we use four chunks of side length of 32 with a global scale factor of 0.05, so that each region is initially a square with a side length equal to 3.2. The vertices located at a distance greater than $\pi/2$ from the region's center are discarded.

Second of all, the scalar field used for the terrain generation has to be modified to facilitate a smooth transition between the two regions. Figure Figure 8.4 depicts the terrain with a discontinuity on the region's boundary (enclosed in a red box) in an early version of the game.

Our solution was to modify the values of the scalar field for each region such that they combine the original scalar field S (as generated by the noise function) with a *boundary value* b (the same for both regions). The resultant scalar field, S_{avg} is defined as

$$S_{\text{avg}}(x, y, z) = \text{StepUp}(d, R) \cdot b(y) + \text{StepDown}(d, R) \cdot S(x, y, z),$$

where d is the distance of the point (x, y, z) from the given region's center, R is the region's radius (around $\pi/2$), and the StepDown amd StepUp functions are defined as follows:

$$\text{StepUp}(d, R) = \frac{1}{2}(\tanh(a(d - kR)) + 1),$$

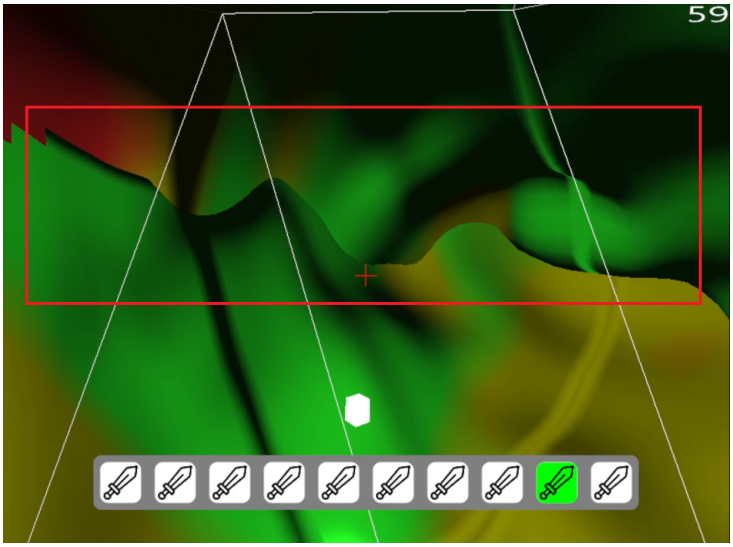
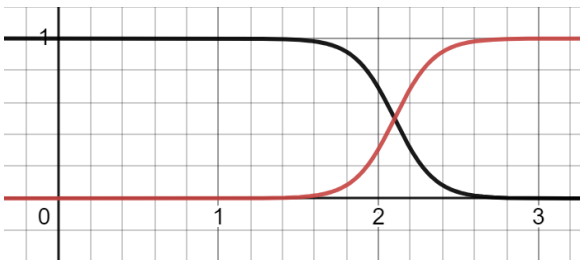


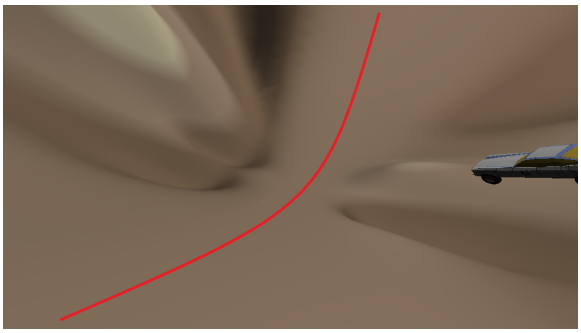
Figure 8.4: Discontinuity between two regions in spherical space

$$\text{StepDown}(d, R) = \frac{1}{2}(-\tanh(a(d - kR)) + 1).$$

The parameter k describes at what distance from the region's center (relative to R) the boundary value starts to dominate and the a parameter describes how aggressive the transition is. Plots of the functions can be seen in Figure 8.5a. Figure 8.5b depicts the result of applying the aforementioned functions (the red line shows the approximate location of the boundary between regions).



(a) StepUp (red) and StepDown (black) functions



(b) The effect of applying the step functions

Figure 8.5: Smooth transition between regions in spherical space

In our implementation, we use $a = 0.5$ and $k = 0.75$. The boundary value depends only on the y coordinate and is simply given by $b(y) = y$ which means that the terrain generated based on this function is flat.

The last modification to the terrain generation system has to do with the average elevation of the terrain. It turns out that points located too far from the camera are not rendered correctly[1].

[1]The problem is probably related to the projection matrix but we didn't investigate this further.

To solve this problem, the terrain is artificially elevated and mining below a certain depth is disallowed.

8.2.3. Lighting in the second semi-hypersphere

The introduction of the translation matrix 3.9 specific to the second semi-hypersphere (the one with negative w coordinates) caused problems with porting vectors that we were not able to fully solve. In particular, the point light sources and spotlights do not behave the same way as in the first semi-hypersphere.

8.3. Terrain problems

The main problem with terrain generation was performance. Generating terrain is not a particularly resource-consuming task, however, editing the terrain is. The main challenge was making the process of editing the terrain appear smooth. To achieve that the terrain has to be updated at least 30 times per second which means that multiple chunks, each one having almost 7000 vertices, have to be updated at least 30 times per second. This process was so resource-consuming that it had to be moved to a separate thread as described in section 4.4. However since the main thread is the one that communicates with the GPU and the physics engine, the main thread still had to be involved in the process. This created many problems connected with making sure that there were no race conditions and discontinuities in the terrain while maintaining performance. The whole logic is described in detail in section 4.4.

8.4. Architecture problems

Clean architecture was very difficult to achieve in this project. The main reason for this was the fact that it uses three different geometries and each of them is governed by different rules. For example, as described in subsection 3.1.3, in hyperbolic geometry the world moves and the camera is stationary. On the other hand in the case of spherical geometry, it has to be known in which of the two regions described in subsection 3.1.3 the object is currently in. These are just two examples but there were many more things that only appeared in one of the geometries. Adding if else statement for each of these would make the code very messy and difficult to read so we decided on a different approach. We used the strategy pattern to define the behaviors in each of the geometries. The strategy pattern is a behavioral design pattern

that enables the algorithm's behavior to be selected at runtime. It was originally described in the book *Design Patterns: Elements of Reusable Object-Oriented Software* [6]. To provide the appropriate strategy to the object we used the factory pattern also described in this book.

9. Improvements

Creating a video game is a process that is never truly finished. There is always some adjustment to make, some bug to fix or some new feature to add. *Minecraft*, one of the games that inspired this project, and one of the most popular games of all time, is still being updated regularly despite being released 2011. This game is no different. Many improvements could be made, some of which are discussed in this chapter. It will focus on improvements and additional features but not bugs as existing problems are discussed in chapter 8.

9.1. Terrain improvements

The modifiable, procedurally generated terrain is one of the hallmarks of the game. It can be modified, it is procedurally generated and it is infinite. Moreover, there are five different biomes which adds some variety to the game. But of course, there is always room for improvement. One such improvement would be to allow for multiple biomes in the same world. Right now the whole world is the same biome but it would be interesting to have different biomes blend into each other.

Another improvement would be to add some structures to the world. Right now the world is just hills and valleys but it could include things like trees, rocks, buildings, roads, rivers, lakes, etc. In particular, lakes and rivers would be interesting as there is no water in the game as of now. Adding water and maybe even different fluids would be a great improvement.

Terrain modification could also be improved. Right now the player cannot choose which materials to build with. It would allow for more interesting structures if the player could choose what type of material to place where. It would also make for more interesting gameplay if the player had to gather resources to build with instead of having an infinite supply of them.

The last upgrade concerning the terrain would be to minimize the amount of space the terrain takes up on disk when game saves are created. Right now each chunk takes 54 KB of memory. This adds up to 20 MB for a new world. Both during standard gameplay and in testing, a considerable amount of data may be saved on disk. Three simple solutions could reduce the

problem but due to the nature of the game, the problem cannot be removed completely. The first solution is to compress the files, which would reduce the memory by around half, however, it would slow the game down as the files would have to be uncompressed either each time a chunk is loaded or all of them at the start of the game. The second solution is to only save the chunks that have been modified. This would reduce the size considerably but it could also make the game slower as the chunks would have to be generated every time the game is loaded. The last solution would be to increase the size of cubes in the chunks. That could reduce the save size but it would also make the game look less smooth by making it more "blocky". All of these solutions are imperfect but some combination of them and making some performance sacrifices could be a viable solution.

9.2. World improvements

The world could be improved in multiple ways. One such way would be to add shadows to the game. Right now nothing casts any shadows but it would be interesting to see what shadows look like in non-Euclidean geometries.

Another improvement to the world would be to add clouds. The sky is very empty during the day with only the sun visible. During the night, due to the stars and other celestial bodies, the sky looks much better but clouds would make it look even better.

Perhaps the most important improvement to the world would be to add sound. Right now the game is completely silent which makes it feel dull. Adding a soundtrack would make the game more pleasant to play and adding sound effects would make the game feel more alive. Adding sound effects would also not be difficult to implement as there already is a system in place that detects collisions between objects.

The last improvement for this section would be distance fog. The world is infinite and is constantly generated which makes looking into the distance weird. At a certain distance, the world just stops and if the player moves in that direction the world will just appear out of nowhere. This undesirable effect could be completely hidden by a fog.

9.3. Fighting improvements

Currently, the bots shoot the player and the player can shoot and kill the bots. However, the bots cannot kill the player and there is no point in killing the bots. It does not reward the player

in any way. The bots could drop resources which the player could use to create new items or build new structures.

The weapon selection is also very limited. Both the player and the bots only have access to a single gun. Other kinds of weapons could be added, especially interesting among those would be weapons that explode and create craters in the ground as a result. The game allows for terrain modification so weapons like grenades or rocket launchers could be more realistic and fun to use than in games where the terrain is static.

If multiple weapons were added multiple types of bots could exist. Some bots with one type of weapon and some with another. Even bots that are not hostile could be added.

9.4. Settings improvements

The settings menu is quite simple. It gives the player options to start a new game, load a game, delete a save, and quit the game. These are the most important features and they work as intended. However, the menu could use some more features, especially considering how much work went into creating the menu framework described in section 4.6 and how developer-friendly the framework is.

These features include:

- Changing the resolution of the game,
- Changing the language of the game,
- Changing the render distance,
- Changing the FPS limit,
- Customizing the controls.

Bibliography

[1] Aidan Bell et al. *Elite*. 1984.

[2] CodeParade. 2022. URL: https://store.steampowered.com/app/1256230/Hyperbolica.

[3] CodeParade. *Hyperbolica - Official Trailer*. 2022. URL: https://www.youtube.com/watch?v=VYfWfrk5P7w&t=29s&ab_channel=CodeParade.

[4] CodeParade. *Spherical Geometry Is Stranger Than Hyperbolic - Hyperbolica Devlog #2*. 2020. URL: https://www.youtube.com/watch?v=yY9GAyJtuJ0&t=167s&ab_channel=CodeParade.

[5] The Editors of Encyclopaedia. *parallel postulate*. In: *Encyclopedia Britannica*. 2010. URL: https://www.britannica.com/science/parallel-postulate.

[6] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[7] Google Inc. *Flutter*. https://flutter.dev. 2018.

[8] Juan Linietsky, Ariel Manzur, and the Godot Community. *Design a title screen*. https://gamedevacademy.org/unity-start-menu-tutorial/. Accessed: 2023-12-29. 2019.

[9] William Lorensen and Harvey Cline. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm". In: *ACM SIGGRAPH Computer Graphics* 21 (Aug. 1987), pp. 163–. DOI: 10.1145/37401.37422.

[10] The Editors of Merriam-Webster. *"Open world." Merriam-Webster.com Dictionary*. Accessed 31 Dec. 2023. URL: https://www.merriam-webster.com/dictionary/open%20world.

[11] Ken Perlin. "An Image Synthesizer". In: *SIGGRAPH Comput. Graph.* 19.3 (July 1985), pp. 287–296. ISSN: 0097-8930. DOI: 10.1145/325165.325247. URL: https://doi.org/10.1145/325165.325247.

[12] Markus Persson and Jens Bergensten. *Minecraft*. 2011.

[13] Mark Phillips and Charlie Gunn. "Visualizing Hyperbolic Space: Unusual Uses of 4x4 Matrices". In: *Proceedings of the 1992 Symposium on Interactive 3D Graphics*. I3D '92. Cambridge, Massachusetts, USA: Association for Computing Machinery, 1992, pp. 209–214. ISBN: 0897914678. DOI: 10.1145/147156.147206. URL: https://doi.org/10.1145/147156.147206.

[14] Cristiano Politowski, Fábio Petrillo, and Yann-Gaël Guéhéneuc. "A Survey of Video Game Testing". In: *CoRR* abs/2103.06431 (2021). arXiv: 2103.06431. URL: https://arxiv.org/abs/2103.06431.

[15] László Szirmay-Kalos and Milán Magdics. "Adapting Game Engines to Curved Spaces". In: *The Visual Computer* 38.12 (Dec. 2022), pp. 4383–4395. ISSN: 1432-2315. DOI: 10.1007/s00371-021-02303-2. URL: https://doi.org/10.1007/s00371-021-02303-2.

[16] Matthew Szudzik and Eric W. Weisstein. *Parallel Postulate. From MathWorld—A Wolfram Web Resource*. Accessed on 12/25/2023. URL: https://mathworld.wolfram.com/ParallelPostulate.html.

[17] Michael Toy et al. *Rogue*. 1980.

[18] Joey de Vries. *Cubemaps*. Accessed on 12/29/2023. URL: https://learnopengl.com/Advanced-OpenGL/Cubemaps.

[19] Joey de Vries. *Light casters*. Accessed on 12/09/2023. URL: https://learnopengl.com/Lighting/Light-casters.

[20] Eric W. Weisstein. *Euclid's Postulates. From MathWorld—A Wolfram Web Resource*. Accessed on 12/25/2023. URL: https://mathworld.wolfram.com/EuclidsPostulates.html.

List of Figures