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 **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. Chunk worker 23

2.2.1. Loading and generating chunks 23

2.2.2. Saving chunks 24

2.2.3. updating chunks 24

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 [5]:

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. [4]

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 [3]. 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

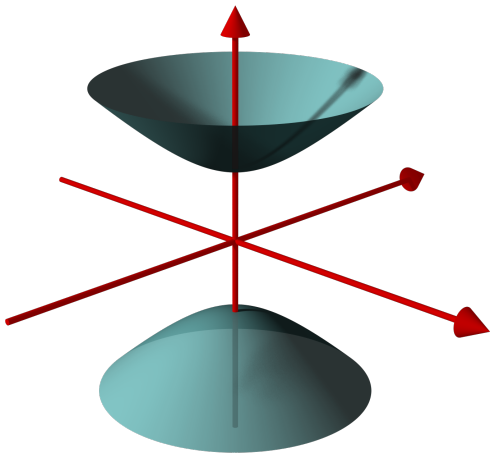

Figure 2.1: 2-dimensional hyperboloid embedded in Euclidean space

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

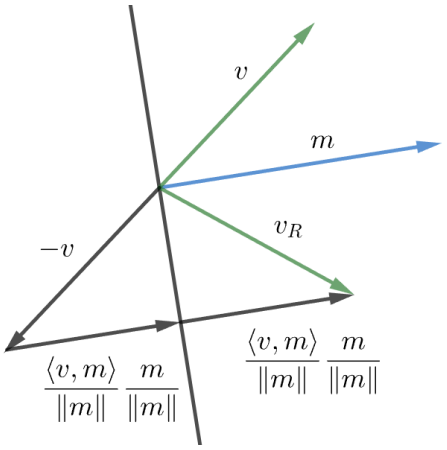

Figure 2.2: Reflection of v on vector m

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

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

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

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

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

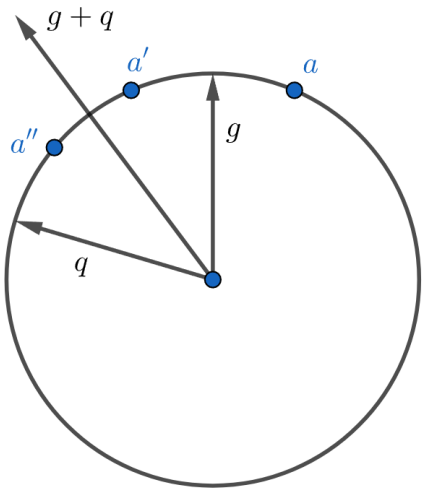

Figure 2.3: Translation of a point a

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

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

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 [2].

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

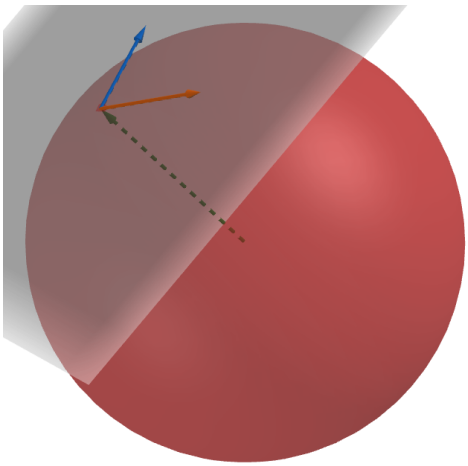

Figure 2.4: Tangent space of the camera

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

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 [3] (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

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

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

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

where T is the translation matrix 2.2.

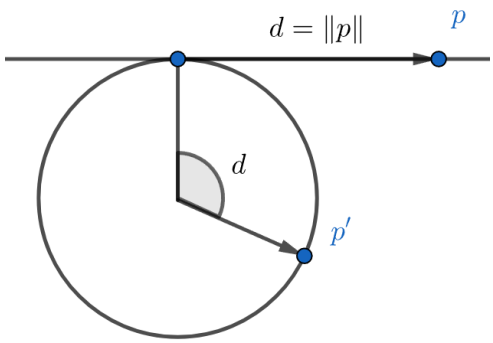

Figure 2.5: Exponential map

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 $(1, 1)$.

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

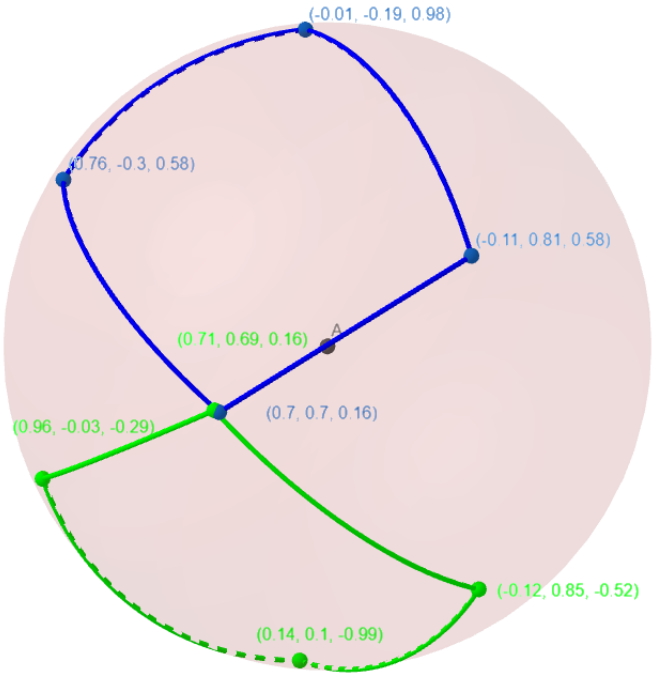

Figure 2.6: Rectangles ported onto a sphere and then translated

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

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

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

$(0, 0)$ 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 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.

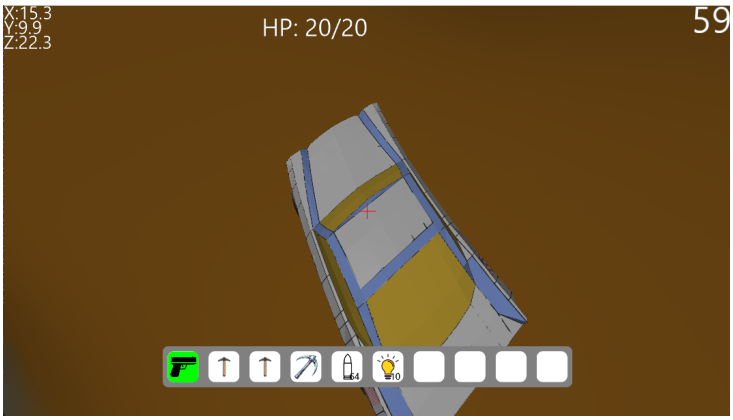

Figure 2.8: Distrotions caused by porting to spherical space

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

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

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

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

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

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. 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 (centered at b), appearing at a location given by the point p':

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

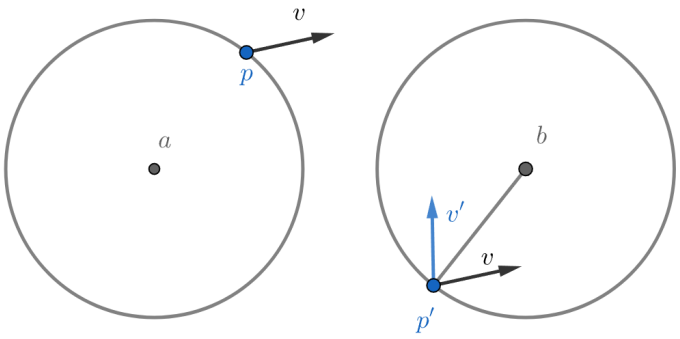

Figure 2.9: Teleportation between two regions

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

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

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. Chunk worker

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.

2.2.1. 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.

2.2.2. Saving chunks

TODO

2.2.3. updating chunks

Terrain modification consists of two main steps:

1. the scalar field for the modified chunk has to be modified, 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 very often how often (each render frame or more often?? – yeah each render frame, there's a time accumulator in place) 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. Once the whole `currentBatch` is updated, the `IsProcessingBatch` flag is set back to false.

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 problem can be seen in Figure 2.10 which comes from an early stage of the game's development.

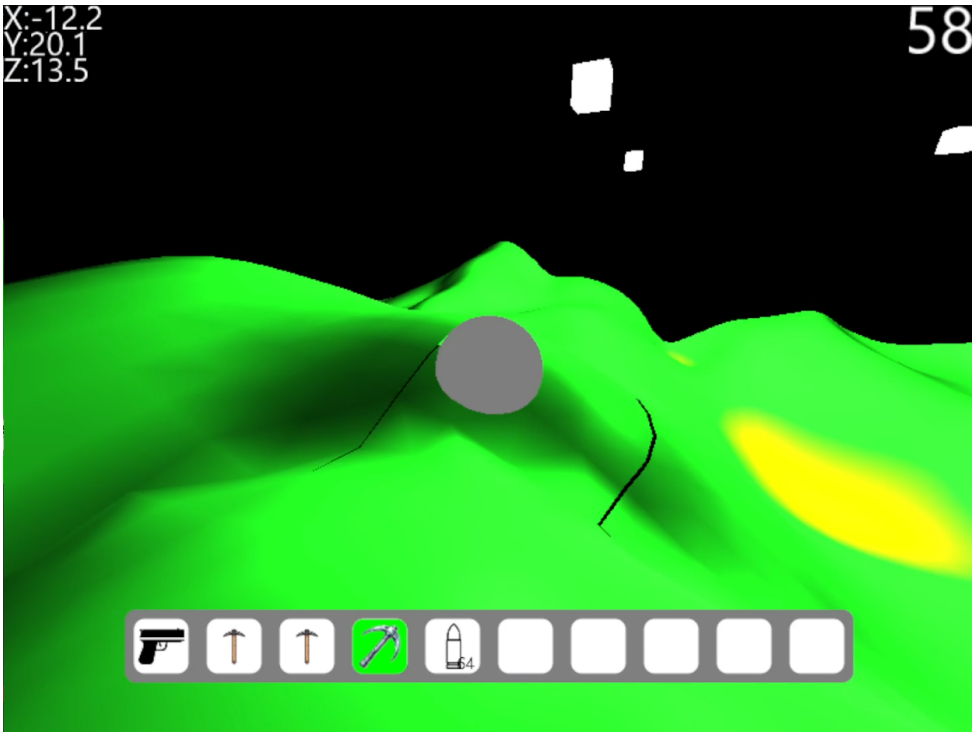

Figure 2.10: Gaps between chunks

TODO: write about the zero time stuff because it may seem like we can be losing modifications f f the workewr thread is takin g unsusually long to preocess

Bibliography

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

[2] 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.

[3] 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.

[4] 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.

[5] 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

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 Distrotions caused by porting to spherical space 20
2.9 Teleportation between two regions 22
2.10 Gaps between chunks 25

If you don't need it, delete it.

Spis tabel

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.