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 obydwa wspomniane elementy.

Słowa kluczowe: slowo1, slowo2, ...

Contents

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 diamvoluptua. 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 diamvoluptua. 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 [11]:

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. [8]

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 [1]. 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 [7]. 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

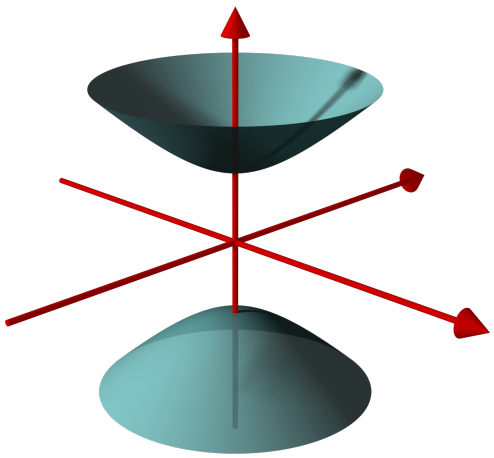


Figure 2.1: 2-dimensional hyperboloid embedded in Euclidean space

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 the generalized inner product

$$\langle u, v\rangle = u_x v_x + u_y v_y + u_z v_z + \mathcal{L} u_w v_w. \tag{2.1}$$

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

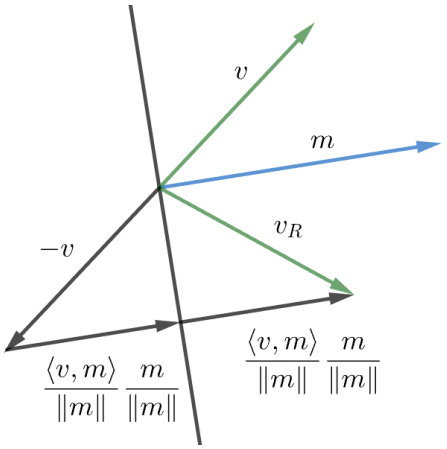


Figure 2.2: Reflection of v on vector m

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

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

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

$$p'' = 2\frac{\langle p', g + q\rangle}{\langle g + q, g + q\rangle}(g + q) - p' = 2p_w q + p - \frac{p_w + \mathcal{L}\langle p, q\rangle}{1 + q_w}(g + q). \tag{2.2}$$

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

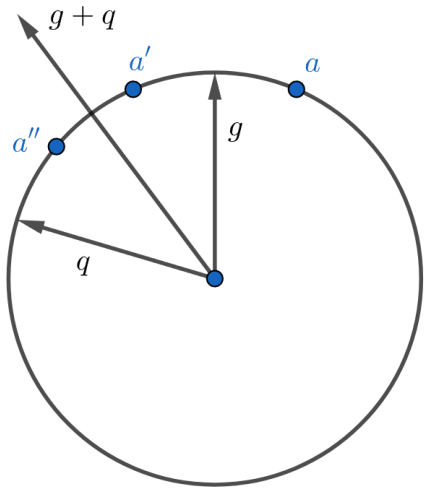


Figure 2.3: Translation of a point a

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

$$T(q) = \begin{bmatrix} 1 - \mathcal{L}\frac{q_x^2}{1+q_w} & -\mathcal{L}\frac{q_x q_y}{1+q_w} & -\mathcal{L}\frac{q_x q_z}{1+q_w} & -\mathcal{L}q_x \\ -\mathcal{L}\frac{q_y q_x}{1+q_w} & 1 - \mathcal{L}\frac{q_y^2}{1+q_w} & -\mathcal{L}\frac{q_y q_z}{1+q_w} & -\mathcal{L}q_y \\ -\mathcal{L}\frac{q_z q_x}{1+q_w} & -\mathcal{L}\frac{q_z q_y}{1+q_w} & 1 - \mathcal{L}\frac{q_z^2}{1+q_w} & -\mathcal{L}q_z \\ q_x & q_y & q_z & q_w \end{bmatrix} \tag{2.3}$$

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 [6].

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

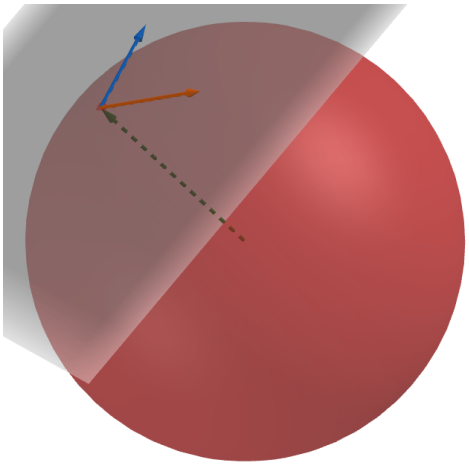


Figure 2.4: Tangent space of the camera

$$V = \begin{bmatrix} i'_x & j'_x & k'_x & \mathcal{L}e_x \\ i'_y & j'_y & k'_y & \mathcal{L}e_y \\ i'_z & j'_z & k'_z & \mathcal{L}e_z \\ \mathcal{L}i'_w & \mathcal{L}j'_w & \mathcal{L}l'_w & e_w \end{bmatrix} \tag{2.4}$$

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 [7] (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

$$\mathcal{P}_E(p) = (p/d\sin(d), \cos(d)), \tag{2.5}$$

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 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

$$v' = (v, 0)T(\mathcal{P}(p)), \tag{2.6}$$

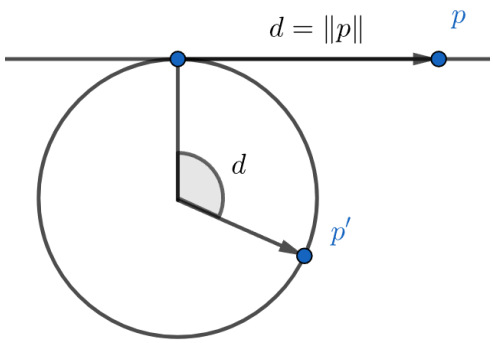


Figure 2.5: Exponential map

where T is the translation matrix 2.3.

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 2.2,

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 2.5. 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 2.5, 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 2.6. The effect is even more visible in the implementation. Figure 2.7 shows how

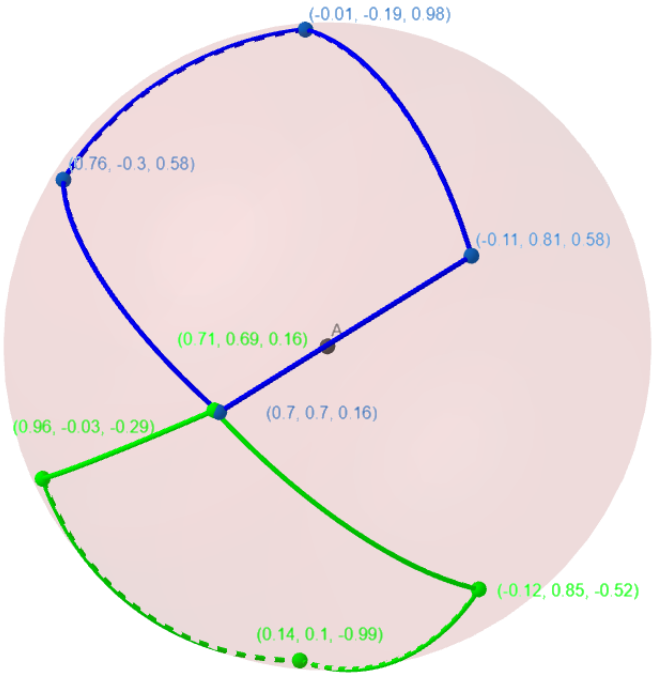


Figure 2.6: Rectangles ported onto a sphere and then translated

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 2.5, 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 geodesics[1] 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 2.5, 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

$$\mathcal{P}_{E,1}(p) = (p/\|p\| \sin(\|p\|), \cos(\|p\|)) \tag{2.7}$$

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

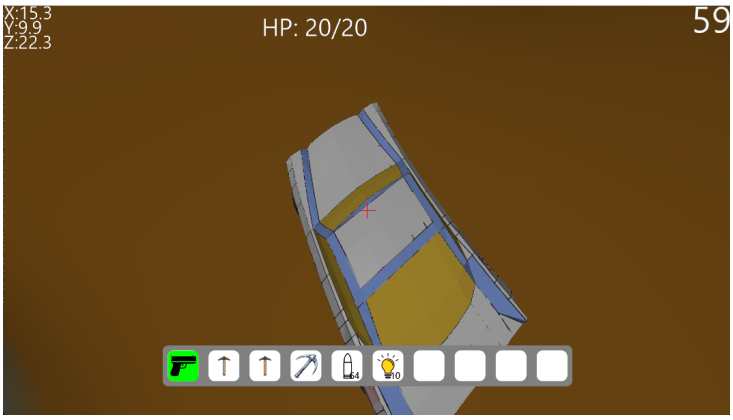


Figure 2.8: Distrotions caused by porting to spherical space

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

$$\mathcal{P}_{E,2}(p) = (p'/\|p'\| \sin(\|p'\|), -\cos(\|p'\|)), \tag{2.8}$$

where $p' = p - c$. In the 2-dimensional case, the regions become disks with radii of length π, and the effect of using Equation 2.7 and Equation 2.8 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 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. 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 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

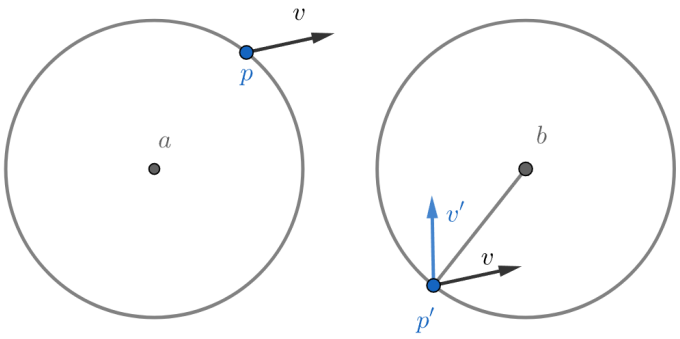


Figure 2.9: Teleportation between two regions

(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 2.8 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 2.4 is as follows. We first port the camera position to non-Euclidean space (in the case of the second region, we use Equation 2.8), and then use the Equation 2.6 to port its right, up, and front vectors. The problem with this approach is that the translation matrix 2.3 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 2.3, 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

$$T_2(q) = \begin{bmatrix} 1-\frac{q_x^2}{1-q_w} & -\frac{q_x q_y}{1-q_w} & -\frac{q_x q_z}{1-q_w} & q_x \\ -\frac{q_y q_x}{1-q_w} & 1-\frac{q_y^2}{1-q_w} & -\frac{q_y q_z}{1-q_w} & q_y \\ -\frac{q_z q_x}{1-q_w} & -\frac{q_z q_y}{1-q_w} & 1-\frac{q_z^2}{1-q_w} & q_z \\ -q_x & -q_y & -q_z & -q_w \end{bmatrix}. \tag{2.9}$$

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.

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 an algorithm that can be easily modified to suit different requirements, as explained in detail in section 3.2. 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 [4]. 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 $v0$ 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.

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

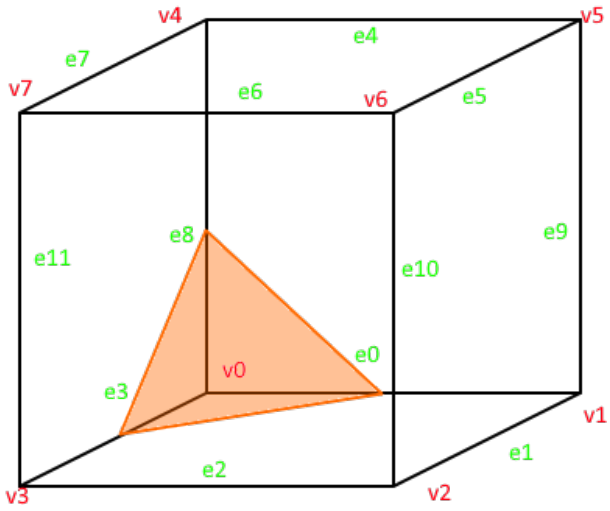


Figure 2.10: "Marching" cubes with only $v0$ below the surface. Source: `https://polycoding.net/marching-cubes/`

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 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.

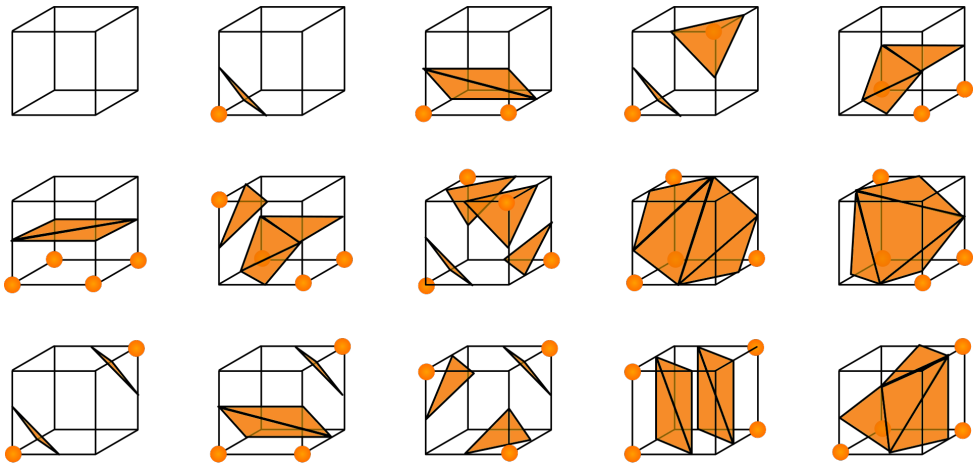


Figure 2.11: Unique marching cubes configurations. Source: `https://polycoding.net/marching-cubes/`

3. System Architecture

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. 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.

3.1. 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 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 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. 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) *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.

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 input) by registering input callbacks, i.e. implementing the `IInputSubscriber` interface.

3.2. 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. 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.

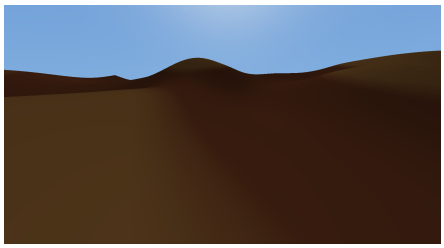
Each of these steps will be described in detail in this section.

3.2.1. Scalar Field

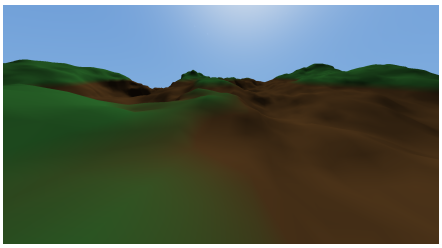
The first step in the terrain generation is to generate a scalar field which is a function that takes a point in 3D space and returns a value. What is important is that this function always returns the same value for the same point. Another important property is that the function should return close values for close points. Our function returns values for points 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 [5] 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 works?** However, unlike some random functions, it returns similar values for similar points. This makes it ideal for this game.

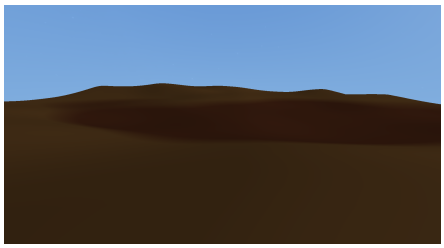
The Perlin noise function is used to generate a value for each point in the scalar field. This value is then modified based on 5 parameters: octaves, initial frequency, frequency multiplier, initial amplitude and amplitude multiplier. How these parameters affect the terrain can be seen in Figure 3.1. These parameters are generated 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.



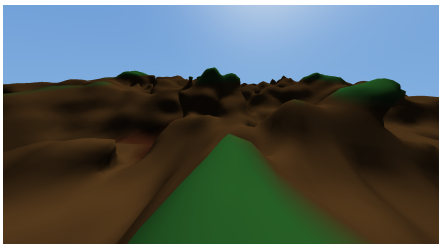
(a) Small octaves (1).



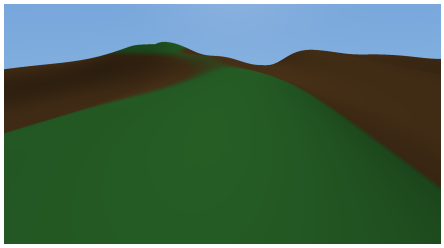
(b) Big octaves (5).



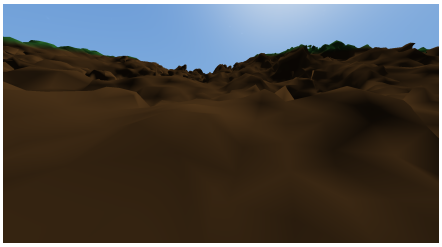
(c) Small initial frequency (1).



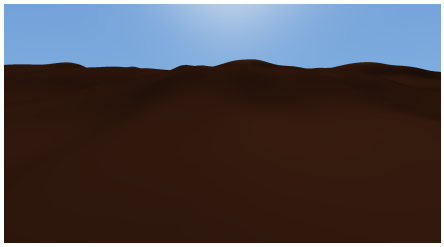
(d) Big initial frequency (0.5).



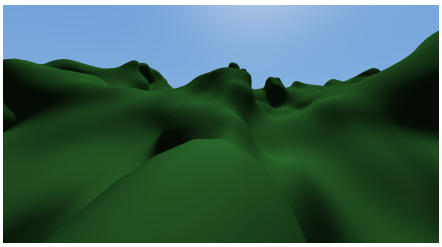
(e) Small frequency multiplier (0.5).



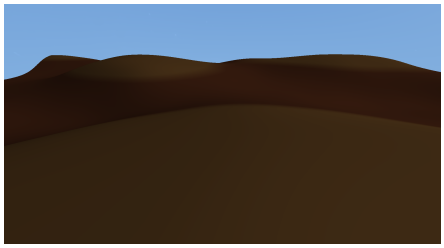
(f) Big frequency multiplier (5).



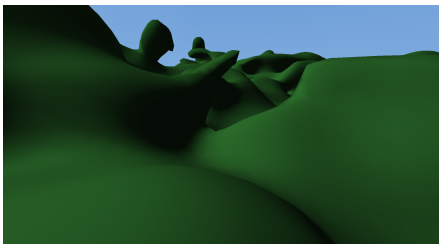
(g) Small initial amplitude (8).



(h) Big initial amplitude (32).



(i) Small amplitude multiplier (0.1).



(j) Big amplitude multiplier (1).

Figure 3.1: Scalar field parameter comparison.

3.2.2. 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 - cubes with side length 16. 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 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 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 3.2.

Another problem is that the algorithm we used for generating the terrain, described in subsection 3.2.3, 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 3.3. 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.

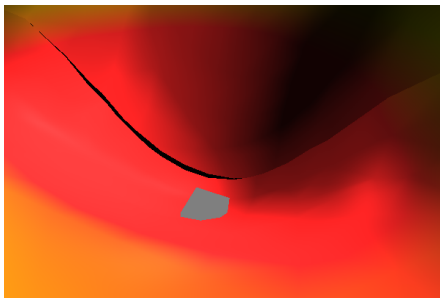


Figure 3.2: Gaps between chunks.

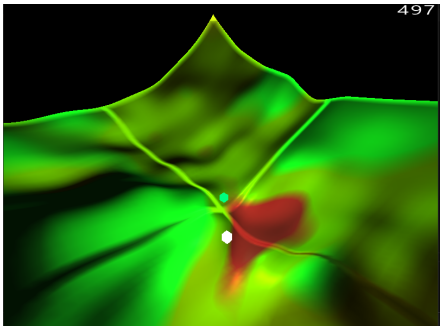


Figure 3.3: Problem with normals at chunk edges.

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

3.2.3. 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.

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

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

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

where $s(x,y,z)$ is the value scalar field function at (x,y,z) 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.

The 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 3.2.1. 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.

3.2.4. 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. Values of the scalar field close to that point are then changed based on the pickaxe the player uses and how long they mine for. The closer the point to the chosen point the more it is affected. A 3D Gaussian function is used to calculate exactly how much each point is affected. The function is shown in Equation 3.3.

$$f(x,y,z) = e^{-a\left((p_x-c_x)^2+(p_y-c_y)^2+(p_z-c_z)^2\right)} \tag{3.3}$$

In Equation 3.3 p_x, p_y and p_z are the coordinates of the point being edited, c_x, c_y and c_z are the coordinates of the chosen point and a is a constant. Moreover, if the distance between the

chosen point and the point being edited is greater than the range of the pickaxe, the point is not affected.

Once the values of the scalar field are updated the chunk 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 4.7.

3.3. 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.

3.4. 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.

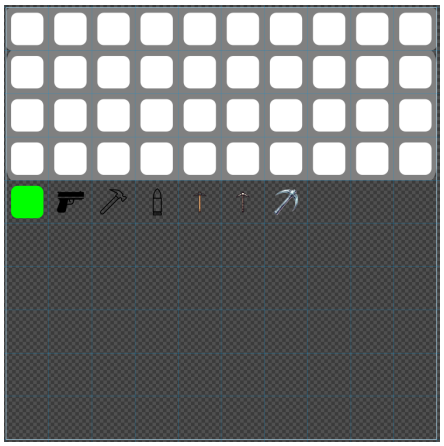
3.4.1. Textures

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 5.6. It also has a JSON file that contains the position and size of each sprite which can be seen in Figure 3.4b. 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, 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

```
{
    "width": 10,
    "height": 10,
    "items": [
        {
        "name": "hotbar",
        "x": 0,
        "y": 0,
        "width": 10,
        "height": 1
        },
    ]
}
```

(b) JSON file

Figure 3.4: Inventory Sprite Sheet

3.4.2. 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 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. 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. 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 positions and sizes. Each element is either placed in some position on the screen or is placed relative to some other element.

3.4.3. Menu

The menu is a lot more complicated than the HUD which is described in subsection 3.4.2. Because of that, we decided to create a framework for creating menus. This framework was heavily inspired by Flutter [2]. 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.

An example of a widget is shown in Figure 3.5b. This widget renders the main menu of the game. The rendering logic of this widget can be seen in Figure 3.5a. The idea is as follows. The root widget calls the render method of its child which is the `Background` widget. The `Background` widget renders a 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 3.5a 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 3.4.2. 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 [3]. **TODO: Is this citation enough for this claim?**

(a) Widget rendering logic.

(b) Main menu.

Figure 3.5: Example widget.

3.5. Technologies selection

TODO: I don't know where to put but it definitely shouldn't be here 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: **TODO: should links be in footnotes or in bib?**

- 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,
- 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.

[1] `https://github.com/opentk/opentk`
[2] `https://github.com/bepu/bepuphysics2`
[3] `https://github.com/mono/SkiaSharp`
[4] `https://github.com/MonoGame/AssimpNet`
[5] `https://assimp.org/`
[6] `https://github.com/StbSharp/StbImageSharp`

4. Main components

TODO: write something about the components design overview or whatever

4.1. Animator

The `Animator` class is responsible for animating the models loaded by `ModelLoader`. It takes animation keyframes and interpolates them using Quaternion Slerp and Vector3 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. **TODO: Do we want to describe this in a little more detail? Also providing screenshots from Blender or from the game would be nice I guess**

4.2. Model Loader

The model loader is responsible for loading models from the file system. It loads COLLADA files using AssipmNet to convert them to model class objects. Each vertex in the mesh of the model must be dependent on at most 4 bones. It creates VAOs 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. **TODO: I think we could write more about the modeling process itself (not necessarily here). Give some screenshots from Blender as well. It was a lot of work after all**

4.3. Printer

The `Printer` class is responsible for printing text on the screen. It has a static constructor in which it creates a texture with the alphabet. To achieve that it uses the SkiaSharp library. After the texture is created `Printer` creates an array of Rectangles that describe the position of each letter on the texture. These rectangles are passed to the shader when the text is printed.

This way only one texture is passed to the shader and the shader uses the rectangles to get the correct letter from the texture which improves performance. The class also provides a method for printing a letter which is used in another method to print whole strings. There are also methods for printing strings with the specified corner at the specified position.

4.4. Sprite Renderer

TODO: this has to be rewritten because it says the same things as textures.tex The `SpriteRenderer` class is responsible for rendering 2D sprites. It takes a PNG file and a JSON file as input and creates a texture and an array of rectangles that describe the position of each sprite on the texture. It then uses the texture and the rectangles to render the sprites by passing the rectangle coordinates to the shader. It only sets the texture once to boost performance.

An exemplary JSON file describing the position of the sprites on the texture is shown below:

```
{
  "width": 10,
  "height": 10,
  "items": [
    {
      "name": "someItem",
      "x": 2,
      "y": 3,
      "width": 2,
      "height": 1
    },
    \dots
  ]
}
```

The width and height describe the size of the sprite sheet. The `x,y` coordinates describe the position of the sprite on the sprite sheet. The width and height describe the size of the sprite. The name of the item is used to identify the sprite.

TODO: Screenshots? FPS, position, ...

4.5. Menu

The `Menu` class is responsible for rendering the menu and handling user input. It makes use of `SpriteRenderer` to render buttons and graphics. It makes use of the `Printer` class to render text. The menu is described in more detail in subsection 3.4.3.

4.6. Chunk Generator

TODO: we should describe that the scalar field generation is different for spherical space ("stitching the spheres") The Chunk Generator component is responsible for procedural terrain generation. The generation process is encapsulated inside the `ChunkFactory` class.

A chunk is generated in two main steps:

1. A scalar field of a given size is created. The values of the scalar field are generated based on the values of Perlin noise. This step is performed by the `ScalarFieldGenerator` class.

2. The marching cubes algorithm is used to create a mesh; positions and normal vectors of the mesh's vertices are obtained in this step using the `MeshGenerator` class.

For more information on terrain generation, refer to section 3.2.

4.7. 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.

4.7.1. 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.

4.7.2. Saving chunks

Saving chunks is handled by a process similar to the one discussed in subsection 4.7.1. 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.

4.7.3. 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 4.1.

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

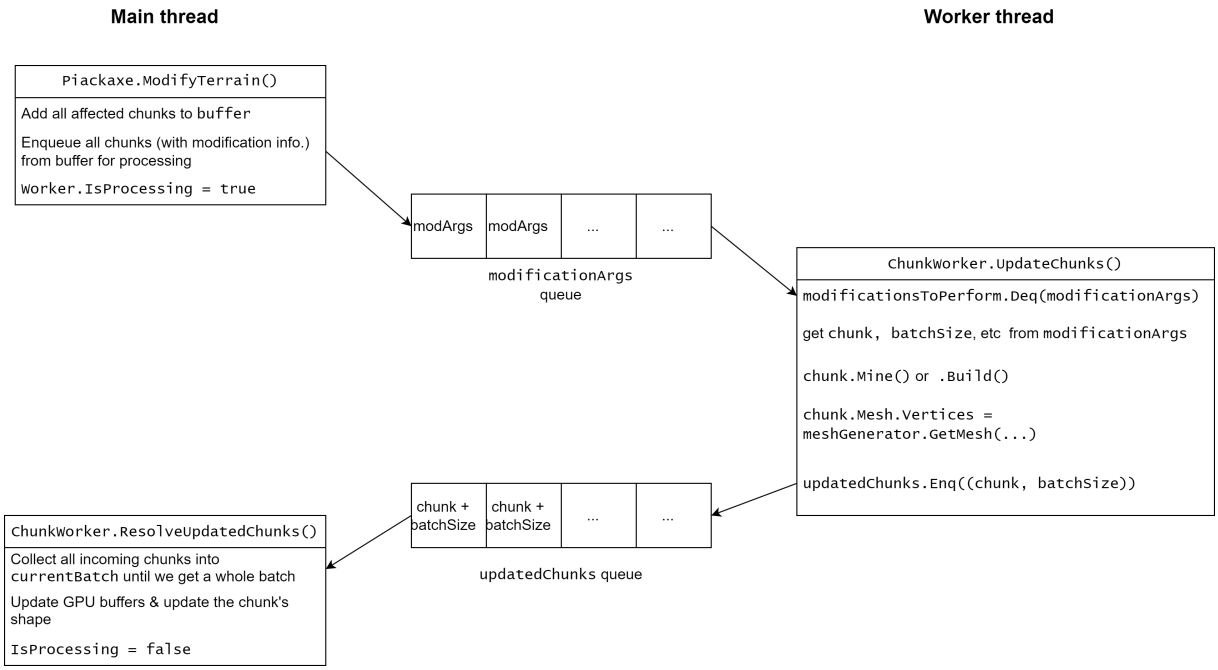


Figure 4.1: Terrain modification in the chunk management system

problem can be seen in Figure 4.2 which comes from an early stage of the game's development.

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.

4.8. Physics

The Physics component is responsible for collision detection, ray casting, and physical modeling. 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. The body that represents the terrain was created as a separate shape from the terrain mesh triangle-by-triangle.

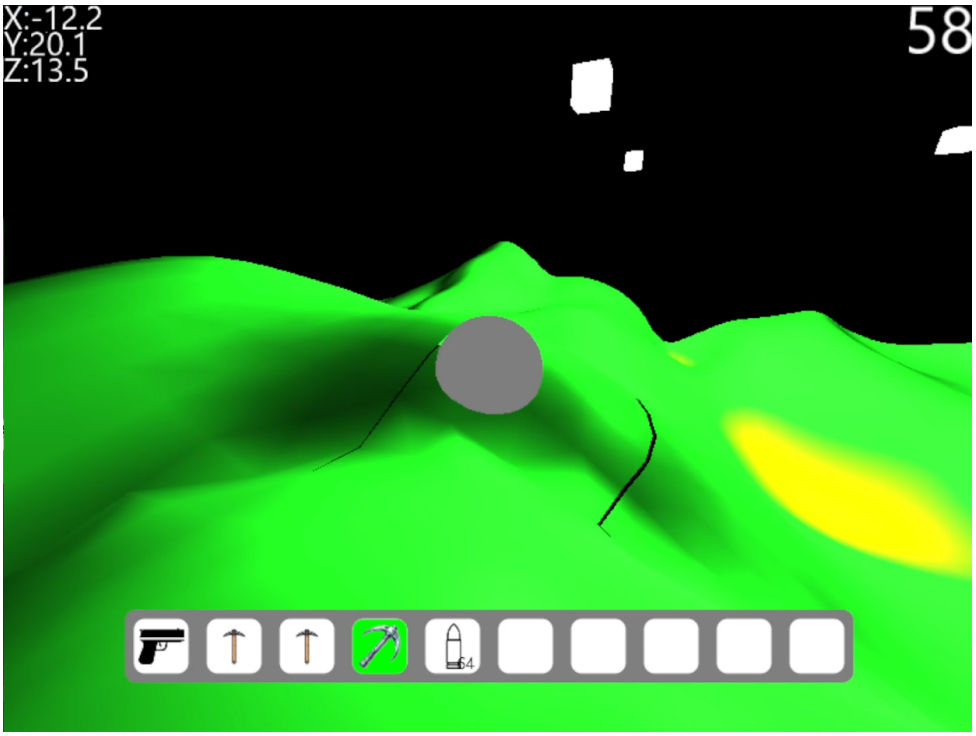


Figure 4.2: Gaps between chunks appearing during terrain modification

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. **TODO: Add screenshots of bounding shapes, maybe a screenshot of some huge number of bots to see that we can handle that.**

4.9. Scene

The `Scene` class is responsible for storing all the objects in the game the player can interact with. It is also responsible for disposing of them. Here is the full list of objects stored in a `Scene` instance:

- Chunks

- LightSources
- Projectiles
- Bots
- Cars
- Player
- Camera
- SimulationMembers
- SimulationManager

4.10. Controllers

There are eight controller classes in the project:

- `PlayerController`
- `BotsController`
- `ChunksController`
- `ProjectilesController`
- `VehiclesController`
- `LightSourcesController`
- `HudController`
- `BoundingShapesController`
- `SkyboxController`

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

4.11. Context

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.

4.12. Transporter

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.

4.13. Bots management

TODO: Not sure if bots logic is described anywhere, we can give a description here

5. User Manual

TODO: Do we want to mention that a log file with exception details is created in case the game crashes? Also should we mention how to download the game (copy installation instruction from deliv. 5?) 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.

5.1. Launching the game

The application starts in the main menu shown in Figure 5.1. 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 5.1: Main menu

5.2. New Game Screen

The game can be started from the New Game screen shown in Figure 5.2. 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.

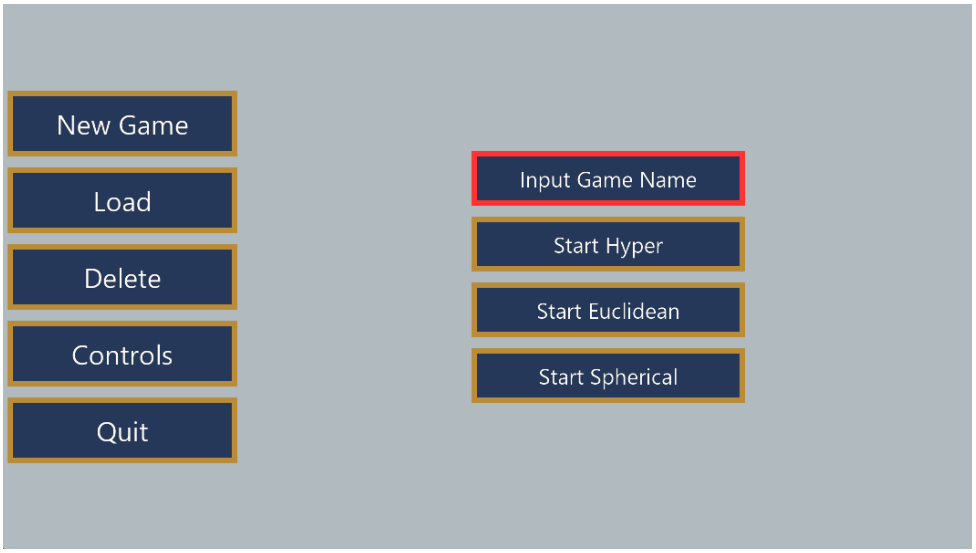


Figure 5.2: New Game screen

5.3. Load Game Screen

The game can be loaded from the Load Game screen shown in Figure 5.3. 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 5.4

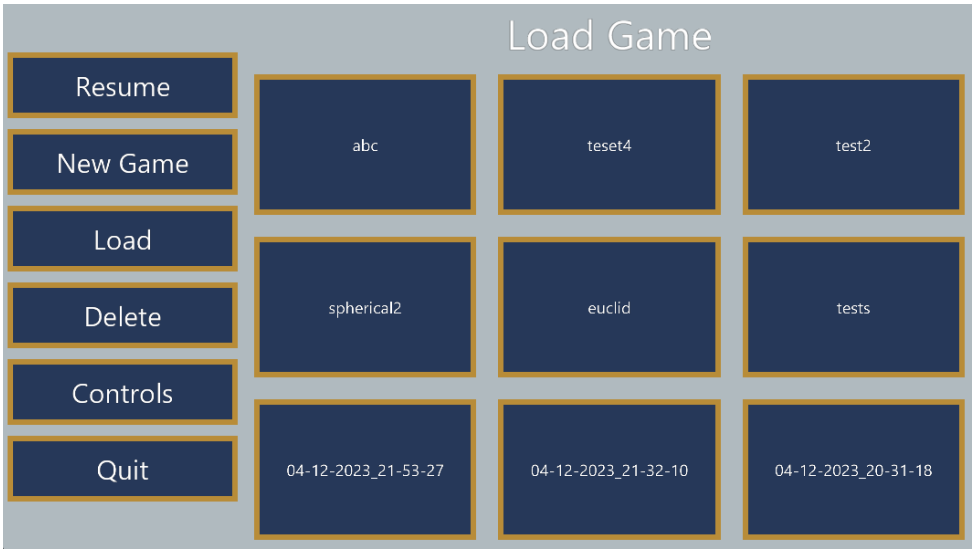


Figure 5.3: Load Game screen

5.4. Delete Save Screen

The "Delete Game" screen shown in Figure 5.4 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.

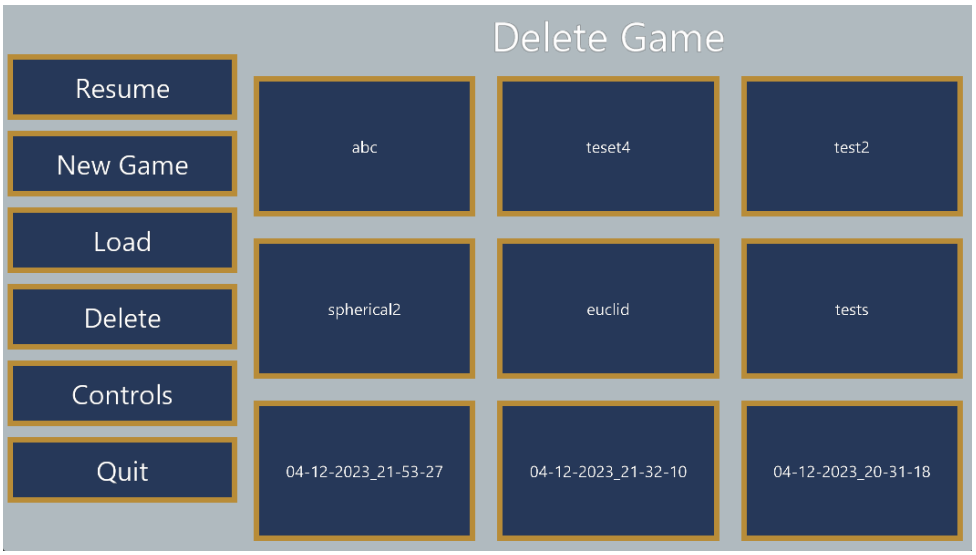


Figure 5.4: Delete Save screen

5.5. Controls

The controls for the game are presented in Table 5.1. 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 5.5.

Figure 5.5: Controls screen

5.6. 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 5.2. 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.
Y	Toggle showing hitboxes.

Table 5.1: 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 5.6, 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.

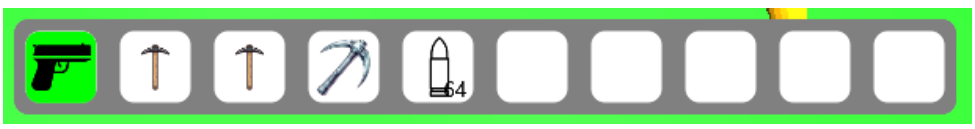


Figure 5.6: Inventory

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 5.2: Table of in-game items

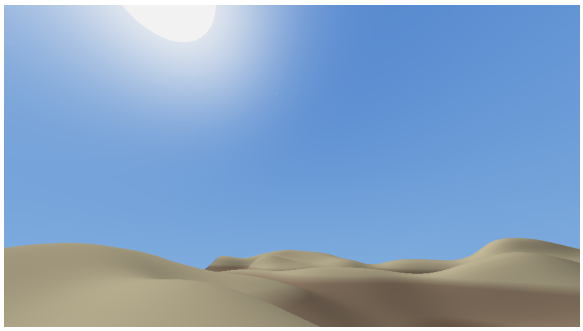
6. Functionalities

TODO: This chapter should have a different folder name and the tex file should also be renamed. Maybe we could split the thesis into chapters like Theoretical foundations, System architecture, Functionalities/Features/??? (move the terrain stuff here as well), ...

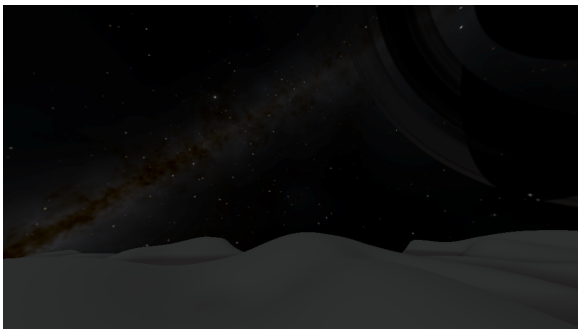
6.1. Environment

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. One such element is the Earth-like terrain, described in detail in the previous sections. Another 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 6.1.

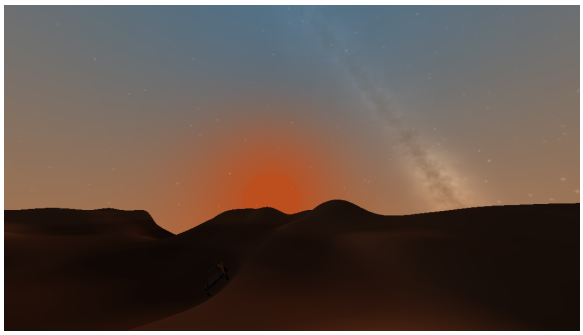
The implementation of the day-night cycle relies on two components: directional lighting (described in subsection 6.2.1) which corresponds to the light coming from the sun and a *skybox* representing the sky. Conceptually, a skybox is a cube made out of 6 images, one per each side, that encompasses the 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 6 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 to make the stars appear larger. **TODO: Could we do it?** 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).



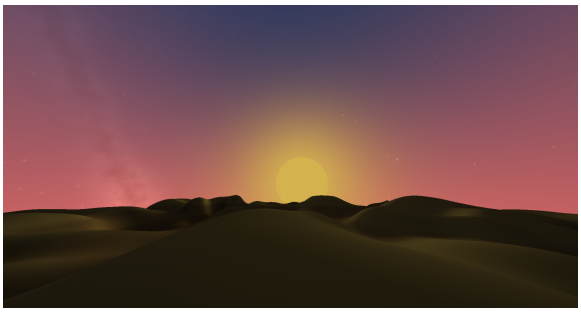
(a) Day



(b) Night



(c) Sunrise



(d) Sunset

Figure 6.1: Day night cycle in the game

The view matrix has to be modified so that the skybox doesn't move along with the camera. The part responsible for translation can be removed from the view matrix by replacing the last row the the view matrix with the vector $[0, 0, 0, 1]$ [9].

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 corresponding colors for the previous and 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[1]. 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.

[1]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.

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".

6.2. Lighting

Lighting is an important aspect of the game, it makes the game more immersive and has a major impact on how the player perceives the game. 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

$$L = k_a I_a + \sum_{i=1}^{N} k_d I_{i,d} \max(0, \langle n, l_i, \rangle) + k_s I_{i,s} \max(0, \langle r_i, v \rangle^{\alpha}), \tag{6.1}$$

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 2.1. The reflected light vector r is calculated using the usual formula

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

6.2.1. 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 6.2 shows the terrain and other game objects illuminated by the directional light of orange color.

6.2.2. Point lights

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

Figure 6.2: Directional light

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.

$$d_E(a, b) = \sqrt{\langle a - b, a - b \rangle_E}. \tag{6.2}$$

For non-Euclidean geometries, we use modified formulas given by [7], namely

$$l = \frac{p - f\cos(d_S(p, f))}{\sin(d_S(p, f))} \tag{6.3}$$

for spherical geometry and

$$l = \frac{p - f\cosh(d_H(p, f))}{\sinh(d_H(p, f))} \tag{6.4}$$

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

$$d_S(a, b) = \cos^{-1}(|\langle a, b \rangle_E|) \tag{6.5}$$

and

$$d_H(a, b) = \cosh^{-1}(-\langle a, b \rangle_L) \tag{6.6}$$

respectively.

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

$$a = \frac{1}{K_c + K_l d + K_q d^2}, \tag{6.7}$$

where d is the distance of the fragment from the source that can be calculated using one of the formulas 6.2, 6.5, or 6.6 depending on the geometry. Multiplying the light by the attenuation factor gives the desired effect of a realistic point light source such as a lamp, see Figure 6.3. Point lights in hyperbolic and spherical geometry are shown in Figure 6.4.

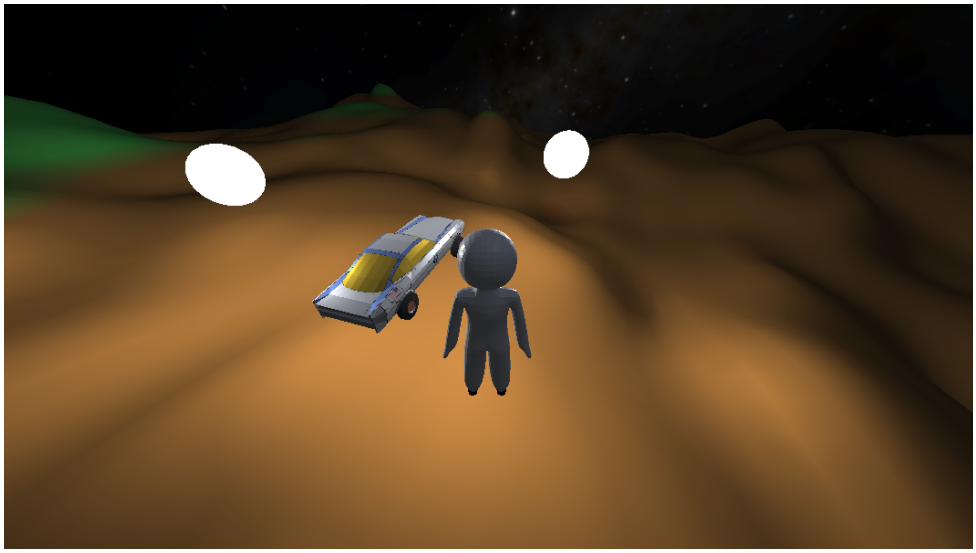
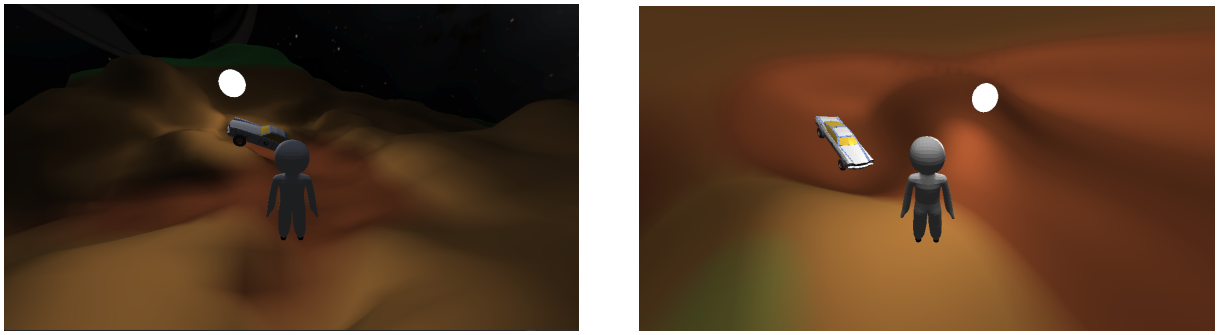


Figure 6.3: Point lights



(a) Hyperbolic space

(b) Spherical space

Figure 6.4: Point lights in non-Euclidean spaces

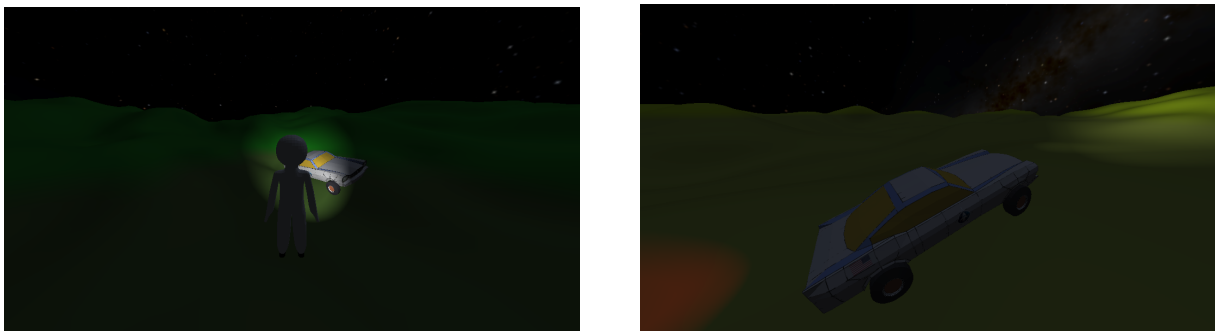
6.2.3. Spotlights

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 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 [10]. The intensity coefficient is then used to multiply each component of the light, giving the result shown in Figure 6.5.



(a) Player's flashlight

(b) Car illumination

Figure 6.5: Spotlights used in the game

TODO: Add chapter with results (what we accomplished, what we failed to accomplish, plans for the future development, etc.)

Bibliography

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

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

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

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

[11] 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 symbols and abbreviations

nzw.	nadzwyczajny
$*$	star operator
$\sim$	tilde

If you don't need it, delete it.

List of Figures

If you don't need it, delete it.

List of Tables

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.