

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. The objective of this thesis is to create a small video game incorporating both of the aforementioned concepts.

Keywords: keyword1, keyword2, ...

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. Za cel niniejszej pracy obrano stworzenie aplikacji graficznej łączącej obydwie wspomniane elementy.

Słowa kluczowe: slowo1, slowo2, ...

Contents

1. Introduction	11
Introduction	11
2. Theoretical foundations	12
2.1. Non-Euclidean geometry	12
2.1.1. Analytic description	13
2.1.2. Transformations	14
2.1.3. Practical considerations	18
2.1.4. Teleporation in spherical space	21
2.2. Marching Cubes	23
3. System Architecture	26
3.1. Terrain Generation	26
3.1.1. Scalar Field	26
3.1.2. Chunks	27
3.1.3. Marching Cubes	28
3.2. Two Dimensional Graphics	29
3.2.1. Textures	29
3.2.2. Heads Up Display	30
3.2.3. Menu	31
4. User Manual	33
4.1. Launching the game	33
4.2. New Game Screen	34
4.3. Load Game Screen	34
4.4. Delete Save Screen	35
4.5. Controls	36
4.6. Items	37

1. Introduction

What is the thesis about? What is the content of it? What is the Author's contribution to it?

WARNING! In a diploma thesis which is a team project: Description of the work division in the team, including the scope of each co-author's contribution to the practical part (Team Programming Project) and the descriptive part of the diploma thesis.

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumyeirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumyeirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

2. Theoretical foundations

2.1. Non-Euclidean geometry

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

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. [**Weisstein-Parallel**]

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 [**Parallel-Postulate**]. 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*.

2.1.1. Analytic description

To describe the non-Euclidean geometries analytically, we will follow the approach given by [Szirmay-Kalos2022]. 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 2.1 if the embedding space was Euclidean. However, the

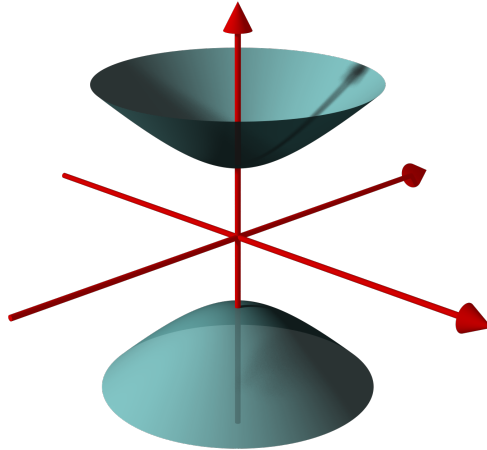


Figure 2.1: 2-dimensional hyperboloid embedded in Euclidean space

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

2.1.2. 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 2.2, can be defined as

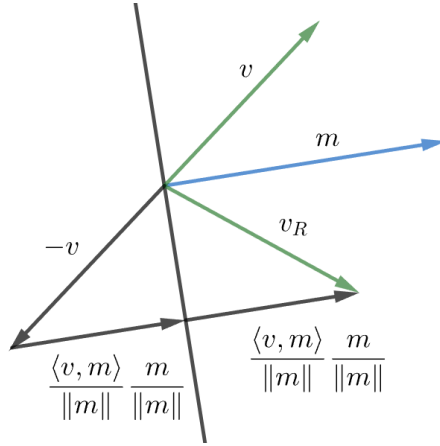


Figure 2.2: Reflection of v on vector m

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

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. $\|u - v\| = \|u_R - v_R\|$.

2.1. NON-EUCLIDEAN GEOMETRY

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

$$p'' = 2 \frac{\langle p', g + q \rangle}{\langle g + q, g + q \rangle} (g + q) - p' = 2p_w q + p - \frac{p_w + \mathcal{L} \langle p, q \rangle}{1 + q_w} (g + q). \quad (2.1)$$

The effect of applying translation to an arbitrary point a is shown in Figure 2.3.

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

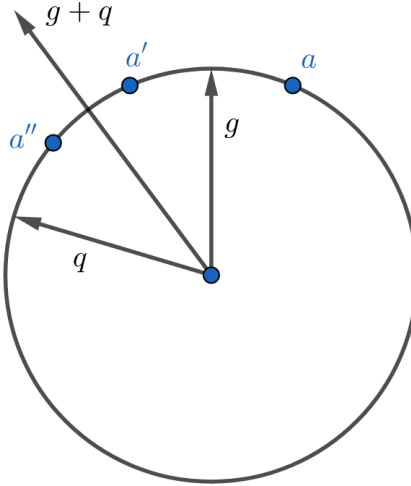


Figure 2.3: Translation of a point a

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

$$T(q) = \begin{bmatrix} 1 - \mathcal{L} \frac{q_x^2}{1+q_w} & -\mathcal{L} \frac{q_x q_y}{1+q_w} & -\mathcal{L} \frac{q_x q_z}{1+q_w} & -\mathcal{L} q_x \\ -\mathcal{L} \frac{q_y q_x}{1+q_w} & 1 - \mathcal{L} \frac{q_y^2}{1+q_w} & -\mathcal{L} \frac{q_y q_z}{1+q_w} & -\mathcal{L} q_y \\ -\mathcal{L} \frac{q_z q_x}{1+q_w} & -\mathcal{L} \frac{q_z q_y}{1+q_w} & 1 - \mathcal{L} \frac{q_z^2}{1+q_w} & -\mathcal{L} q_z \\ q_x & q_y & q_z & q_w \end{bmatrix} \quad (2.2)$$

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 [Philips-Mark-Gunn1992].

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 2.4 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

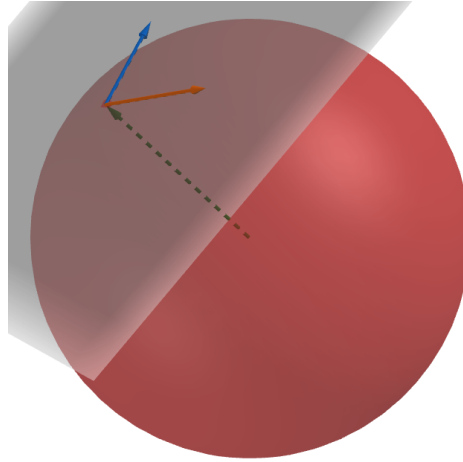


Figure 2.4: Tangent space of the camera

$$V = \begin{bmatrix} i'_x & j'_x & k'_x & \mathcal{L}e_x \\ i'_y & j'_y & k'_y & \mathcal{L}e_y \\ i'_z & j'_z & k'_z & \mathcal{L}e_z \\ \mathcal{L}i'_w & \mathcal{L}j'_w & \mathcal{L}k'_w & e_w \end{bmatrix} \quad (2.3)$$

2.1. NON-EUCLIDEAN GEOMETRY

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 [Szirmay-Kalos2022] (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

$$\mathcal{P}_E(p) = (p/d \sin(d), \cos(d)), \quad (2.4)$$

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 2.5.

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

$$v' = (v, 0)T(\mathcal{P}(p)), \quad (2.5)$$

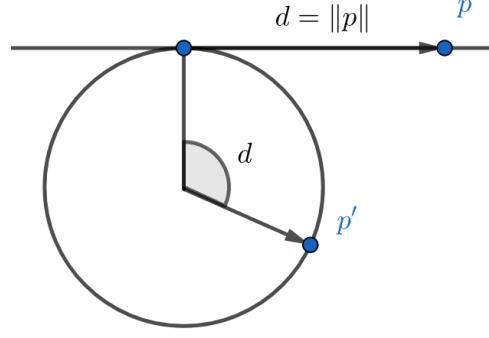


Figure 2.5: Exponential map

where T is the translation matrix 2.2.

2.1.3. 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 2.1,
2. Translate the object using ordinary Euclidean translation and then port it to non-Euclidean geometry.

We will now shortly discuss both options.

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 to spherical geometry using Equation 2.4. 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 $(1, 1)$.

2.1. NON-EUCLIDEAN GEOMETRY

When we perform the translation to point t_1 (the corresponding translation target is obtained by porting t_1 using Equation 2.4, 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 2.6. The effect is even more visible in the implementation. Figure 2.7 shows how

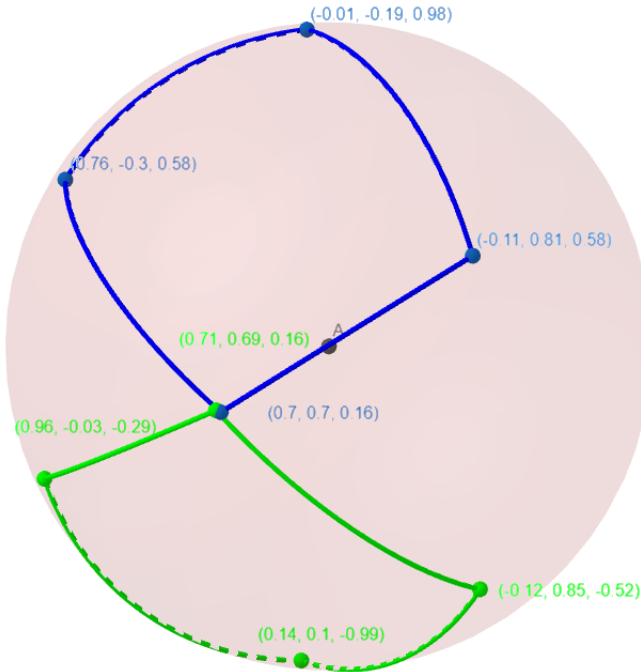


Figure 2.6: Rectangles ported onto a sphere and then translated

the wheels of the car get misaligned from their wheel arches.



Figure 2.7: Non-Euclidean translation causing a misalignment of objects

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 2.4, we get squares with side lengths (listed counter-clockwise starting at the bottom edge):

$$0.500, 0.479, 0.479, 0.500$$

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 geodesics¹ 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 2.8.

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 2.4, 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

$$\mathcal{P}_{E,1}(p) = (p/\|p\| \sin(\|p\|), \cos(\|p\|)) \quad (2.6)$$

¹This is the "great-circle distance" equal to $2 \arcsin(c/2)$, where c is the chord length.

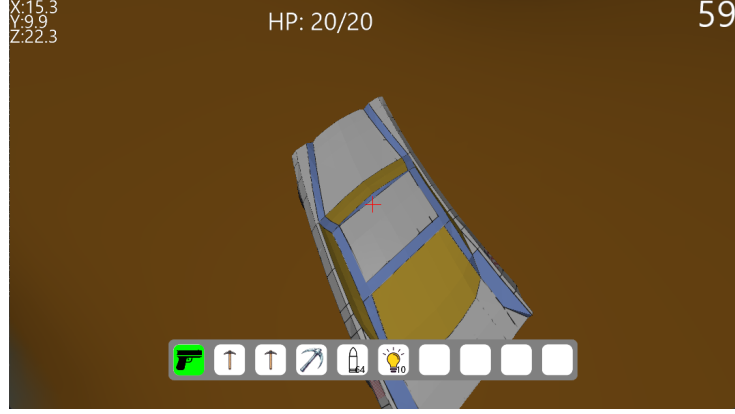


Figure 2.8: Distortions caused by porting to spherical space

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

$$\mathcal{P}_{E,2}(p) = (p'/\|p'\| \sin(\|p'\|), -\cos(\|p'\|)), \quad (2.7)$$

where $p' = p - c$. In the 2-dimensional case, the regions become disks with radii of length π , and the effect of using Equation 2.6 and Equation 2.7 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 2.1.4.

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.

2.1.4. Teleportation 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 2.9. In this 2-dimensional example, an object leaves the first region (centered at a) at the point p and is teleported to the second region

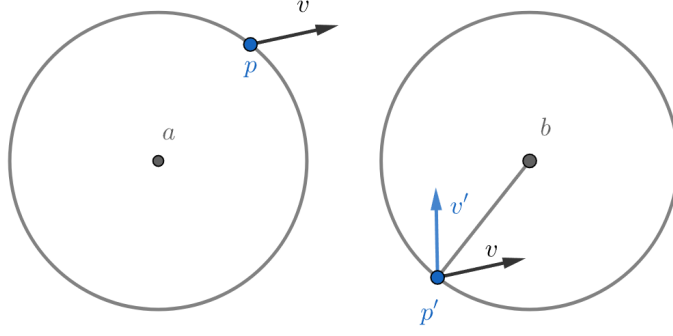


Figure 2.9: Teleportation between two regions

(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 2.7 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 2.3 is as follows. We first port the camera position to non-Euclidean space (in the case of the second region, we use Equation 2.7), and then use the Equation 2.5 to port its right, up, and front vectors. The problem with this approach is that the translation matrix 2.2 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 2.2, but in the context of

T_2 , the translation target q is the point that the point $(0, 0, 0, -1)$ is translated to. The matrix is given by

$$T_2(q) = \begin{bmatrix} 1 - \frac{q_x^2}{1-q_w} & -\frac{q_x q_y}{1-q_w} & -\frac{q_x q_z}{1-q_w} & q_x \\ -\frac{q_y q_x}{1-q_w} & 1 - \frac{q_y^2}{1-q_w} & -\frac{q_y q_z}{1-q_w} & q_y \\ -\frac{q_z q_x}{1-q_w} & -\frac{q_z q_y}{1-q_w} & 1 - \frac{q_z^2}{1-q_w} & q_z \\ -q_x & -q_y & -q_z & -q_w \end{bmatrix}. \quad (2.8)$$

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.

2.2. 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 a relatively straightforward algorithm that can be easily modified to suit different requirements, as explained in detail in section 3.1. 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 [**Marching-Cubes**]. The fundamental idea behind this algorithm is to generate a mesh from a scalar field. An isolevel is chosen, 0 being the most common choice and the one we used. Points 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 2.10, where v_0 is below the surface, while the remaining vertices are above it. It is important to note that the same effect would be achieved if v_0 were above the surface and the other vertices were below it.

TODO: Can we use images found online? If so how do we cite them?

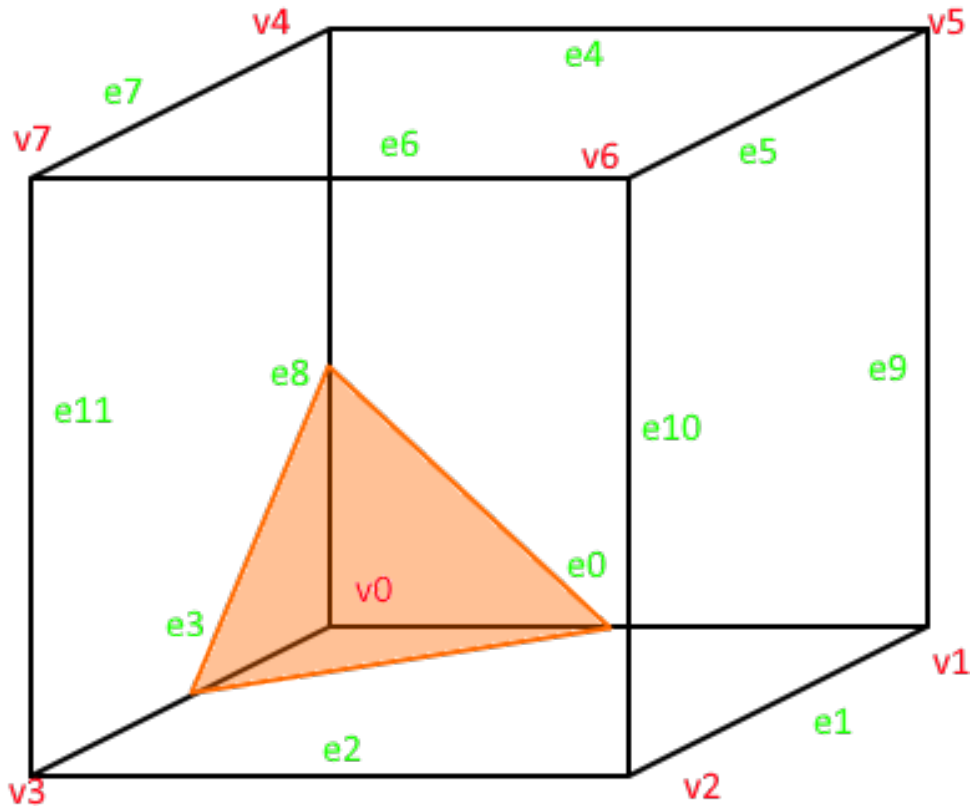


Figure 2.10: "Marching" cubes with only v_0 below the surface. Source: <https://polycoding.net/marching-cubes/>

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 2.11, 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.

2.2. MARCHING CUBES

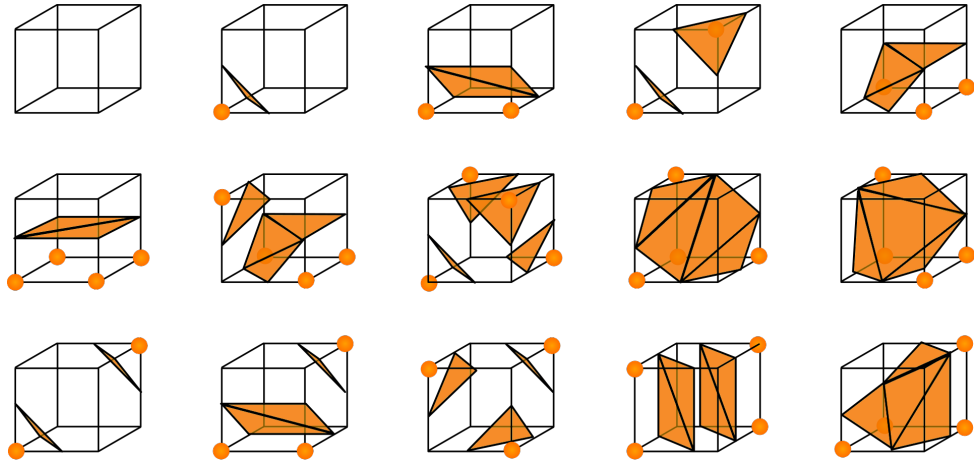


Figure 2.11: Unique marching cubes configurations. Source: <https://polycoding.net/marching-cubes/>

3. System Architecture

This section describes the system architecture of the game.

3.1. Terrain Generation

The terrain was designed to randomly generate so that the player can have a new, unique map every time they play. The second important part of the design was making sure that the player could edit the terrain in any way they wanted. As described in section 2.2 the marching cubes algorithm was chosen as the base of the terrain generation process. This section will describe how we used this algorithm to generate the terrain and allow the player to edit it.

The algorithm consists of the following steps:

- Define the scalar field function.
- Divide the world into chunks.
- Generate the mesh.

Each of these steps will be described in detail in this section.

3.1.1. Scalar Field

The first step in the terrain generation is to generate a scalar field which is a function that takes a point in 3D space and returns a value. 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 close values for close points. Our function returns values for points which 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 [**Perlin-Noise**] 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. **TODO: Should we explain how the Perlin noise function**

3.1. TERRAIN GENERATION

works? However unlike some 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 5 parameters: octaves, initial frequency, frequency multiplier, initial amplitude and amplitude multiplier. These parameters are generate based on the seed of the world from 5 different sets of options which gives the game 5 terrains. However we need more than just the value for each point and a normal vector. Each point is also assigned a type based on the position that is later used to determine the type of the block which in turn determines its color.

3.1.2. Chunks

As mentioned before one of the most important thinks 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 - cubes with side length 16. Each chunk is a separate object and can be edited independently. This solution is much more efficient but is 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 the same 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 exact 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 3.1.

Another problem is that the algorithm we used for generating the terrain, described in subsection 3.1.3, calculates normal vectors based on the values of the scalar field around the point at which the normal is calculated. This means that 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 3.2. 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 the 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.

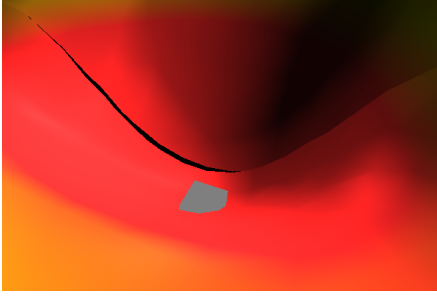


Figure 3.1: Gaps between chunks.

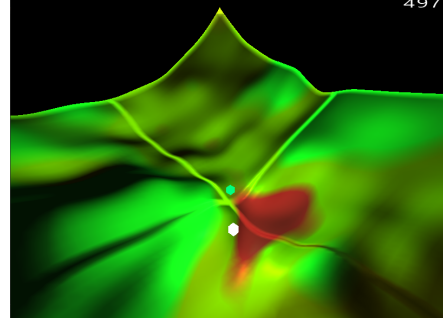


Figure 3.2: Problem with normals at chunk edges.

3.1.3. Marching Cubes

The idea of the algorithm is described in section 2.2. 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 on the edges based on the values of the scalar field at the vertices. This is done by using the linear interpolation given by equation Equation 3.1.

$$P = V_1 + \frac{\text{IsoLevel} - v_1}{v_2 - v_1} \times (V_2 - V_1) \quad (3.1)$$

where P is the resulting position of the vertex, V_1 and V_2 are the positions of the vertices on the edge, 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 for each vertex of the scalar field are calculated using Equation 3.2

$$n(x, y, z) = \begin{bmatrix} s(x+1, y, z) - s(x-1, y, z) \\ s(x, y+1, z) - s(x, y-1, z) \\ s(x, y, z+1) - s(x, y, z-1) \end{bmatrix} \quad (3.2)$$

where s is the scalar field and n is the normal vector. These vectors are used to calculate the mesh normals using the same interpolation used for the mesh Equation 3.1.

Last part of creating the mesh is assigning the colors to each vertex. Each vertex of the scalar field is assigned a type which is described in subsection 3.1.1. Each type has a color assigned to it. The color of each vertex of the mesh is calculated by interpolating the colors of the vertices of the scalar field using Equation 3.1.

3.2. Two Dimensional Graphics

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.

3.2.1. 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 (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 2 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 4.6. It also has a JSON file which contains the position and size of each sprite which can be seen in Figure 3.4. 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 SkiaSharp library to create a bitmap with all the ASCII characters. This bitmap is then converted to a texture. The position and size of each symbol is calculated using the font metrics.

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

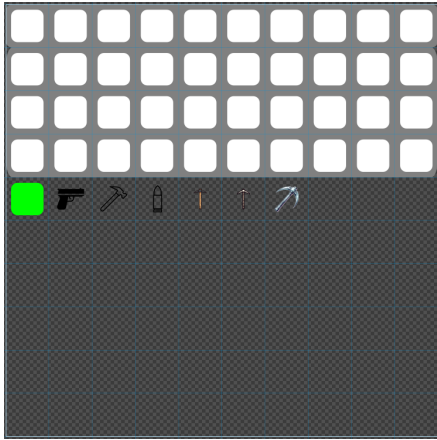


Figure 3.3: Inventory Sprite Sheet

```

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     }

```

Figure 3.4: Inventory Sprite Sheet JSON

3.2.2. 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 2 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 symbols 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 symbols texture.

The inventory is a collection of images that represent the items the player has. It is drawn at the bottom of the screen. It uses both the items texture containing the HUD items and the texture containing the alphabet. The images of the items are drawn first and then the text is

3.2. TWO DIMENSIONAL GRAPHICS

drawn on top of them. It is done in this order to optimize the performance by minimizing the number of times a texture is set to 2.

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

3.2.3. Menu

Menu is a lot more complicated than the HUD which is described in subsection 3.2.2. Because of that we decided to create a framework for creating menus. This framework was heavily inspired by Flutter. 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 which 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.

An example of a widget is shown in Figure 3.6. This widget renders the main menu of the game. The rendering logic of this widget can be seen in Figure 3.5. The idea is as follows. The root widget calls the render method of it's child which is the **Background** widget. The **Background** widget renders a background color. Then the **Background** widget calls the render method of it's child which is the **Column** widget. This widget renders it's 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 it's 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 it's 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 3.5 itself is a tree of widgets.

This approach of 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 3.2.2. 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 most popular game engines like Unity or Godot.

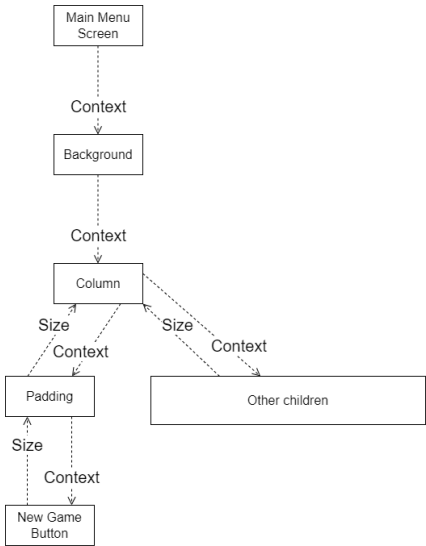


Figure 3.5: Widget rendering logic.

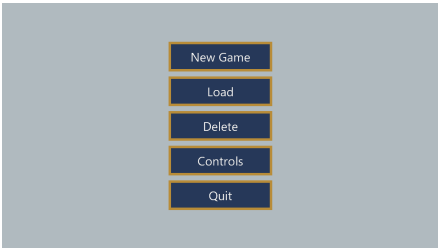


Figure 3.6: Main menu.

4. 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 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.

4.1. Launching the game

The application starts in the main menu shown in Figure 4.1. 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 show 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.

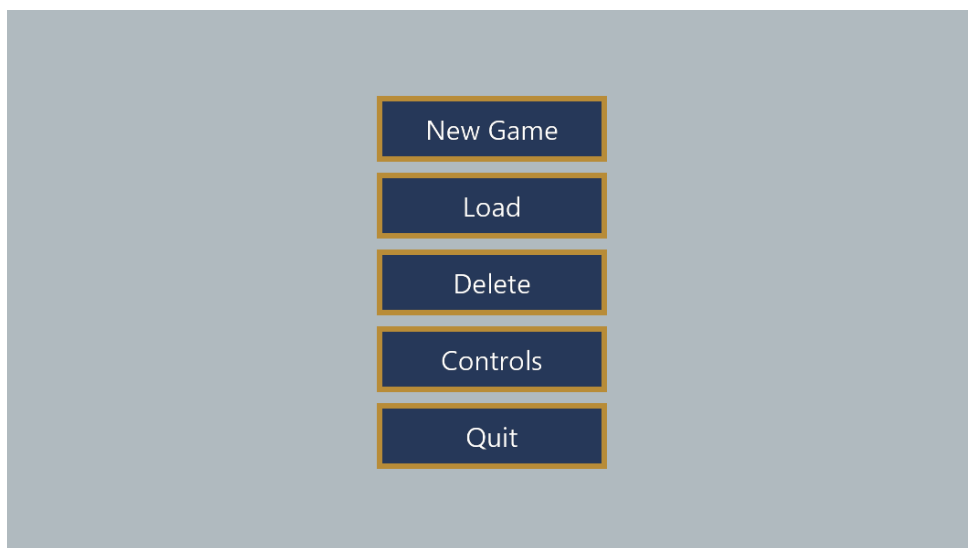


Figure 4.1: Main menu

4.2. New Game Screen

The game can be started from the New Game screen shown in Figure 4.2. 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 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.

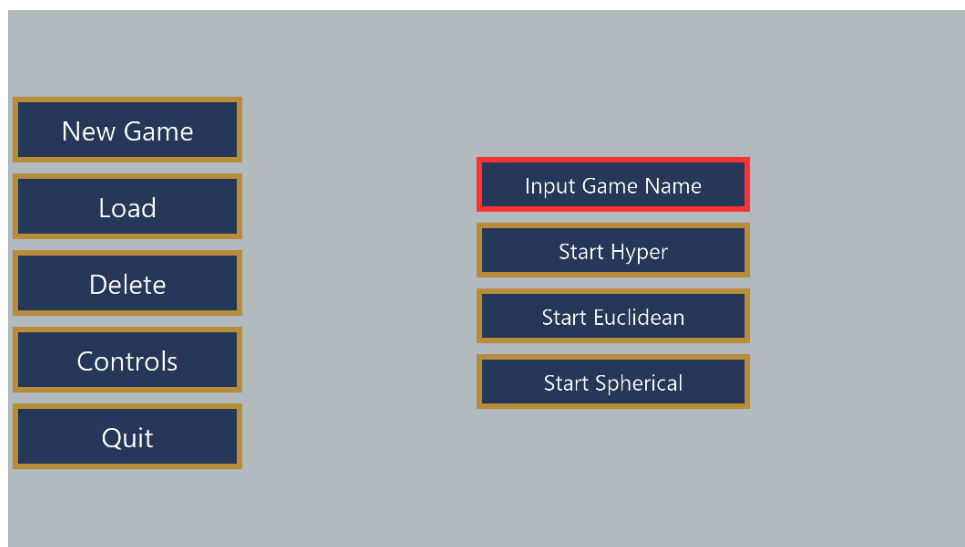


Figure 4.2: New Game screen

4.3. Load Game Screen

The game can be loaded from the Load Game screen shown in Figure 4.3. This screen displays your 9 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 4.4

4.4. DELETE SAVE SCREEN

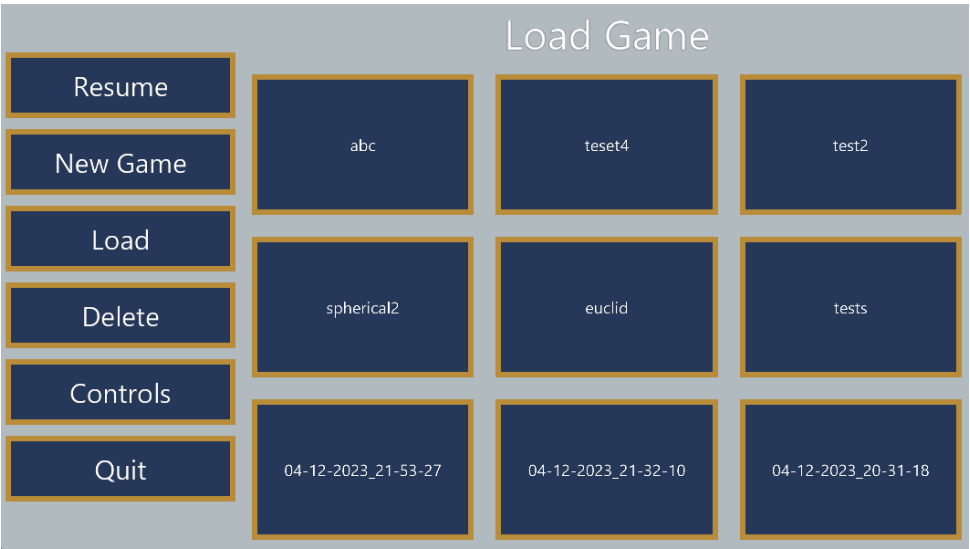


Figure 4.3: Load Game screen

4.4. Delete Save Screen

The "Delete Game" screen shown in Figure 4.4 allows you to delete your saves. It will display your 9 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 the currently running save.



Figure 4.4: Delete Save screen

4.5. Controls

The controls for the game are presented in Table 4.1. 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 4.5.















Key	Function
	Move Forward
	Move Back
	Move Left
	Move Right
	Sprint
	Jump
Left Mouse	Use Item
Right Mouse	Use Item's second ability
	Show/Hide Menu
	Switch Camera between 1st and 3rd person
Scroll	Change curvature (works only in hyperbolic geometry)
 through 	Select Item
	Enter Car
	Flip Car (works only outside the car)
	Leave Car
	Toggle Flashlight/Toggle car reflectors (when inside the car)

Table 4.1: Keyboard Key Functions for Game Controls



Figure 4.5: Controls screen

4.6. 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 4.2. To use the items the player has to first select the desired item using number keys through . The selected item will be highlighted in the inventory as shown in Figure 4.6, in which 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.






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 4.2: Table of in game items

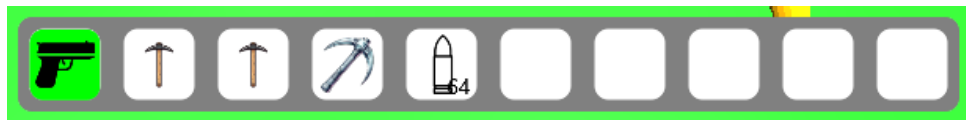


Figure 4.6: Inventory

List of symbols and abbreviations

nzw. nadzwyczajny

* star operator

~ tilde

If you don't need it, delete it.

List of Figures

2.1	2-dimensional hyperboloid embedded in Euclidean space	13
2.2	Reflection of v on vector m	14
2.3	Translation of a point a	15
2.4	Tangent space of the camera	16
2.5	Exponential map	18
2.6	Rectangles ported onto a sphere and then translated	19
2.7	Non-Euclidean translation causing a misalignment of objects	20
2.8	Distortions caused by porting to spherical space	21
2.9	Teleportation between two regions	22
2.10	"Marching" cubes with only v_0 below the surface.	24
2.11	Unique marching cubes configurations	25
3.1	Gaps between chunks.	28
3.2	Problem with normals at chunk edges.	28
3.3	Inventory Sprite Sheet	30
3.4	Inventory Sprite Sheet JSON	30
3.5	Widget rendering logic.	32
3.6	Main menu.	32
4.1	Main menu	33
4.2	New Game screen	34
4.3	Load Game screen	35
4.4	Delete Save screen	35
4.5	Controls screen	37
4.6	Inventory	

If you don't need it, delete it.

Spis tabel

4.1 Keyboard Key Functions for Game Controls 36

4.2 Table of in game items 37

If you don't need it, delete it.

List of appendices

1. Appendix 1
2. Appendix 2
3. In case of no appendices, delete this part.