City, University of London

School of Mathematics, Engineering and Computer Science

Department of Computer Science

Real-Time Physics and Graphics Engine for non-Euclidean Geometry using Spherical and Hyperbolic Trigonometry

Daniil Osudin

A thesis submitted for the degree of
Doctor of Philosophy at City, University of London

April 2022

Contents

List of Figures

ACHNOWLEDGEMENTS:

I’d like to thank my first supervisor, Dr Christopher Child, for the guidance and support throughout the length of this project as well as taking over the role of my first supervisor. Our biweekly meetings have been vital to my academic research, providing in-depth discussions, helpful advice and constructive critique at every stage of the project. I would also like to thank my second supervisor, Professor Yang-Hui He, who has helped me better understand the mathematics behind non-Euclidean geometry and has given great advice on developing the approaches implemented as part of this project. I would also like to thank Dr Alena Denisova for her help with developing and distributing the survey. Additionally, I’d like to thank Dr Gregory Slabaugh for giving me the opportunity to commence this academic research and guiding me through the start of my PhD research. Thank you to all the staff of the School of Informatics at City University for your continued support and assistance.

I would like to thank my family, who supported my decision to pursue academic research and helped me emotionally and financially during the entire project. I would also like to thank my friends and teammates for the distractions that allowed me to get through the difficult stages of research process.

Declaration

I hereby declare that:

- my submission as a whole is not substantially the same as any that I have previously made or am currently making, whether in published or unpublished form, for a degree, diploma, or similar qualification at any university or similar institution
- the following parts of the work or works now submitted have previously been submitted for a qualification at a university or similar institution (only brief details required):

  . . . .

  . . . .

  . . . .
- until the outcome of the current application to this University is known, the work or works submitted will not be submitted for any qualification at another university or similar institution.

Date: . . . . Signature: . . . .

Print Name: . . . .

ABSTRACT:

This thesis presents an implementation of a 2D non-Euclidean physics and graphics engine using spherical and hyperbolic trigonometry. The engine is capable of working with a 2D space of constant negative or positive curvature. It uses polar coordinates to record the parameters of the objects as well as an azimuthal equidistant projection to render the space onto the screen. A polar coordinate system works well with trigonometric calculations, due to the distance from the reference point (analogous to origin in Cartesian coordinates) being one of the coordinates by definition. Azimuthal equidistant projection is not a typical projection, used for neither spherical nor hyperbolic space, however one of the main features of the engine relies on it: changing the curvature of the world in real-time.

Any 2D shape can be created and used in the engine, not a pre-determined list of standard shapes. Shapes can be moved around the curved space via user input controls.

This thesis describes approaches to improve performance of the engine by analysing and subsequently attempting to reduce the time-complexity of the algorithm as well as parallelizing the calculations by performing them on a GPU in order to avoid a major bottleneck. Empirical tests were performed and it was found that different approaches have an impact on overall engine performance, but the improvement is negligible compared to that gained by parallelisation.

A method for texturing shapes in non-Euclidean 2D space in real-time using spherical and hyperbolic trigonometry is introduced. Stress test results show that the engine can render high load scenes in real-time.

This thesis presents survey results showing participants' generally positive feedback upon playing through two different classic games modified to work within the non-Euclidean engine.

Overall, the project has been successful in developing a novel method of rendering non-Euclidean geometry in real-time using Spherical and Hyperbolic trigonometry; implementing it within a framework which allows the creation of custom environment; and gauging the interest in non-Euclidean games.

1. Introduction

The field of non-Euclidean geometry encompasses any geometry that arises from either changing the parallel postulate (Euclid's fifth postulate) or the metric requirement. This thesis will be focusing solely on traditional non-Euclidean 2D geometries: Spherical geometry and Hyperbolic geometry, illustrated on Figure 1.1 (a) and (c) respectively.

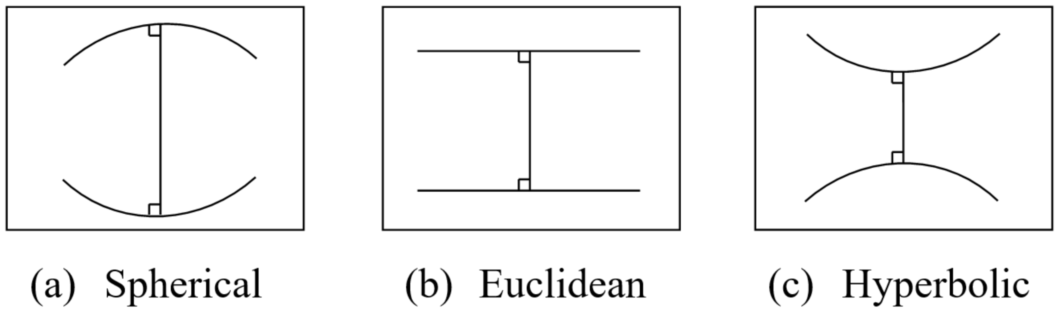


Figure 1.1: *Comparison of parallel lines in the 2D spaces of different curvature*

In spherical geometry all geodesics (straight lines in a non-planar space) intersect, so there are no parallel lines. Even if the lines start parallel, they don't preserve the same distance along their length and instead appear to 'bend' towards each other. In fact, any two great circles will intersect twice (unless they are one and the same). Spherical geometry is used in multiple fields: navigation, GPS, architecture and aerospace engineering among others.

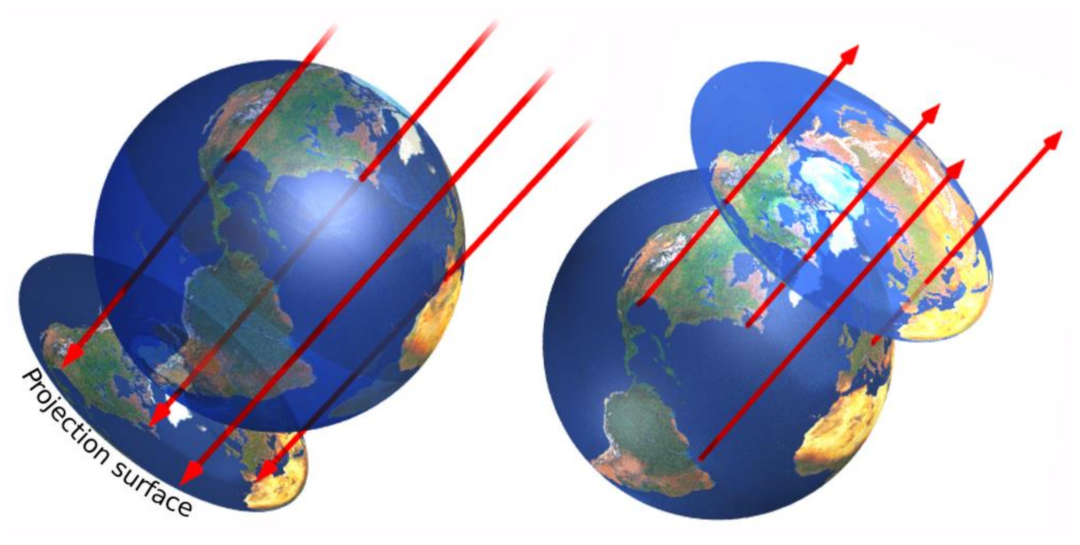


Figure 1.2: Orthographic projection of the sphere onto a tangent plane. Parallel rays are cast onto the surface (left) or from the surface (right) of a sphere; and the points are transferred onto the points of intersection of these rays with a tangent plane.

However, in most circumstances the calculations are done on a surface of a 3D sphere instead of the 2D spherical plane, and rendering, if any uses the orthographic projection of the sphere. Figure 1.2 (Furuti, 2012), for example, illustrates Earth's northern hemisphere.

In cartography, multiple other projections are used and one of them has been chosen to be a focus of this study: Azimuthal equidistant projection. A detailed explanation of this projection and its advantages for this model is described in the Method section.

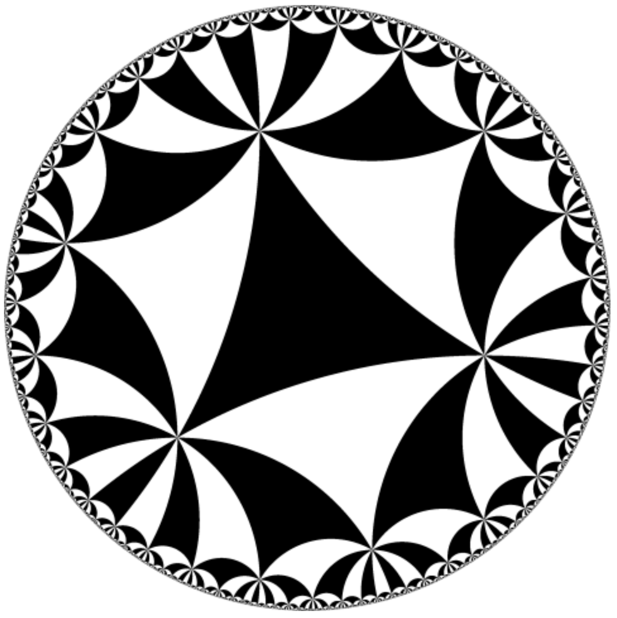


Figure 1.3: Poincaré disc with hyperbolic equilateral triangle tiling. Straight lines in this projection are represented as arcs of circles perpendicular to the disc's boundary. Triangles tiling the plane are identical on the hyperbolic plane, but appear exponentially smaller towards the boundary of the projection disc.

In Hyperbolic geometry, any line can have an infinite number of parallel lines, as the lines appear to bend away from each other. Elliptic geometry is more tangible and intuitive than hyperbolic geometry, due to people interacting with it more and the possibility of a 2D elliptic plane to be embedded into a 3D space. Hyperbolic geometry is more abstract; however, it is used heavily in mathematics, astrophysics and theoretical physics, particularly for calculations involving general relativity (Einstein, 1921). One of the standard projections used to represent hyperbolic geometry is the Poincaré disc (Poincaré, 1881), illustrated in Figure 1.3 (Tamfang, 2011).

1.1 Motivation

Non-Euclidean geometry plays an important role in physics, especially within the context of general relativity (Wald, 2010), where Riemannian geometry is used to calculate the curvature of space-time (Hehl & Kerlick, 1978) and astronomy, where the question of whether the universe has flat curvature of space is still unanswered (Kragh, 2012). However, there are potential applications in other fields, such as games. Implementing non-Euclidean geometry elements within a video game could present interesting challenges to the player. This project aims to create an intuitive and user-friendly method for simulating and rendering curved space.

Subsequently, this method is used to create software that can calculate and render arbitrary shapes in curved space in real-time, while remaining intuitive people using it to be able to create physical worlds with non-Euclidean space. This software is able to define the parameters of the objects, allow for object interactions, create physical environments in a curved space, render curved shapes on screen using a projection and be visually appealing. To make the idea of curved space more intuitive, an additional aspiration was to make it possible for the curvature to be modified in real-time during the execution of the engine.

1.2 Aims & Objectives

Techniques exist for visualising non-Euclidean geometry; however, the aim of this project is to create intuitive and efficient technology for rendering curved space:

- *Create a method for rendering shapes in non-Euclidean geometry:* it should be intuitive for the users and developers.
- *Implement the method to create a rendering engine:* the software should be capable of recording objects' parameters, calculate object transformations and render them on screen.
- *Expand the engine for object movement:* record additional parameters for the objects and simulate the movement of the objects along the geodesics of planar, spherical or hyperbolic space.
- *Optimize performance of the engine:* the software should be able to render complex scenes in real-time to be a viable game engine.
- *Expand the engine to render textured shapes:* create a method for texturing shapes in curved space and implement it within the engine.

- *Evaluate people's opinions towards non-Euclidean elements in games:* use the engine to adapt well-known games to work in non-Euclidean space. Run a survey to learn whether participants like or dislike the idea.

1.3 Contributions

The research makes contributions in the following areas:

- *A method for rendering shapes using spherical and hyperbolic trigonometry is defined:* a model for recording object's parameters is created and used to calculate and render the objects in real-time.
- *The method is expanded to cover object movement:* additional object parameters like speed and acceleration are recorded and the object's movement following the forward vector geodesic is calculated.
- *A method to parallelise the calculations for rendering shapes in curved space is defined:* OpenGL Shading Language (GLSL) is used to perform calculations on the GPU to improve the performance of the engine.
- *A method for texturing objects using spherical and hyperbolic trigonometry is defined:* the model above is expanded to record and subsequently calculate object's texture coordinates in real-time.
- *Software capable of rendering non-Euclidean 2D geometry is created based on the defined methods:* The latest version of the software is available on GitHub (Osudin, GitHub.com, 2022).

Experimentation is performed to:

- Evaluate the performance of the designed non-Euclidean engine as well as alternative tessellation approaches. Stress testing is used see whether the engine is fit to perform in real-time.
- Evaluate the fitness of the engine to perform in real-time simulations.
- Compare the CPU and GPU based versions of the engine.
- Create games in non-Euclidean environment. Two classical games are adapted to work in curved space using the created software.

A survey is conducted to research:

- The intuitiveness of games in non-Euclidean environment.
- The enthusiasm of participants towards playing games in curved space.

1.4 Structure

Chapter 2 details the background for the field of non-Euclidean geometry, from roots and early developments to the current applications and developments in its visualisation.

Chapter 3 describes the structure of the developed engine, including the overall framework and the non-Euclidean specific sections; this chapter also covers initial developments and ideas for the 'field of view' approach, which has been ultimately rejected in favour of the trigonometry-based method.

Chapter 4 explains the method for rendering vector graphics using spherical and hyperbolic trigonometry. This approach is split into three subsections, finding the global coordinates of each vertex of an object; finding preliminaries for edge tessellation; and finding the intermediate points along each edge of the object.

Chapter 5 expands the method to cover the object movement within curved space. Given the object's current position as well as acceleration, velocity, rotation and torque information, its position in the next frame is calculated.

Chapter 6 explores alternative line equation-based approaches for rendering shapes. The three methods explored include: Great Circle Navigation, Orthogonal Vectors and Poincaré disc approaches.

Chapter 7 describes the method for improving the engine's performance using parallelisation of the algorithm described in Chapter 4 using OpenGL Shading Language (GLSL). It details the implementation using pseudocode of each of the shaders.

Chapter 8 describes the testing used to compare the performance of different approaches and provides the analysis of the results.

Chapter 9 explains the method for texturing the shapes in curved space using spherical and hyperbolic trigonometry. This method splits a shape into sections, which are then subsequently subdivided into a series of parallel lines used to find the intermediate points throughout the shape.

Chapter 10 describes testing performed to compare the line and texture rendering approaches; it also provides the analysis of the results of said testing.

Chapter 11 discusses the survey designed to find out people's experience playing classical games in non-Euclidean space.

Chapter 12 describes the achievements of the engine and potential future improvements.

1.5 Publications

Published:

- "Rendering Non-Euclidean Geometry in Real-Time Using Spherical and Hyperbolic Trigonometry" (Osudin, Child, & He, Rendering non-euclidean space in real-time using spherical and hyperbolic trigonometry, 2019)
    - Presented at International Conference on Computational Science 2019 (ICCS 2019) in Faro, Portugal.
    - Described in chapters 4 and 5.

In preparation for publication:

- "Parallelisation of the non-Euclidean Engine for Rendering Textures in Curved Space"
    - Described in chapters 7, 8, 9 and 10.
    - Part of this article has been submitted to Computer Graphics forum under the title of "Optimisation and Parallelisation of a non-Euclidean Geometry Rendering Engine", which only focussed on improving the performance of the engine, covered in chapters 6, 7 and 8. The rejection cited the article not having enough new material.
    - It has been decided to restructure it to cover the texture rendering as a consequence of performance improvement achieved via parallelisation.
    - Preparation for resubmission to Computer Graphics Forum
- "Impact of the non-Euclidean environment on game development and gameplay"
    - Described in chapter 11. Additional statistical analysis will be performed and described in the article.
    - Preparation for submission to the 18th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-22).

1.6 Terminology

- **Gaussian curvature** – Gaussian curvature, subsequently denoted as $\boldsymbol{K}$**,** sometimes also called total curvature, is an intrinsic property of a space independent of the coordinate system used to describe it (Kreyszig, 1991). For a point $\boldsymbol{p}$ on a 2D surface, $\boldsymbol{K}$ is found by multiplying the largest and smallest (principal) curvatures at point $\boldsymbol{p}$. For example, both the largest and the smallest curvature at point $\boldsymbol{p}$ on the surface of a sphere is $\mathbf{1}$, so $\boldsymbol{K} = \mathbf{1} \times \mathbf{1} = \mathbf{1}$. However, for a cylinder the smallest curvature is $\mathbf{0}$, while the largest curvature is $\mathbf{1}$, so $\boldsymbol{K} = \mathbf{0} \times \mathbf{1} = \mathbf{0}$.

- **Conformal projection / conformal model** – A map projection / model which is a conformal mapping, i.e., one for which local (infinitesimal) angles on a surface are mapped to the same angles in the projection. On maps of an entire sphere, however, there are usually singular points at which local angles are distorted (Weisstein, 2002).

- **Azimuthal projection** – A map projection in which a globe, as of the Earth, is assumed to rest on a flat surface onto which its features are projected. An azimuthal projection produces a circular map with a chosen point – the point on the globe that is tangent to the flat surface – at its centre. When the central point is either of Earth's poles (as shown in Figure 1.5 and Figure 1.6), parallels appear as concentric circles on the map and meridians as straight lines radiating from the centre (The American Heritage Science Dictionary, 2016).

- **Azimuthal Equidistant Projection –** An azimuthal map projection of the surface of the earth so cantered at any given point that a straight line radiating from the centre to any other point represents the shortest distance and can be measured to scale (Merriam-Webster.com Dictionary, 2009).

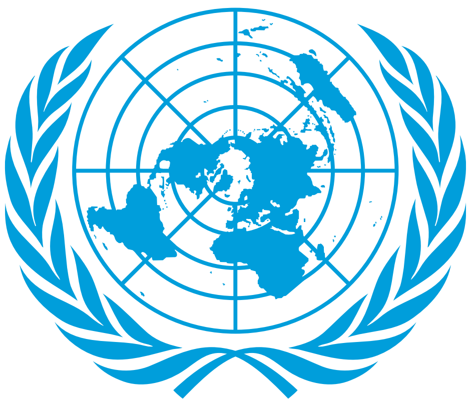


Figure 1.4: Azimuthal equidistant projection used in the United Nations Emblem (United Nations, 2021).

- **Stereographic projection** – a map projection of a hemisphere showing the earth's lines of latitude and longitude projected onto a tangent plane by radials from a point on the surface of the sphere opposite to the point of tangency (Merriam-Webster.com Dictionary, 2015). Illustrated in Figure 1.5.

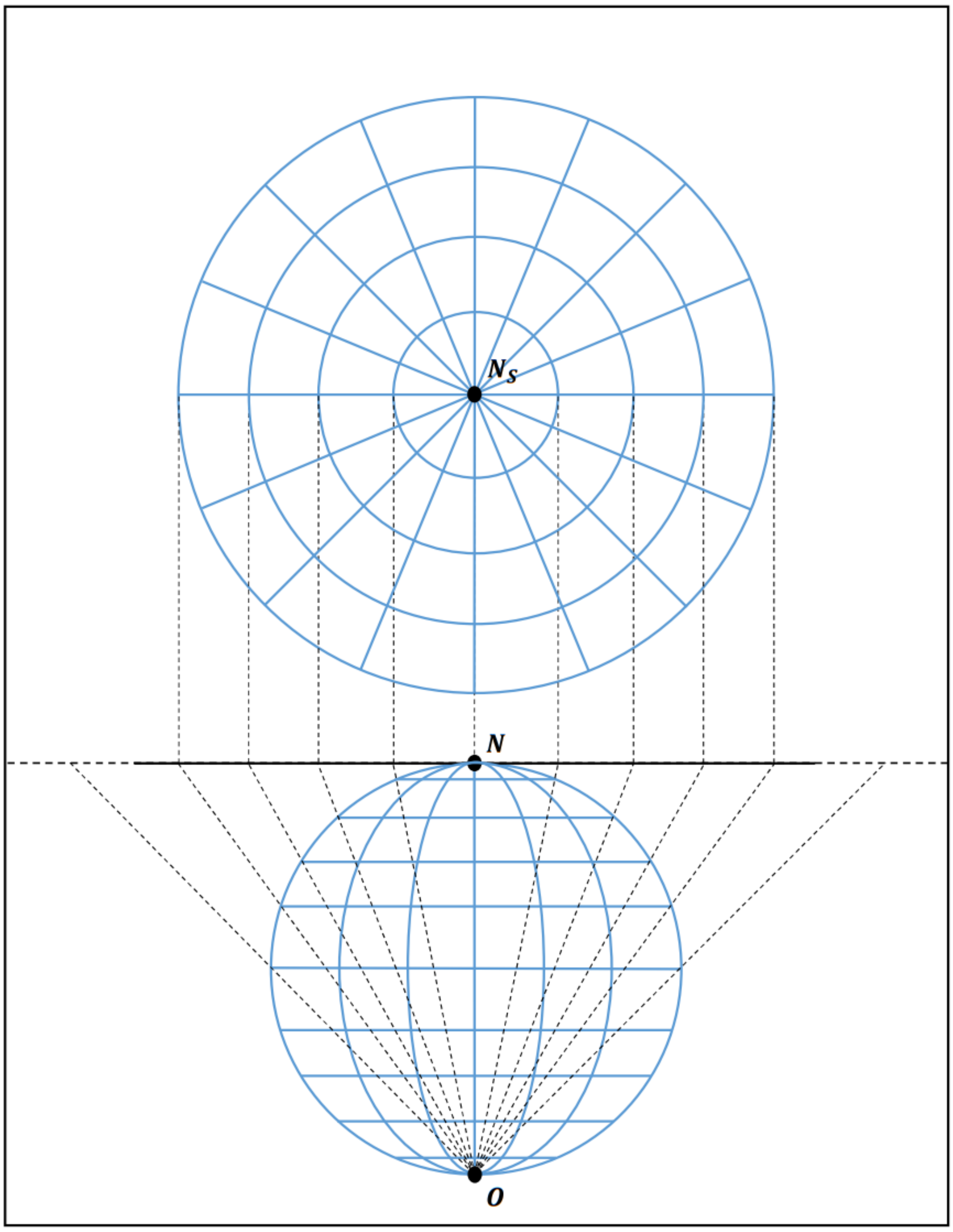


Figure 1.5: Stereographic projection of a globe onto a tangent plane. The globe is tangent to the plain at the North Pole, $\boldsymbol{N}$, so it is chosen as the central point of the projection,$\boldsymbol{N_S}$. The focal point of the projection, $\boldsymbol{O}$, is located at the South Pole.

- **Gnomonic projection** – an azimuthal projection of a part of a hemisphere showing the earth's grid as projected by radials from a point at the centre of the sphere onto a tangent plane so that all straight lines represent arcs of great circles (Merriam-Webster.com Dictionary, 2011). Illustrated in Figure 1.6.

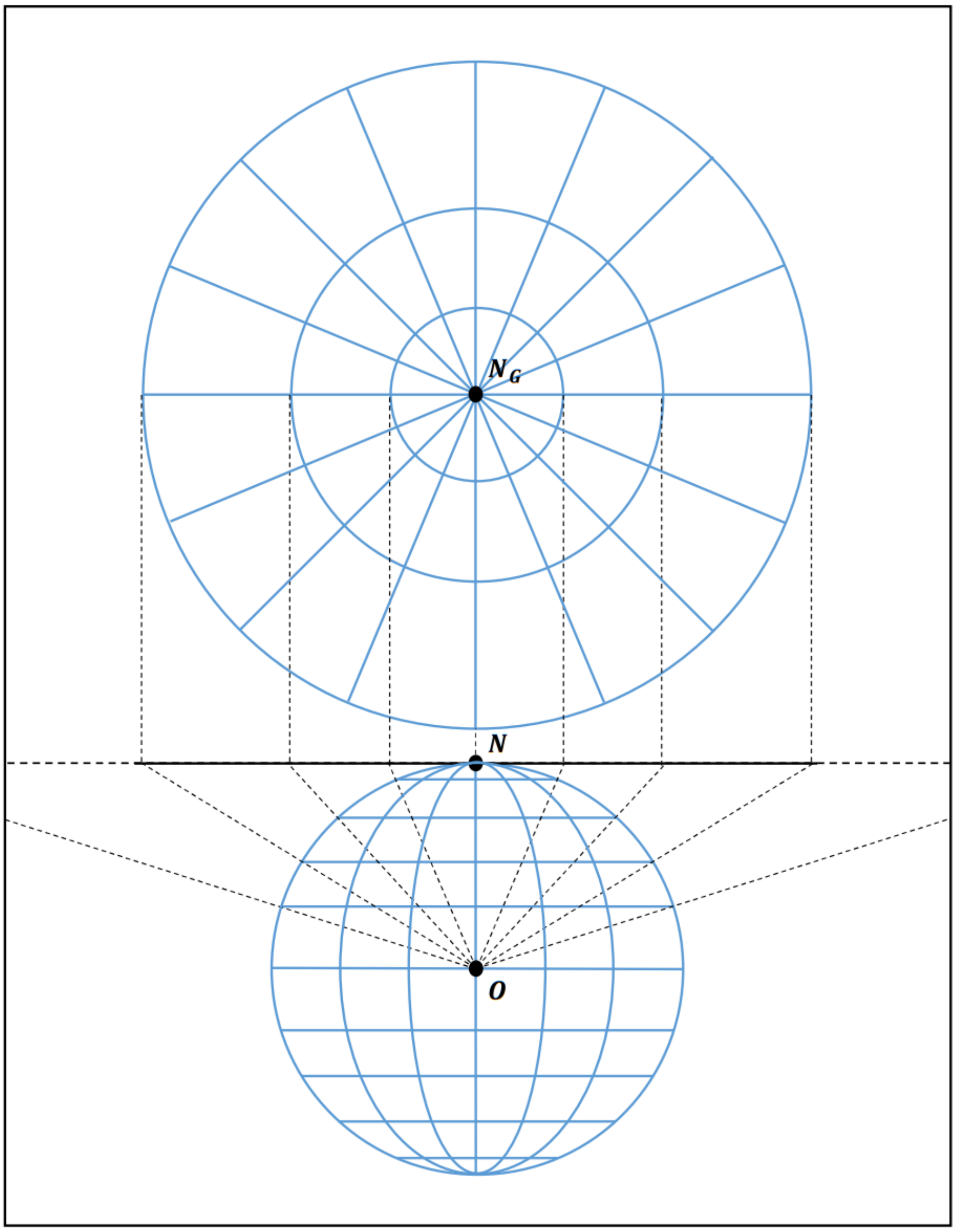


Figure 1.6: Gnomonic projection of a globe onto a tangent plane. The globe is tangent to the plain at the North Pole, $\boldsymbol{N}$, so it is chosen as the central point of the projection,$\boldsymbol{N_G}$. The focal point of the projection, $\boldsymbol{O}$, is located at the centre of the globe.

- **Projective geometry** – projective geometry, branch of mathematics that deals with the relationships between geometric figures and the images, or mappings, that result from projecting them onto another surface. Common examples of projections are the shadows cast by opaque objects and motion pictures displayed on a screen (Artmann, 2018).

- **Manifold** – A manifold is a topological space that is locally Euclidean (i.e., around every point, there is a neighbourhood that is topologically the same as the open unit ball in $\mathbb{R}^{\boldsymbol{n}}$). To illustrate this idea, consider the ancient belief that the Earth was flat as contrasted with the modern evidence that it is round. The discrepancy arises essentially from the fact that on the small scales that we see, the Earth does indeed look flat. In general, any object that is nearly "flat" on small scales is a manifold, and so manifolds constitute a generalization of objects we could live on in which we would encounter the round/flat Earth problem.

  One of the goals of topology is to find ways of distinguishing manifolds. For instance, a circle is topologically the same as any closed loop, no matter how different these two manifolds may appear. Similarly, the surface of a coffee mug with a handle is topologically the same as the surface of the donut, and this type of surface is called a torus (Rowland, 2000).

- **Orientable manifold** – A manifold is said to be orientable if it can be given an orientation. Note the distinction between an "orientable manifold" and an "oriented manifold," where the former implies the possibility of giving the manifold in question an orientation, while the latter implies that the manifold has already been given an orientation (Hedegaard, 2004).

- **Tessellation** – Tessellation is a process that reads a patch primitive and generates new primitives used by subsequent pipeline stages. The generated primitives are formed by subdividing a single triangle or quad primitive according to fixed or shader-computed levels of detail (i.e., Tessellation Variable) and transforming each of the vertices produced during this subdivision (Khronos group, 2010).

- **Barycentric coordinate system** – Barycentric coordinates can be used to express the position of any point located on the triangle with three scalars. The location of this point includes any position inside the triangle, any position on any of the three edges of the triangles, or any one of the three triangle's vertices themselves. Let three real numbers $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{w}$ be the normalized barycentric coordinates of a point $\boldsymbol{p}$, such that $\boldsymbol{u}+\boldsymbol{v}+\boldsymbol{w}=\mathbf{1}$. Then the vertices of the triangle are given by $(\mathbf{1},\mathbf{0},\mathbf{0})$, $(\mathbf{0},\mathbf{1},\mathbf{0})$, and $(\mathbf{0},\mathbf{0},\mathbf{1})$ (Scratchapixel, 2015).

2. Literature Review: History and Applications of non-Euclidean Geometry

This chapter contains a review of the existing research in the field of non-Euclidean geometry, from its roots to current developments in its applications and visualisation.

2.1 Early Developments

Non-Euclidean geometry takes its origin from Euclid's work Elements, where he defined his five postulates (translated by Thomas Heath) (Heath, 1956):

"Let the following be postulated:

1. To draw a straight line from any point to any point.
2. To produce (extend) a finite straight line continuously in a straight line.
3. To describe a circle with any centre and distance (radius).
4. That all <u>right angles</u> are equal to one another.
5. That, if a straight line falling on two straight lines make the interior angles on the same side less than two right angles, the two straight lines, if produced indefinitely, meet on that side on which the angles are less than two right angles."

Euclid has developed the axiomatic method and described Euclidean geometry as an axiomatic system, a set of fundamental statements (axioms) which can be used to derive all other propositions (theorems) in a consistent and relatively self-contained body of knowledge (theory) (Novikov, 2001).

When compared to the other postulates, the 5th postulate is less intuitive and not immediately obvious. Euclid's postulates 1 through 4 were thought to be more fundamental than the fifth postulate, so mathematicians have tried to prove the fifth postulate by contradiction or replace it with a more fundamental one. Proving the parallel postulate given the others would mean it is a theorem and not an axiom. These attempts are detailed by M. Eder (Eder, 2000) and M. J. Greenberg (Greenberg, 2008). Over the centuries many Mathematicians have attempted this: Proclus (A.D. 410 – 485), Nasir Eddin al-Tusi (1201 – 1274), John Wallis (1616 – 1703), Girolamo Saccheri (1667 – 1733), Johann Heinrich Lambert (1728 – 1777) among others.

Every attempt to prove the parallel postulate had at least one unjustified statement, which made the proof flawed. Each proposed axiom to replace the parallel postulate can be shown to be equivalent to it.

For example, a Greek mathematician Proclus attempted to replace the parallel postulate with the following axiom:

"If a line intersects one of two parallel lines, both of which are coplanar with the original line, then it must intersect the other also." (Weisstein, Proclus' Axiom – MathWorld, 2000)

M. Eder (Eder, 2000) describes this attempt by first re-phrasing the axiom given by Proclus:

"Given $\alpha + \beta < 2d$, prove that the straight lines g' and g'' meet at a certain point C."

Where, d, is a magnitude of a right angle. An illustration of this proof is shown in Figure 2.1.

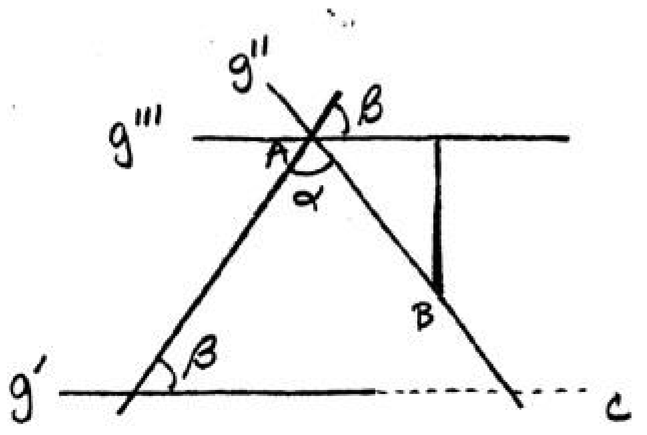


Figure 2.1: Diagram from the article by M. Eder. Parallel lines g' and g''; α and β, interior angles between g', g'' and a line intersecting both lines. $g' \parallel g'''$; $B \in g''$; $C \in g'$; $A \in g''$; $A \in g'''$.

Proclus begins the proof by drawing a straight line, g''' parallel to g' through the intersection of g'' and g''', point A. Then he marks a point, B, on g'' and traces a line from point B to line g''' perpendicular to it. From this, the proof shows that as distance between A and B increases, the distance from B to g''' also grows without limit. And as the distance between g' and g''' is constant, there must be a point C, where g' and g'' meet.

This proof relies on a notion of straight parallel lines, which is not explicitly defined by the proof itself or the four other Euclid's postulates. Ultimately, Proclus' axiom is recognised to be identical to Euclid's parallel postulate.

2.2 Hyperbolic Geometry

Although these pursuits were unsuccessful, they have paved the way for mathematicians of the 19th century to develop non-Euclidean geometry (Bonola, 1912).

Janos Bolyai (1802 – 1860), Carl Friedrich Gauss (1777 – 1855) and Nikolai Ivanovich Lobachevsky (1792 – 1856) have independently discovered and developed Hyperbolic geometry by negating the fifth postulate. They found that a consistent geometry is described by assuming a negation of Euclid's 5th postulate as an axiom. (Halsted, 1900) (Gray, 2006) (Gray, János Bolyai, non-Euclidean geometry, and the nature of space, 2004) (Rodin, 2015) (Petrunin, 2019).

Gauss wrote in his letter to F. A. Taurinus (Gauss, 1824):

"...The assumption that the sum of the three angles is less than 180° leads to a curious geometry, quite different from ours [the Euclidean], but thoroughly consistent, which I have developed to my entire satisfaction, so that I can solve every problem in it with the exception of the determination of a constant, which cannot be designated *a priori*. The greater one takes this constant, the nearer one comes to Euclidean geometry, and when it is chosen infinitely large the two coincide. The theorems of this geometry appear to be paradoxical and, to the uninitiated, absurd; but calm, steady reflection reveals that they contain nothing at all impossible. For example, the three angles of a triangle become as small as one wishes, if only the sides are taken large enough; yet the area of the triangle can never exceed a definite limit, regardless of how great the sides are taken, nor indeed can it never reach it.

All my efforts to discover a contradiction, an inconsistency, in this non-Euclidean geometry have been without success, and the one thing in it which is opposed to our conceptions is that, if it were true, there must exist in space a linear magnitude, determined for itself (but unknown to us)..."

From this quote, it is clear that Gauss has developed non-Euclidean geometry in an attempt to discover a contradiction, which would prove Euclidean geometry as fundamental. However, instead he found that the geometry is self-consistent and simply alternative to the Euclidean.

C. F. Gauss chose not to publish his works and urged J. Bolyai to do the same in private correspondence on the subject. In 1829 he wrote to F. W. Bessel that he was scared of the "the howl from the Boeotians" upon him publishing his works on non-Euclidean geometry. Such a discovery would contradict the established philosophical ideas at the time, in particular of I. Kant. Because of this the first to publish research in this area was N. I. Lobachevsky in 1829. The approaches Bolyai and Lobachevsky had were almost the same and both have developed the formulas for non-Euclidean trigonometry used later in this thesis.

2.3 Spherical Geometry

The earliest works describing geometry on the surface of a sphere date to Greek mathematicians. In particular, Theodosius of Bithynia's (c. 169 BC – c. 100 BC) work "Sphaerica", which described the geometry of a sphere (Rosenfeld, 2012). Later Al-Jayyani (989 – 1079), Islamic mathematician, wrote "The Book of Unknown Arcs of a Sphere", which described the formulas for spherical trigonometry (Hairetdinova, 1986). Another important work formalising the foundations of Spherical geometry were the memoirs of Leonhard Euler (Papadopoulos, 2014).

While hyperbolic geometry emerges if Euclid's fifth postulate is replaced with the following: "given line l and point p, there exists an infinite number of lines through p parallel to l" (Hoboken, 1994), spherical geometry can be derived by replacing this postulate with "given line l and point p, there exists no line through p parallel to l". In order to achieve a consistent model, other axioms need to be changed as well, depending on which system of axioms is used.

2.4 Visualisation Techniques

The first insights into visualising hyperbolic geometry came from proving the consistency of hyperbolic geometry. Eugenio Beltrami (1835 - 1900) worked on finding a Euclidean model of hyperbolic space, for if such a model was found, hyperbolic geometry could be proven to be consistent if Euclidean geometry is assumed to be consistent. This was later written in a form of a metamathematical theorem:

"If Euclidean geometry is consistent, then so is hyperbolic geometry."

2.4.1 Beltrami-Klein Model

Eugenio Beltrami and Felix Klein (1849 - 1925) independently created a model of hyperbolic geometry based on projective geometry ideas developed by Arthur Cayley (1821 - 1895) (Cayley, 1859). Although Beltrami's memoirs (Beltrami, 1868a) (Beltrami, 1868b) on hyperbolic geometry were published earlier (in 1868), they were not as well recognised as Klein's published method (Klein, 1871).

Beltrami-Klein is a model in n-dimensions, but in two dimensions it is represented as a disk $\boldsymbol{\gamma}$ on the Euclidean plane. The interior of the disk represents the entirety of the hyperbolic plane. The points on the circumference of $\boldsymbol{\gamma}$ are the ideal points and do not belong to the hyperbolic plane.

Segments joining points on the circumference of $\boldsymbol{\gamma}$ without their end points are open chords, which are the lines of the hyperbolic plane in this model.

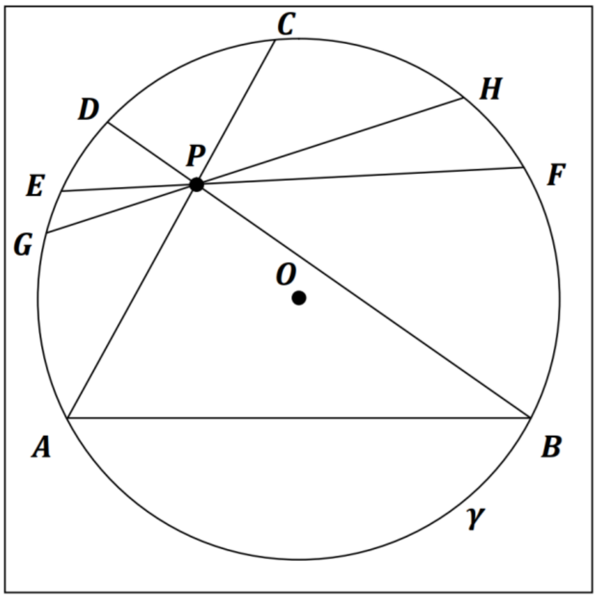


Figure 2.2: Hyperbolic plane in the Beltrami-Klein model. Disk $\boldsymbol{\gamma}$ with an origin $\boldsymbol{O}$; $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, $\boldsymbol{D}$, $\boldsymbol{E}$, $\boldsymbol{F}$, $\boldsymbol{G}$, $\boldsymbol{H}$, ideal points. Hyperbolic lines $\overline{\boldsymbol{AC}}$, $\overline{\boldsymbol{BD}}$, $\overline{\boldsymbol{EF}}$, $\overline{\boldsymbol{GH}}$ through point $\boldsymbol{P}$ are parallel to $\overline{\boldsymbol{AB}}$

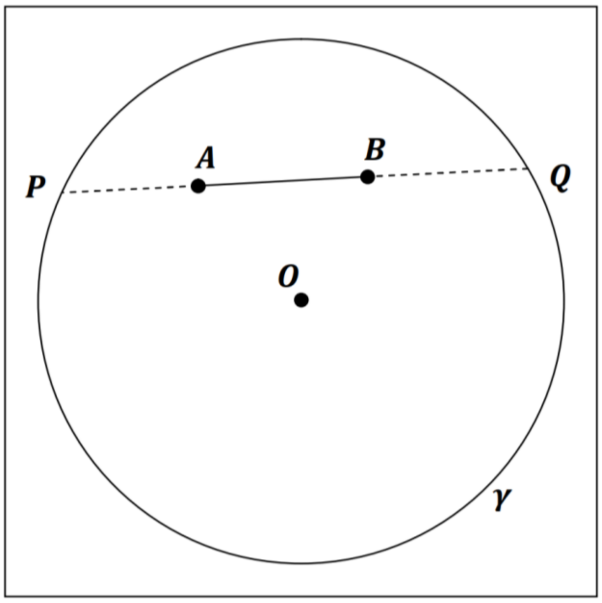


Figure 2.3: Hyperbolic plane in the Beltrami-Klein model. Disk $\boldsymbol{\gamma}$ with an origin $\boldsymbol{O}$; $\boldsymbol{P}$, $\boldsymbol{Q}$, ideal points; $\overline{\boldsymbol{PQ}}$, hyperbolic line; $\boldsymbol{A}, \boldsymbol{B} \subset \overline{\boldsymbol{PQ}}$.

Two lines are said to be parallel in Beltrami-Klein model if they do not intersect in the interior of $\boldsymbol{\gamma}$, even if they meet at a point on the circumference of $\boldsymbol{\gamma}$ (Figure 2.2).

The model is not conformal, meaning that the angle magnitudes are not preserved, when translated to a different point on the Beltrami-Klein model. Trade-off when compared with the conformal models (e.g., Poincare disc model described in section 2.4.2), is that geodesics of the hyperbolic plane are represented by straight line segments inside $\boldsymbol{\gamma}$.

The distances on the hyperbolic plane between two points $\boldsymbol{A}$ and $\boldsymbol{B}$ in this model are given by the Cayley-Klein metric. It requires the endpoints of the extended line joining $\boldsymbol{A}$ and $\boldsymbol{B}$. Let $\boldsymbol{P}$ and $\boldsymbol{Q}$ be ideal points on the circumference of $\boldsymbol{\gamma}$ (Figure 2.3), such that $\overline{\boldsymbol{AB}} \subset \overline{\boldsymbol{PQ}}$ and $|\overline{\boldsymbol{PB}}| > |\overline{\boldsymbol{PA}}|, |\overline{\boldsymbol{AQ}}| > |\overline{\boldsymbol{BQ}}|$, then distance, $\boldsymbol{d}$, is given by:

$$\boldsymbol{d(A,B)} = \frac{\mathbf{1}}{\mathbf{2}}\mathbf{log}\frac{|\overline{\boldsymbol{PB}}||\overline{\boldsymbol{AQ}}|}{|\overline{\boldsymbol{PA}}||\overline{\boldsymbol{BQ}}|} \tag{2.1}$$

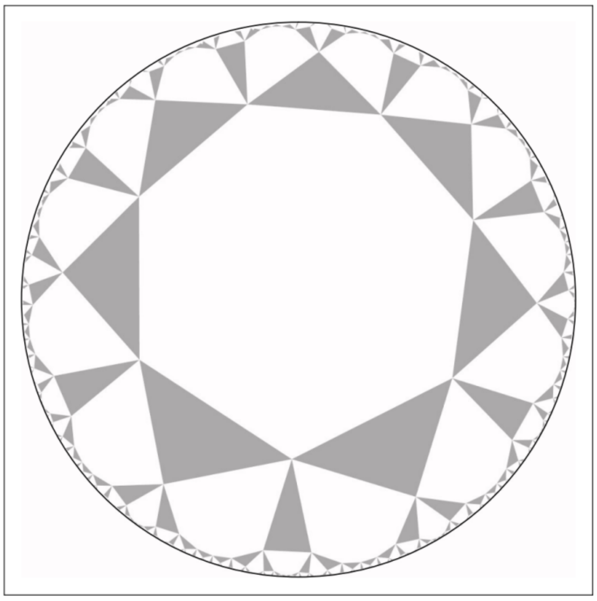


Figure 2.4: Quasiregular $\boldsymbol{r\{7,3\}}$ tiling of the hyperbolic plane in the Beltrami-Klein model. Heptagons and triangles are alternated to tile the plane. Rendered using KaleidoTile software (Weeks, KaleidoTile, 2020).

A popular way to visualise non-Euclidean geometry models is to tile the hyperbolic plane with regular polygons. This produces an image with a pattern which gets increasingly smaller towards the circumference of the rendered disk. One such tiling is illustrated on Figure 2.4. KaleidoTile software has been used to render it by setting the symmetries to (3, 3, 7). It produces a quasiregular tiling, where heptagons and triangles are alternated to tile the hyperbolic plane.

2.4.2 Poincaré Disc Model

A different model of hyperbolic geometry has also been developed by Eugenio Beltrami, however a later rediscovery by Henri Poincaré (1854 -1912) in 1881 has become more widely recognised, hence the model is named after Poincaré (Poincaré, 1881).

In this model the interior of a disk is also used to represent the entirety of the hyperbolic plane. Likewise, the points on the circumference of the disk do not belong to the hyperbolic plane and are ideal points.

The hyperbolic lines in this model are constructed differently from the Beltrami-Klein model. There are two types of lines in this model: open chords and open arcs.

Open chords are only considered hyperbolic lines in this model if they are diameters of $\boldsymbol{\gamma}$. For example, in Figure 2.5 a line $\overline{\boldsymbol{CD}}$ passes through the origin $\boldsymbol{O}$ of the Poincaré Disk $\boldsymbol{\gamma}$, so it is a hyperbolic line. Like in the Beltrami-Klein model, the endpoints of the line (points $\boldsymbol{C}$ and $\boldsymbol{D}$) do not belong to the hyperbolic line.

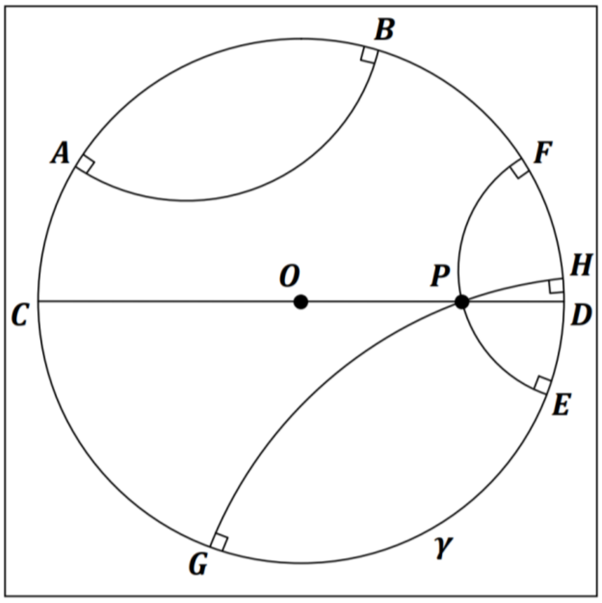


Figure 2.5: Hyperbolic plane in the Poincaré Disk model. Disk $\boldsymbol{\gamma}$ *with an origin* $\boldsymbol{O}$*;* $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, $\boldsymbol{D}$, $\boldsymbol{E}$, $\boldsymbol{F}$, $\boldsymbol{G}$, $\boldsymbol{H}$, *ideal points; hyperbolic lines* $\overline{\boldsymbol{CD}}$, $\overline{\boldsymbol{EF}}$, $\overline{\boldsymbol{GH}}$ *through point* $\boldsymbol{P}$ *are parallel to* $\overline{\boldsymbol{AB}}$**.**

Open arcs are only considered hyperbolic lines in the Poincaré Disk model if they intersect the circumference of $\boldsymbol{\gamma}$ at a right angle. A circle $\boldsymbol{\sigma}$ is orthogonal to $\boldsymbol{\gamma}$ if the radii of these circles meet at a right angle at each point of intersection. The arc of $\boldsymbol{\sigma}$ within the interior of $\boldsymbol{\gamma}$ is a hyperbolic line in this model. For example, in Figure 2.5, open arcs $\overline{\boldsymbol{AB}}$, $\overline{\boldsymbol{EF}}$, $\overline{\boldsymbol{GH}}$ are hyperbolic lines due to intersecting circumference of $\boldsymbol{\gamma}$ at a right angle. Like with open chords, the endpoints of these arcs do not belong to the hyperbolic line.

In the Poincaré Disk model, hyperbolic lines are considered to be parallel if they do not intersect in the interior of $\boldsymbol{\gamma}$. For example, in Figure 2.5, hyperbolic lines $\overline{\boldsymbol{CD}}$, $\overline{\boldsymbol{EF}}$, $\overline{\boldsymbol{GH}}$ through point $\boldsymbol{P}$ are parallel to $\overline{\boldsymbol{AB}}$. Note that hyperbolic lines which meet at an ideal point are also considered parallel.

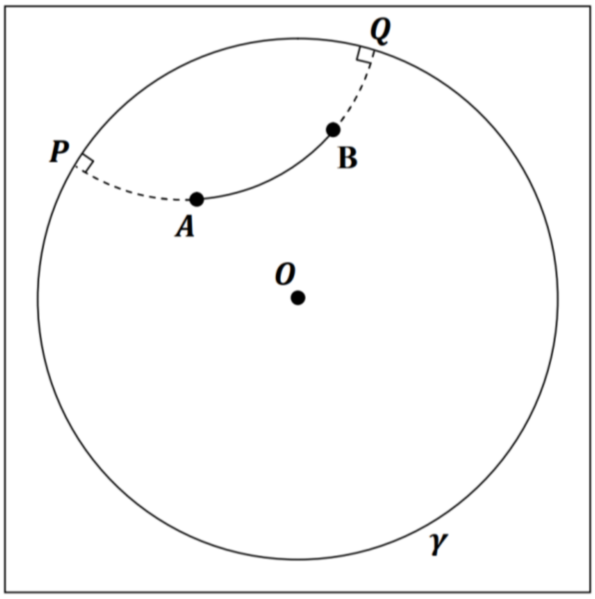


Figure 2.6: Hyperbolic plane in the Poincaré disk model. Disk $\boldsymbol{\gamma}$ with an origin $\boldsymbol{O}$; $\boldsymbol{P}$, $\boldsymbol{Q}$, ideal points; $\overline{\boldsymbol{PQ}}$, hyperbolic line; $\boldsymbol{A}, \boldsymbol{B} \subset \overline{\boldsymbol{PQ}}$.

The Poincaré Disk model is conformal, meaning that angle magnitudes are preserved, but this comes at a cost of hyperbolic lines not appearing straight in the model.

Like in the Beltrami-Klein model, the distances on the hyperbolic plane between two points $\boldsymbol{A}$ and $\boldsymbol{B}$ in Poincaré Disk model are given by the Cayley-Klein metric. It requires the endpoints of the extended arc joining $\boldsymbol{A}$ and $\boldsymbol{B}$. Let $\boldsymbol{P}$ and $\boldsymbol{Q}$ be ideal points on the circumference of $\boldsymbol{\gamma}$ (Figure 2.6), such that $\overline{\boldsymbol{AB}} \subset \overline{\boldsymbol{PQ}}$ and $|\overline{\boldsymbol{PB}}| > |\overline{\boldsymbol{PA}}|, |\overline{\boldsymbol{AQ}}| > |\overline{\boldsymbol{BQ}}|$, then distance, $\boldsymbol{d}$, is given by:

$$\boldsymbol{d}(\boldsymbol{A}, \boldsymbol{B}) = \log \frac{|\overline{\boldsymbol{PB}}||\overline{\boldsymbol{AQ}}|}{|\overline{\boldsymbol{PA}}||\overline{\boldsymbol{BQ}}|} \qquad (2.2)$$

Each point on the Poincare disk can be mapped to a point on a disk in a Beltrami-Klein model. This isometry can be represented by the following relationships between coordinates in the respective models.

A point $\boldsymbol{A}$ in the Poincaré disk model maps to a point $\boldsymbol{A'}$ in the Beltrami-Klein model:

$$\boldsymbol{A(x,y) = A'\left(\frac{2x}{1+x^2+y^2}, \frac{2y}{1+x^2+y^2}\right)} \qquad (2.3)$$

A point $\boldsymbol{A}$ in the Beltrami-Klein model maps to a point $\boldsymbol{A'}$ in the Poincaré disk model:

$$\boldsymbol{A(x,y) = A'\left(\frac{x}{1+\sqrt{1-x^2-y^2}}, \frac{y}{1+\sqrt{1-x^2-y^2}}\right)} \qquad (2.4)$$

A Hyperbolic plane in the Poincaré disk can also be tiled with regular polygons. Like with Beltrami-Klein model, this produces an image with a pattern which gets increasingly smaller towards the circumference of the rendered disk. A quasiregular tiling of alternating heptagons and triangles is rendered on Figure 2.7. It is a rendering of the same tiling as in Figure 2.4. KaleidoTile software has been used to render it by setting the symmetries to (3, 3, 7), where the second and third parameters determine the number of sides of the alternating regular polehedra (in this case regular triangle and regular heptagon); and the first parameter determines the number of shapes of each type that touch at each vertex (in this case there are 3 triangles and 3 heptagons meeting at each vertex).

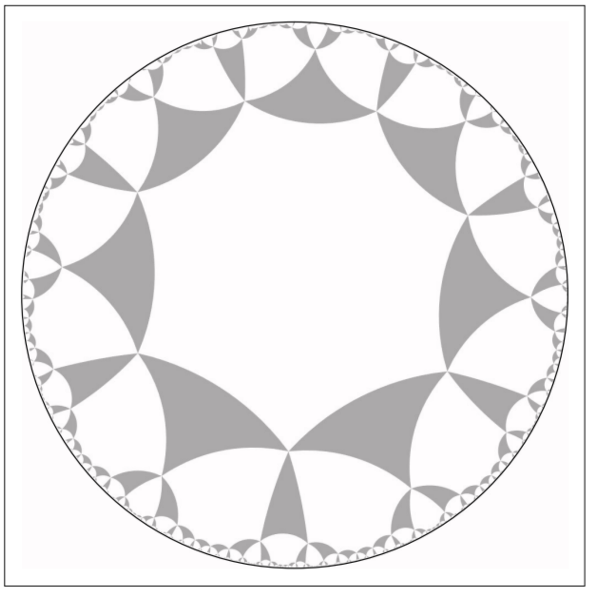


Figure 2.7: Quasiregular $\boldsymbol{(3,3,7)}$ *tiling of the hyperbolic plane in the Poincaré Disc model. Heptagons and triangles are alternated to tile the plane. Rendered using KaleidoTile software (Weeks, KaleidoTile, 2020).*

2.4.3 Visualisation Mediums

Visualisation of non-Euclidean geometry has been developed more extensively with the invention of personal computers; however creative visualisations of the hyperbolic plane have preceded it. The most famous examples of creative visualisation are wood engravings by Maurits Cornelis Escher (1898 - 1972). Circle limit series is a precise and creative tiling of the hyperbolic plane using Poincaré disk model. 'Circle Limit III', shown on Figure 2.8 has been said to be the most sophisticated of Escher's mathematics inspired works (Coxeter H. S., 1979).

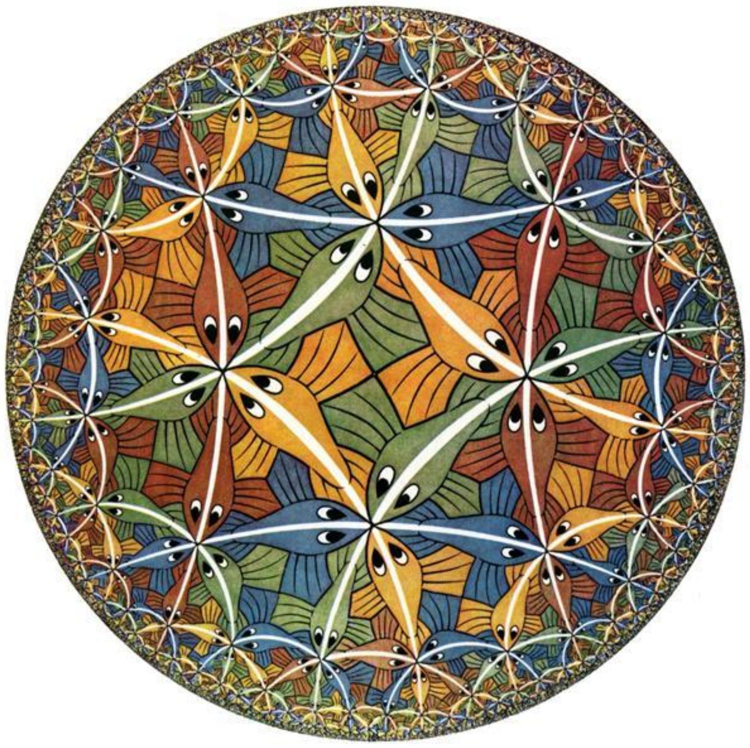


Figure 2.8: Circle Limit III, (Escher, 1959).

There are visualisations of hyperbolic plane in other mediums, for example Daina Taimiņa invented a method of crocheting a model of hyperbolic plane as a teaching tool in 1997. She later described the method in her book on the subject (Taimina, 2018). The resulting crochets have the same structure as some coral reefs, shown on Figure 2.9. In 2005, project Crochet Coral Reef has been created by Christine Wertheim and Margaret Wertheim and the Institute for Figuring (Marzec, 2010).

"The *Crochet Coral Reef* is an artwork responding to climate change, an exercise in applied mathematics, and a wooly experiment in evolutionary theory."

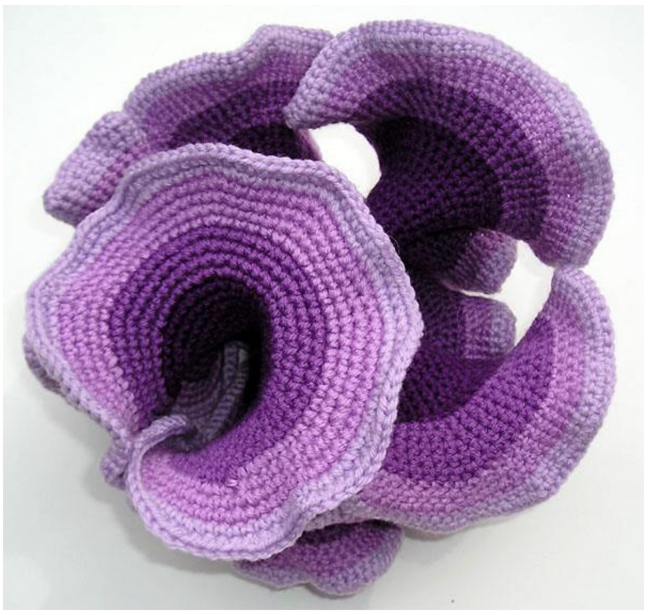


Figure 2.9: Crochet of the hyperbolic plane by Daina Taimiņa.

2.4.4 Visualisation Software

With the increase in computational power of personal computers, it became possible to simulate a visualisation of non-Euclidean space. The Geometry Center was founded to research geometric visualisation by, among others, William Thurston (Mervis, 2002). In 1993, GeomView, software capable of rendering curved 3D spaces was created by the researchers at Geometry Center (Amenta, Levy, Munzer, & Phillips, 1995). GeomView was innovative in two areas: "interactive exploration of curved spaces, and of topological manifolds modelled on these spaces".

GeomView uses projective geometry (Coxeter H. S., 2003) and homogenous coordinates to render non-Euclidean geometry (Gunn, 1993). An example is given in two dimensions, but the method is generalized for higher dimensions. The method derives distance functions, with the same generalised structure for Euclidean, Spherical and Hyperbolic geometries.

A more detailed discussion of the use of 4×4 matrices to render hyperbolic geometry is discussed in the 1992 study by the Geometry Center (Phillips & Gunn, 1992). In it the authors present formulas for computing reflections, translations, and rotations in hyperbolic space.

The Geometry Center has also created videos of hyperbolic geometry rendering, the most renowned of them is called "Not Knot" (Gunn, Epstein, & Maxwell, 1991). One of the renderings in the video shows the tiling of a hyperbolic space by regular dodecahedra (illustrated on Figure 2.10).

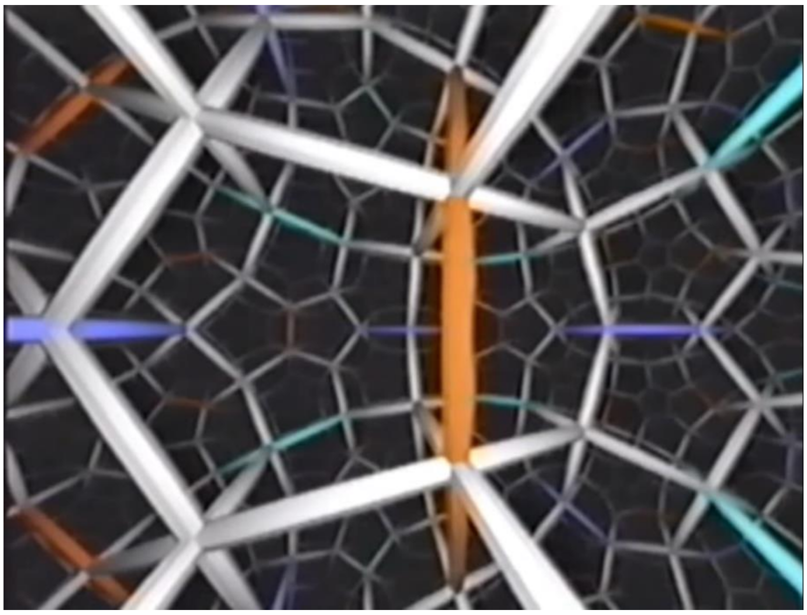


Figure 2.10: 3D hyperbolic space tiling with regular dodecahedra. Frame taken from a video "Not Knot" by the Geometry Center (Gunn, Epstein, & Maxwell, 1991).

A method for use of matrices to render non-Euclidean geometry has been detailed by Jeff Weeks in his article on real-time rendering in curved spaces (Weeks, 2002). In this article the rendering pipelines of Euclidean, Spherical and Hyperbolic spaces are compared; similarities and differences between them are explained and examples of non-Euclidean spaces are illustrated. Figure 2.11, Figure 2.12 and Figure 2.13, taken from the article by Jeff Weeks show the differences between rendering pipelines of 3D Euclidean, Spherical and Hyperbolic spaces.

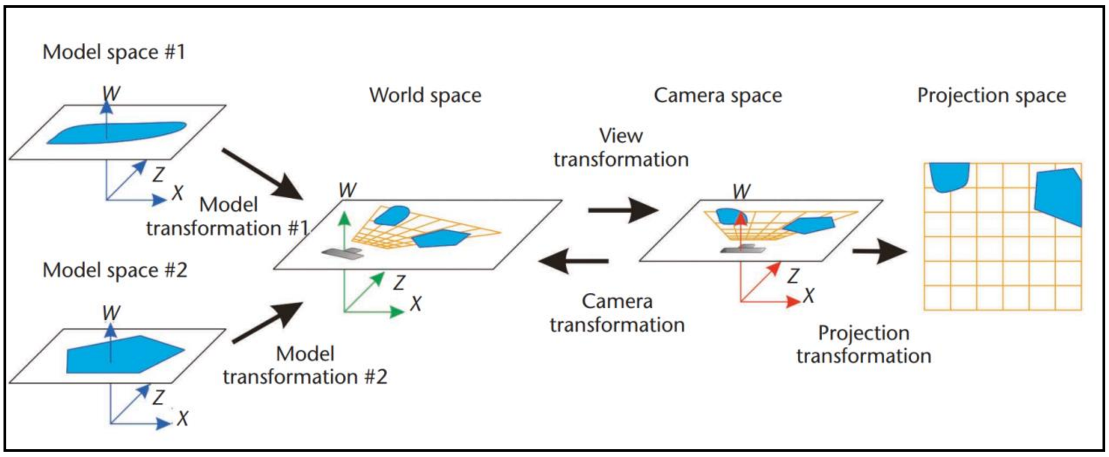


Figure 2.11: Four coordinate systems of standard rendering pipeline for 3D graphics, represented as a 3D hyperplane in 4D space and transformations connecting them represented as matrices (Weeks, 2002).

The four different coordinate systems of the standard rendering pipeline are illustrated in Figure 2.11, these are: Model Space, the local coordinate system for each of the objects (with the object usually centred on the origin of the local coordinate system); World space, the global coordinate system within the simulation's environment; Camera space, global coordinate system with the camera as the origin; and Projection space, the view from the camera's position 'inside the simulation'. Representing each as a separate space is convenient for visualising the transformations between these coordinate systems. Model transformation includes translation, rotation and scaling of the model to fit the scene. The view transformation changes the global coordinate system such that the camera is in a standardised position. The camera transformation is an inverse of the view transformation. The projection transformation fits what the camera sees in the world onto the flat plane, ready to be rendered on the screen.

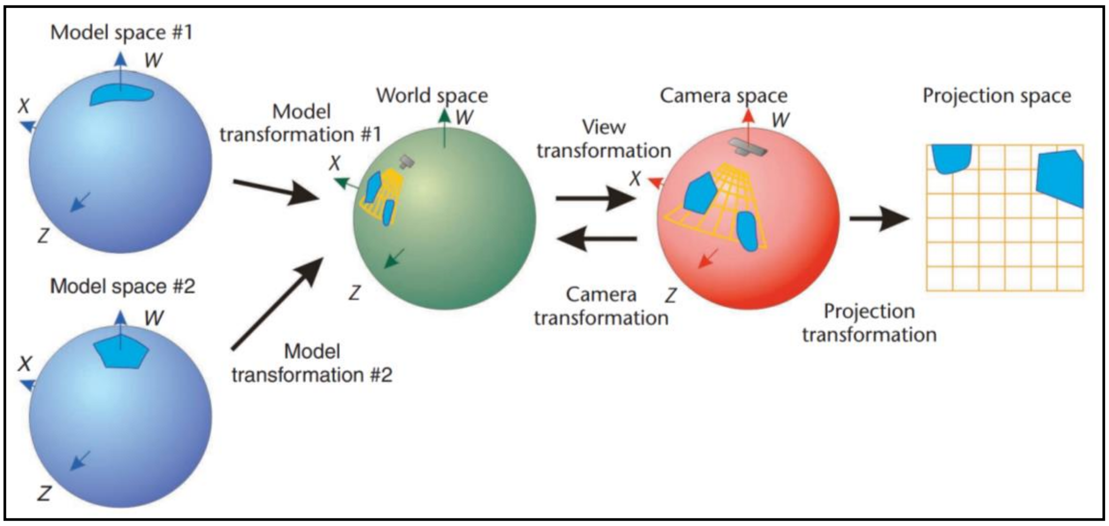


Figure 2.12: The rendering pipeline for a hypersphere (Weeks, 2002).

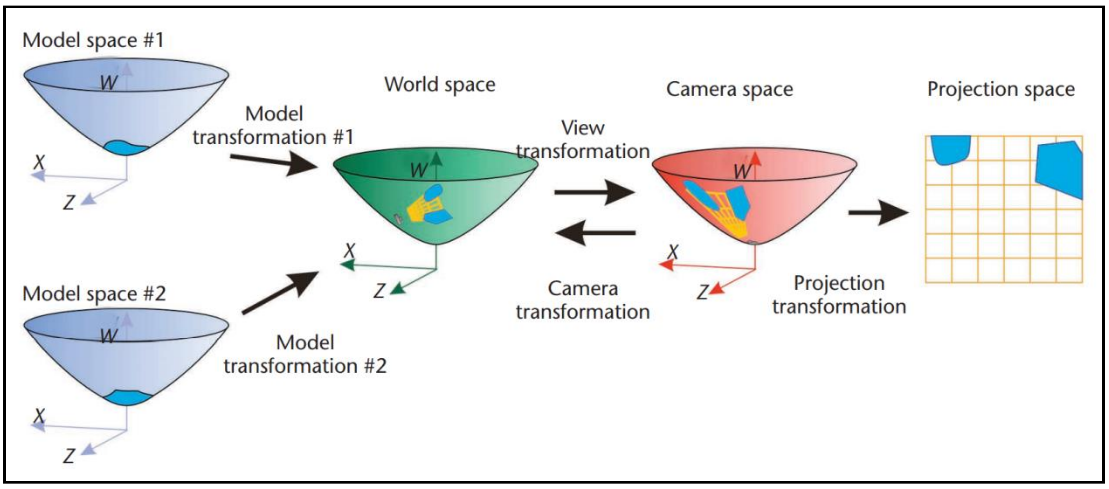


Figure 2.13: The rendering pipeline for a hyperbolic plane (Weeks, 2002).

The rendering pipeline of a positively curved space (illustrated in Figure 2.12) and negatively curved space (illustrated in Figure 2.13) look similar to the standard Euclidean pipeline, but with a few key differences. For example, when rendering on a hypersphere, the model, view and camera transformations are rotations. When rendering on a hyperbolic plane, these are Lorentz transformations instead.

Another example of software designed to render non-Euclidean geometry is jReality:

"A Java library for creating real-time interactive audiovisual applications with three-dimensional computer graphics and spatialized audio." (Weißmann, Gunn, Brinkmann, Hoffmann, & Pinkall, 2009)

The research into metric-neutral visualisation has begun using GeomView system, but jReality allowed researchers to explore these ideas further (Gunn, 2010). Metric-neutral is software which treats Euclidean and classical non-Euclidean spaces (i.e. elliptic and hyperbolic) equally. This is done via the projective geometry described above. This article describes methods for metric-neutral non-interactive issues, interactivity and immersive visualisation; it also explains innovative methods for metric-neutral tracking tubing and realtime shading.

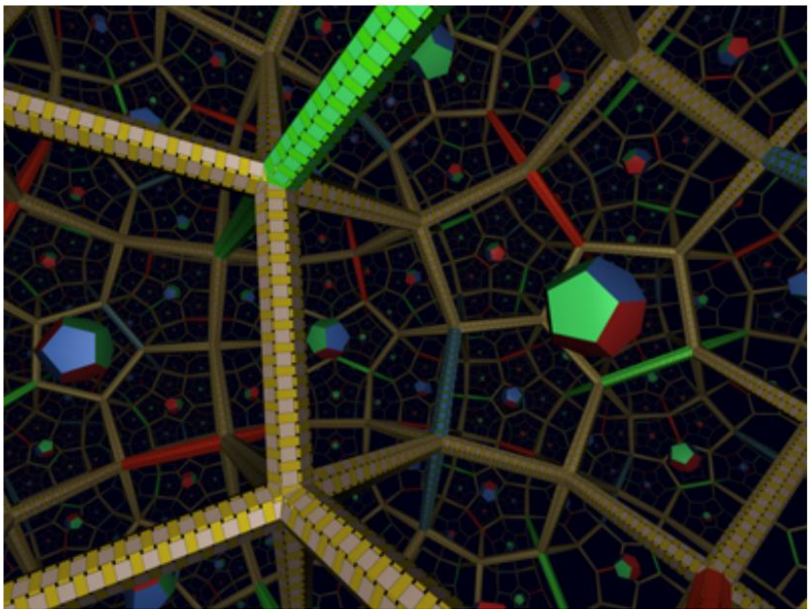


Figure 2.14: rendering of 3D hyperbolic space using jReality by Charles Gunn. It shows a tessellation of hyperbolic space by regular right-angled dodecahedra. Smaller dodecahedra are placed in the local centre of each larger dodecahedron (Gunn, 2010).

This research focuses on "insider's view" visualisation, representing the view that an observer embedded within the said geometry would see. Gunn states that it is a better way of visualising curved space as opposed to conformal models, like Poincaré disk, which visualise the space, but from an outsider’s perspective. An example of such a visualisation is illustrated in Figure 2.14.

This research discusses ways to solve issues with immersive environments in non-Euclidean spaces. These issues include: metric-neutral tracking, unit lengths and scaling problem.

Virtual reality technology can trace its roots to the research by Charles Wheatstone which first described binocular vision (Wheatstone, 1838). The first VR machine was created in 1957 by a cinematographer Morton Heilig (Brockwell, 2016). This technology has been used to create immersive training simulators and display 3D movies, but in the 1990s researchers attempted to create the first non-Euclidean immersive environments (Hudson, Gunn, Francis, Sandin, & DeFanti, 1995). An example of a VR environment they created is shown on Figure 2.15. It is a tessellation of a 3D spherical space with regular dodecahedrons. Later George K. Francis collaborated with enclosed virtual reality theatre researchers to visualize the three-dimensional curved geometries (Francis, Goudeseune, Kaczmarski, Schaeffer, & Sullivan, 2003).

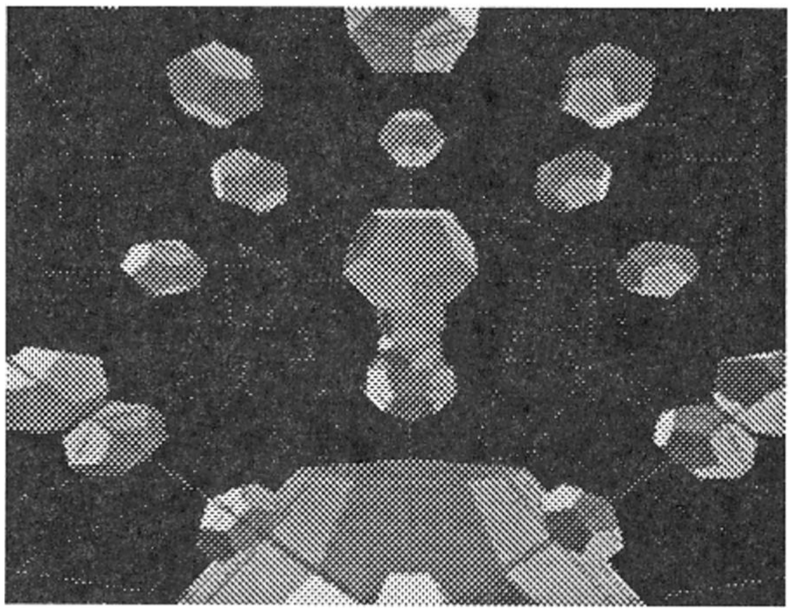


Figure 2.15: Dodecahedral tessellation of spherical 3-space. Rendering taken from an article by Hudson et al. (Hudson, Gunn, Francis, Sandin, & DeFanti, 1995).

Research in this area is ongoing, for example Jeff Weeks has created a Non-Euclidean Billiards in VR (Weeks, 2020), a virtual reality video game which can be played in three-dimensional spherical, Euclidean and hyperbolic spaces.

A video by David Madore gives a good overview of multiple projections of sphere and hyperbolic plane (Madore, Visualizing the sphere and the hyperbolic plane: five projections of each, 2013). It illustrates that Poincaré disk model is analogous to a stereographic projection.

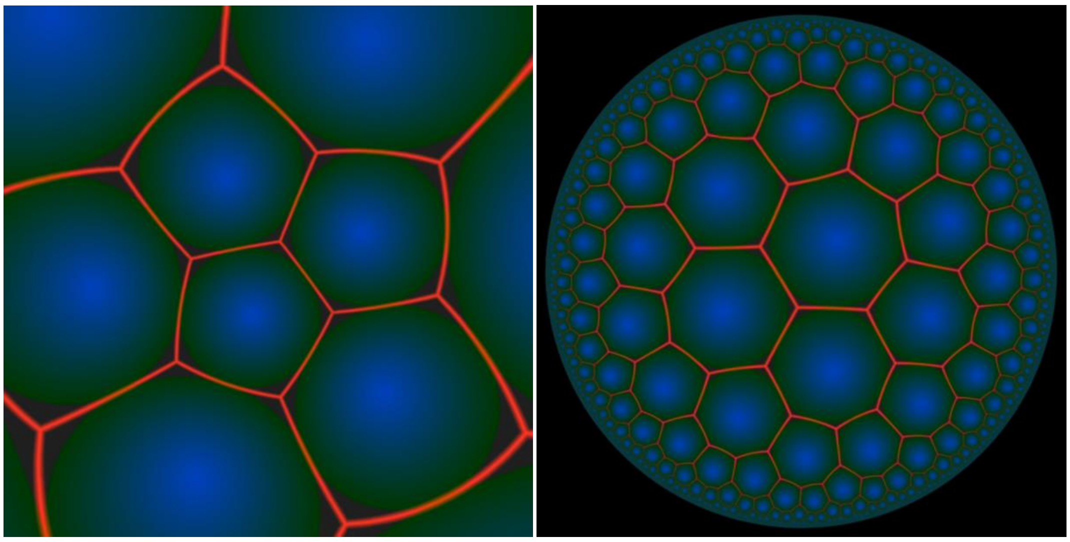


Figure 2.16: Stereographic projection of a sphere (left) and hyperbolic plane (right); from a video by David Madore (Madore, Visualizing the sphere and the hyperbolic plane: five projections of each, 2013).

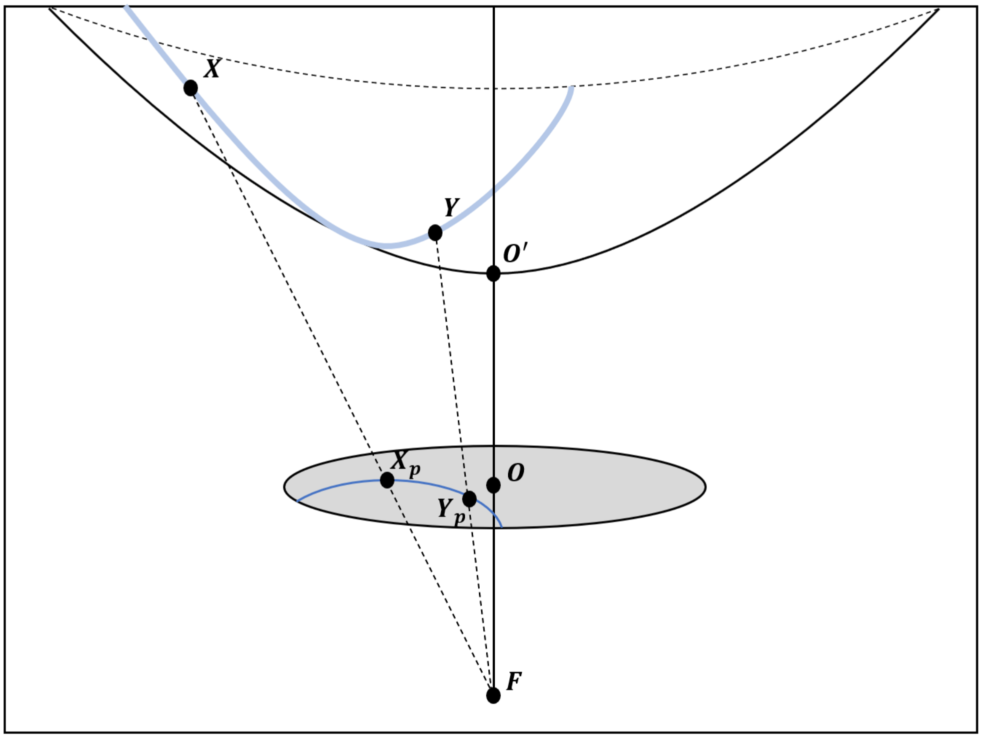


Figure 2.17: Stereographic projection of a hyperboloid onto a unit Poincaré disc centred on the origin $\boldsymbol{O}(\mathbf{0},\mathbf{0},\mathbf{0})$. $\boldsymbol{O}'(\mathbf{0},\mathbf{1},\mathbf{0})$ is the lowest point of the hyperboloid. $\boldsymbol{F}(\mathbf{0},-\mathbf{1},\mathbf{0})$ is the focal point of the stereographic projection. Points $\boldsymbol{X}$ and $\boldsymbol{Y}$ lying on a line on the surface of the hyperboloid are projected to points $\boldsymbol{X}_{\boldsymbol{p}}$ and $\boldsymbol{Y}_{\boldsymbol{p}}$ on the Poincaré disc.

Figure 2.16 shows a comparison between a stereographic projection of a sphere and a stereographic projection of a hyperboloid (Poincaré disk model). Figure 2.17 shows a how the stereographic projection of a hyperboloid onto a Poincaré disk is performed.

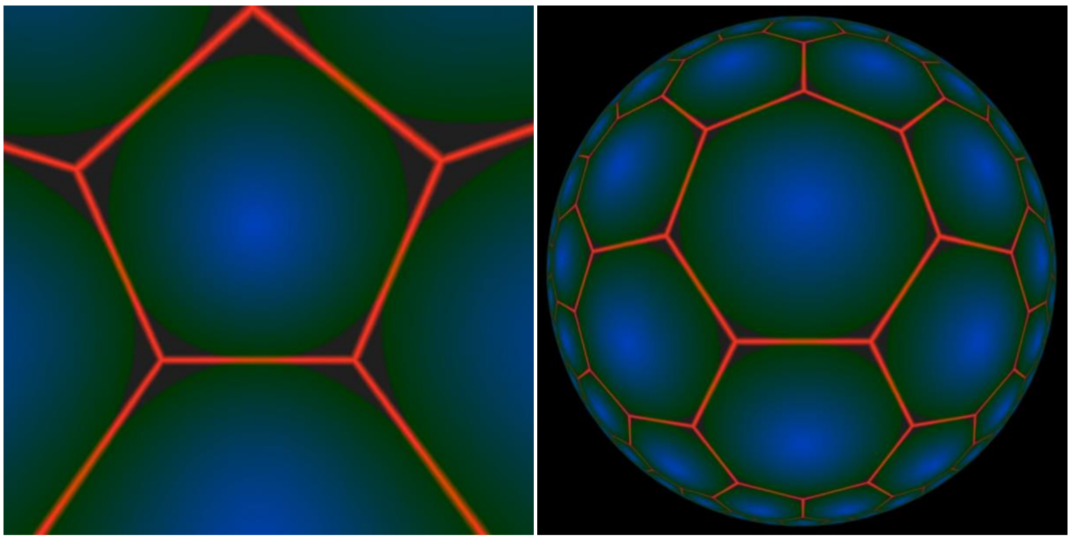


Figure 2.18: Gnomonic projection of a sphere (left) and hyperbolic plane (right); from a video by David Madore (Madore, Visualizing the sphere and the hyperbolic plane: five projections of each, 2013).

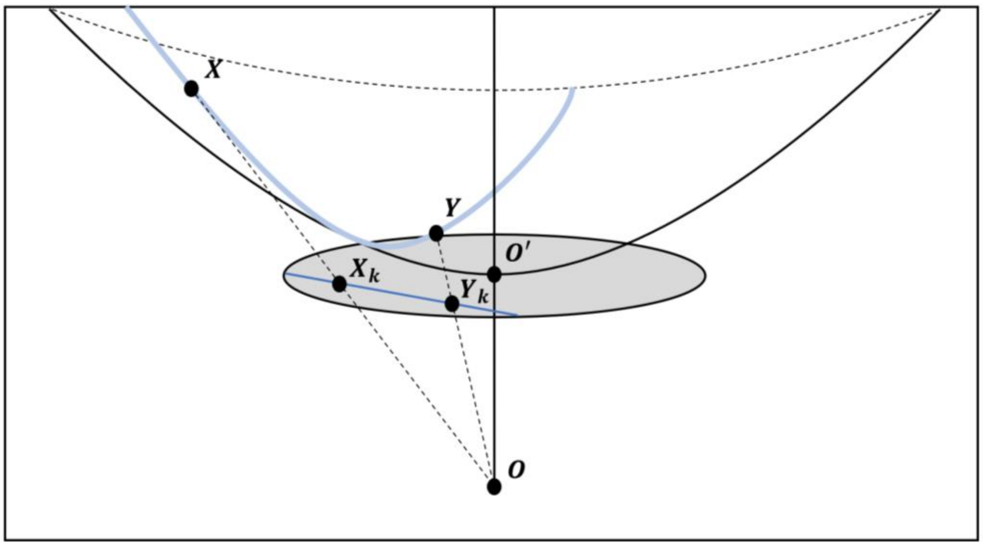


Figure 2.19: Gnomonic projection of a hyperboloid onto a unit Klein disc centred on the point $\boldsymbol{O'}(\mathbf{0},\mathbf{1},\mathbf{0})$, which is also the lowest point of the hyperboloid. $\boldsymbol{O}(\mathbf{0},\mathbf{0},\mathbf{0})$ is the origin as well as the focal point of the gnomonic projection. Points $\boldsymbol{X}$ and $\boldsymbol{Y}$ lying on a line on the surface of the hyperboloid are projected to points $\boldsymbol{X_k}$ and $\boldsymbol{Y_k}$ on the Klein disc.

Similarly, it shows that Beltrami-Klein model is analogous to a gnomonic projection and a comparison between a gnomonic projection of a sphere and hyperbolic plane is illustrated on

Figure 2.18. Figure 2.19 shows a how the gnomonic projection of a hyperboloid onto a Klein disk is performed.

Such online resources and articles are helpful to get a clearer picture of non-Euclidean spaces.

David Madore has also created a puzzle game in non-Euclidean space (Madore, Hyperbolic maze, 2013). It makes a maze by tiling the plane with five regular quadrilaterals around each point. Then some edges are removed to make routes for the player to move through. It shows how vast a hyperbolic space is and how disorienting it is for humans, due to being used to flat space. Hyperbolic maze can be played using both Poincaré Disk (shown on Figure 2.20) and Beltrami-Klein (shown in Figure 2.20) models.

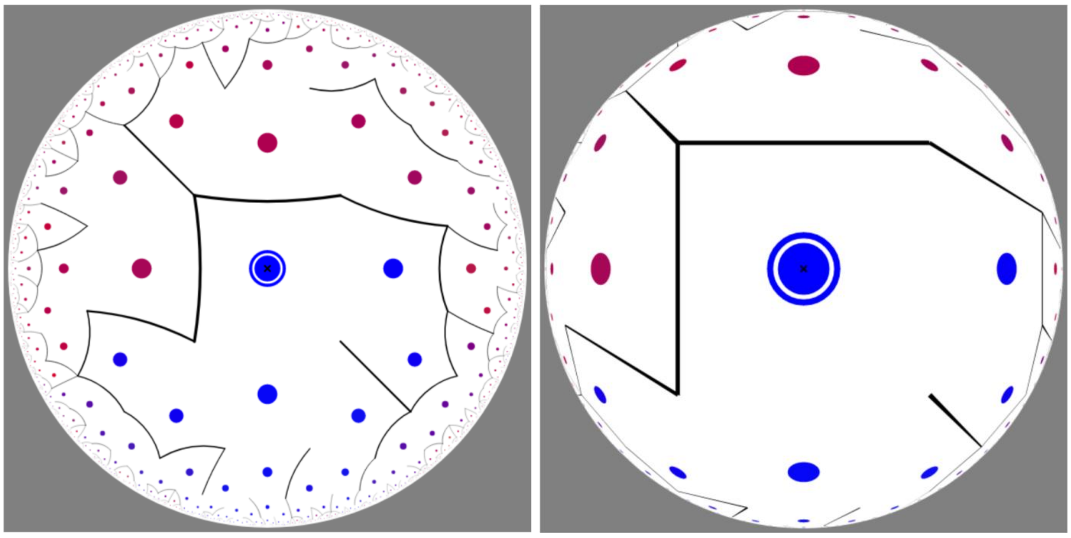
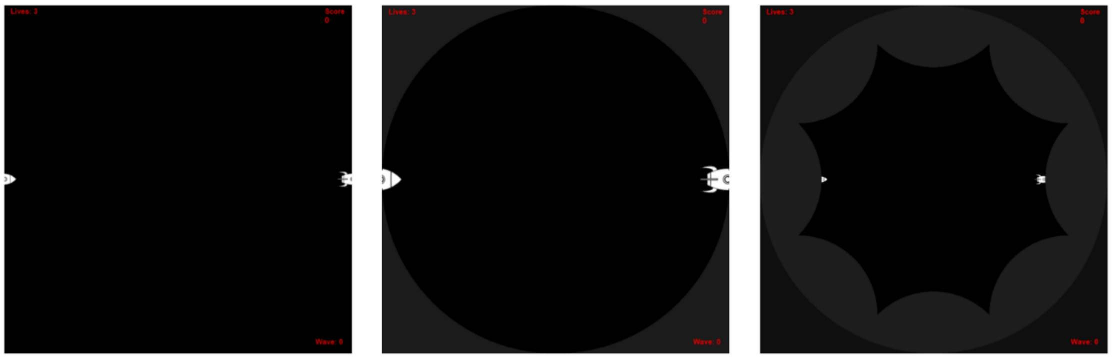


Figure 2.20: Hyperbolic Maze by David Madore. Rendered in a Poincaré disk model (left) and Beltrami-Klein model (right). The maze consists of 88,110 cells and has 73,700 walls. (Madore, Hyperbolic maze, 2013)

There is research in adapting existing games to non-Euclidean space, using both conformal (Guimarães, Mello, & Velho, 2015) and projective (László & Magdics, 2021).

Guimarães, et al. focussed on adapting a 2D game to spherical and hyperbolic space, specifically the Asteroids game (shown in Figure 2.21). The results described have similarities to the games built for the survey covered by chapter 11. Both studies are researching a way to use non-Euclidean geometry in 2D game development, but the approach taken is different. The authors describe a way of encapsulating internal controls of the game to separate them from the on-screen representation. This project instead focusses on creating a unified coordinate system which would allow seamless transition between the different curvatures at runtime.



(a) Glued Euclidean. (b) Glued Elliptic. (c) Glued Hyperbolic.

Figure 2.21: Space continuity in an Asteroids game for different types of geometries rendered by Guimarães, et al. (Guimarães, Mello, & Velho, 2015).

Hyperrogue is a turn based exploration game created by Zeno Rogue in late 2011. The world is tessellated via a heptagonal honeycomb into spaces the objects can occupy. This means that the world is discrete with objects moving in specific increments throughout the world.

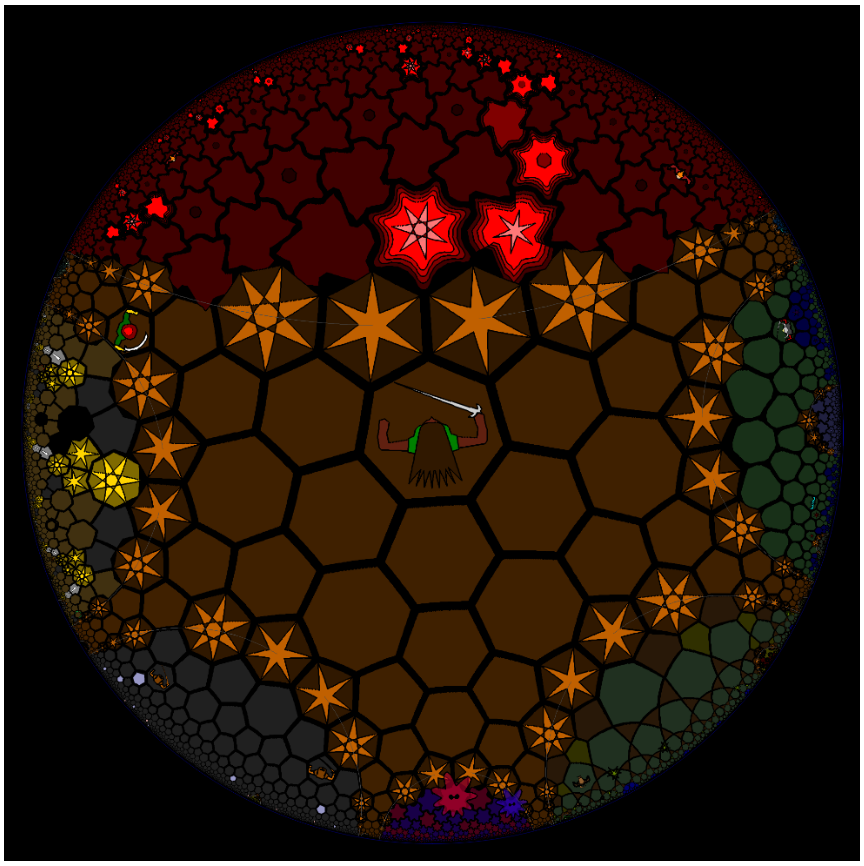


Figure 2.22: Screenshot of the Hyperrogue playthrough on the crossroads II land using the Escher art style (Zeno Rogue, 2022).

The environment is being procedurally generated to create an infinite world around the player. By default the Poincaré disk model is used to display the game, but other projections are also available and player can switch to using them instead. The art style is inspired by the paintings by M. C. Escher, specifically the Circle Limit series (one of the paintings in this series is shown in Figure 2.8).

Szirmay-Kalos László and Milán Magdics have considered and described the conversion of Euclidean objects, geometric calculations, transformation pipeline, lighting and physical simulation to non-Euclidean space. They have also demonstrated the results of the research by implementing three games, "Fight in space", "Lego" and "Museum", in non-Euclidean geometry. Only the space game (shown in Figure 2.23) has used dynamic simulation, while the other two games only focussed on converting the rendering, but not the object movement within the game world.

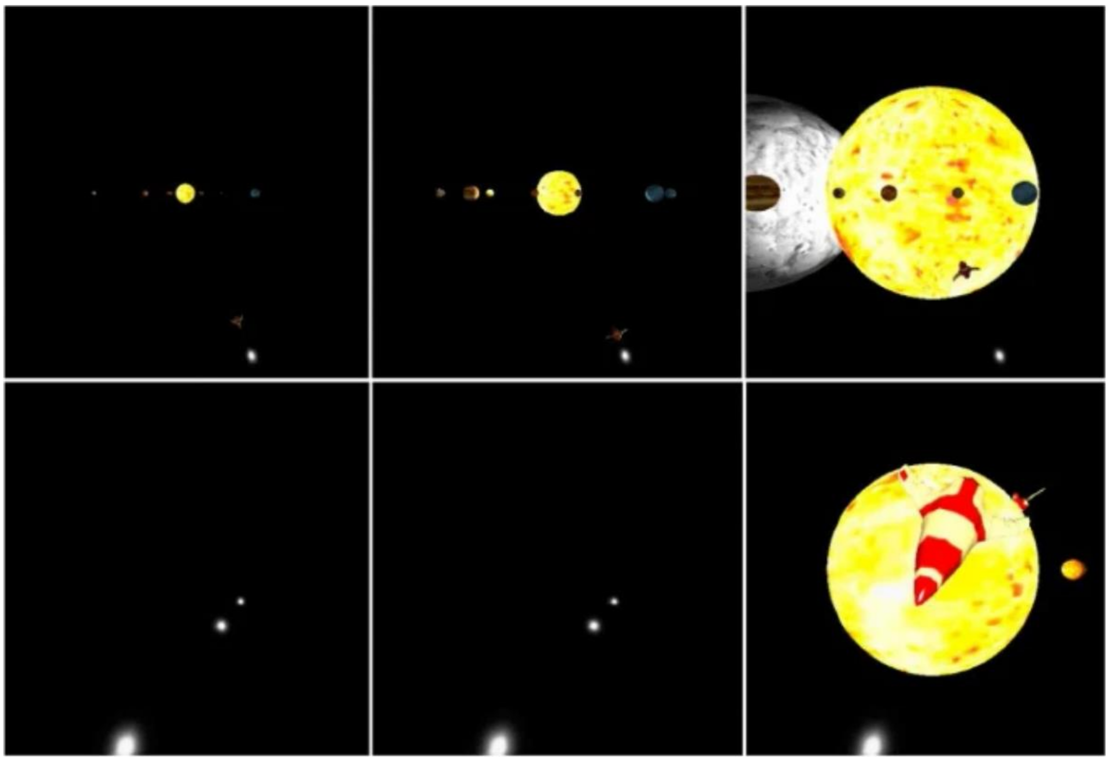


Figure 2.23: "Fight in space" game in hyperbolic (left) Euclidean (middle), and elliptic (right) spaces when looking forward (upper row) and backward (lower row). Rendered by Szirmay-Kalos László and Milán Magdics (László & Magdics, 2021).

Apart from applications in mathematics and physics research, video games and art, non-Euclidean geometry has found uses in data visualisation. Researchers at Stanford University have created a visualisation tool that can visualise large graphs by converting them into spanning trees and subsequently carrying out layout and drawing of the graph in 3D hyperbolic space (Munzner, 1998). Figure 2.24 shows a series of drawings of a large graph rendered using the visualisation tool called H3Viewer, developed by the article's author. It shows (from left to right) translation of a node to the centre of the graph. Such a representation allows users to see high information density, while obscuring clutter. The author states that the visualisation tool can handle graphs two orders of magnitude larger than previously developed systems at the time.

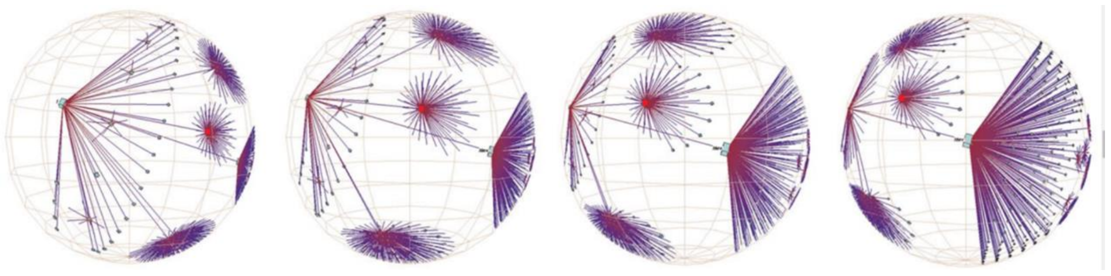


Figure 2.24: Hyperbolic motion over a 30,000-element Unix file system rendered with H3Viewer. Many nodes and edges project to subpixel areas and are not visible. Left to right, it shows translation of a node (cyan box in the figure) to the centre of the projection space (Munzner, 1998).

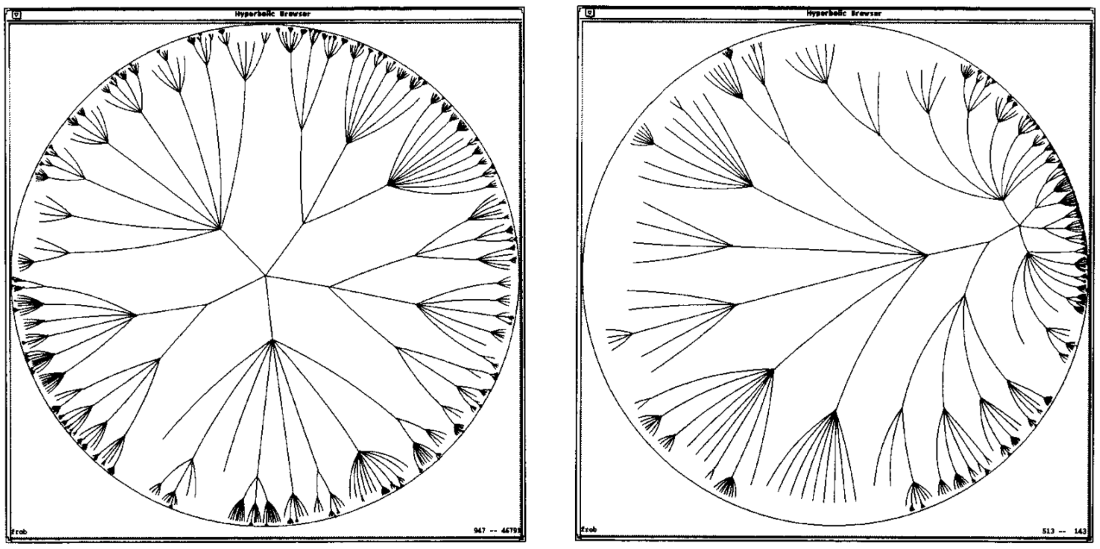


Figure 2.25: A large graph rendered with Hyperbolic Browser. The image on the right shows a transition, where the focus has moved to a node that was to the left and slightly below the origin in the left image (Lamping & Rao, 1996).

H3Viewer uses 3D projective model to render graphs, but other research has instead chosen to represent large hierarchies in a 2D conformal model (Lamping & Rao, 1996). Authors have developed a system called Hyperbolic Browser, which also works by representing a graph as a tree, but the drawing is done on a Poincaré disk. The conformal model allows users to follow the transitions in the graph more easily, due to the absence of flattening near the edge of the disk, and because the nodes are closer to the origin than in the Klein disk model. Figure 2.25 has a series of two renderings done with Hyperbolic Browser, which show the transition of focus to a different node. The authors describe the implementation of the tool as well as trade-offs between density of the nodes displayed and space to display node's information.

2.5 Summary

This chapter has covered the history of non-Euclidean geometry, its roots in the works of Euclid to the development of spherical and hyperbolic geometries. It also described the efforts in ways of visualising curved spaces, including conformal and projective models; research in creating immersive environments and potential applications of non-Euclidean visualisation.

3. Engine Structure and Planning

This chapter describes the structure of the software developed as part of the original research during this research project. Additionally, it describes the function of the components of the software, the coordinate system and screen-limitation of the space used within the software.

3.1 Engine Structure

The aim of this project was to create software capable of calculating and rendering real-time visualisations in curved space. These goals are similar to capabilities of game engines. As such, layered architecture pattern has been chosen to organise the structure of the software.

“Components within the layered architecture pattern are organized into horizontal layers, each layer performing a specific role within the application.” (Richards, 2015)

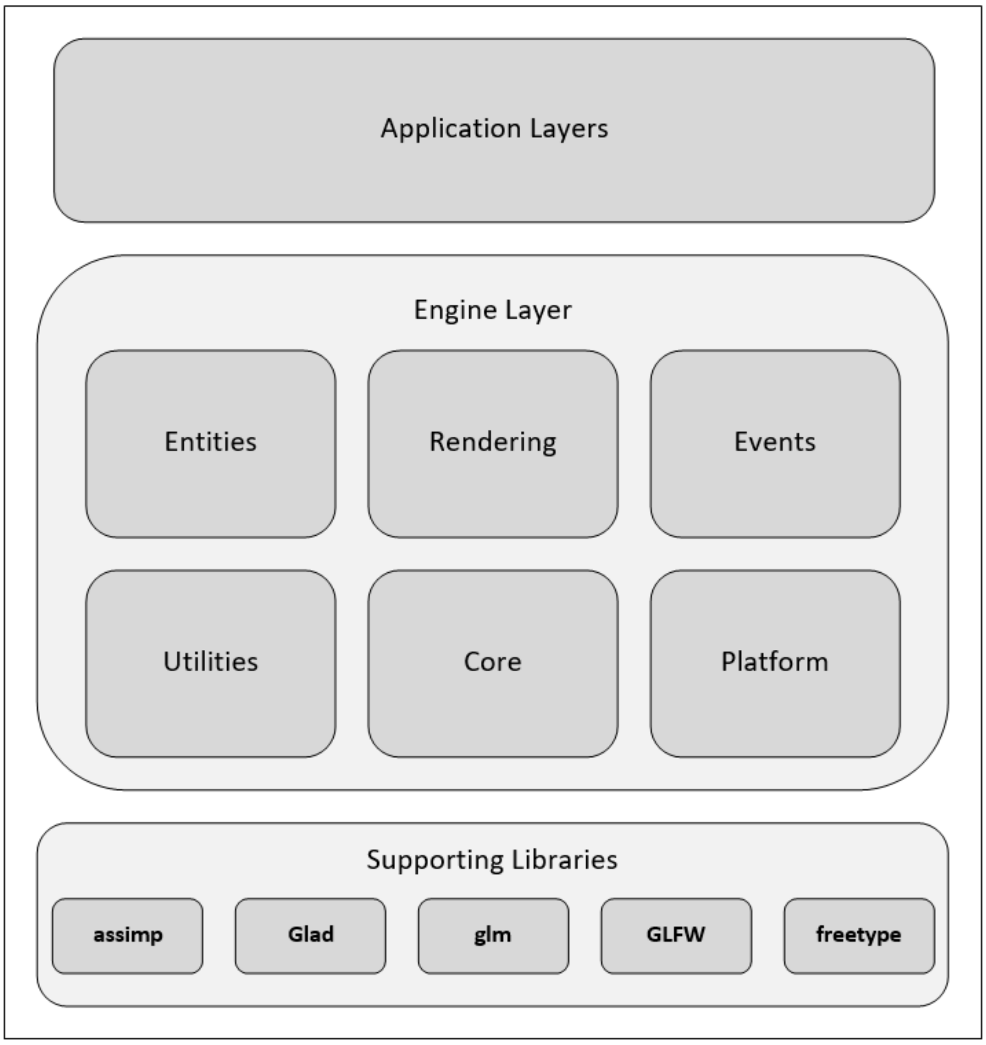


Figure 3.1: The high-level overview of the non-Euclidean engine’s structure, following the layered architecture pattern. Supporting libraries used are described in section 3.1.1.

The layered architecture pattern is widely used in game architecture because it inherently allows multiple levels of abstraction and loose coupling of the code. First, a framework is developed with common functionality for any game, like rendering engine, asset importers, platform interfacing, etc. Engine code is separated from the logic developed for each individual game to ensure adaptability and reusability of the code. Both of these traits are important for the project described. The Cherno project has been chosen as a good starting point to develop the base functionality of the engine (Chernikov, 2012). The Cherno project is an open source game engine written in C++ language using OpenGL. It is built to be versatile and adaptable making it a good starting point for this project, as it needed to have a custom made rendering pipeline, coordinate system and game physics, so having low level access to these parts of a game engine was crucial.

Figure 3.1 illustrates the structure of the engine developed during this project. It shows a clear separation of the engine functionality, external dependencies and the logic for the application (game or simulation) being developed.

3.1.1 Supporting Libraries

Functionality from the following external libraries have been used to develop the project:

- **GLFW (Graphics Library Framework):** lightweight, open-source, multi-platform library, which provides a simple, platform-independent API for creating windows, contexts and surfaces, reading input, handling events, etc (The GLFW Development Team, 2002).
- **GLM (OpenGL Mathematics):** header only mathematics library, providing classes and functions designed and implemented with the same naming conventions and functionality than GLSL so that anyone who knows GLSL, can use GLM as well in C++ (Riccio, 2005).
- **Glad:** a version of the OpenGL Loading Library. It loads pointers to OpenGL functions at runtime, which is required to access functions from OpenGL versions above 1.1 on most platforms. Additionally, it acts as an extension loading library, so abstracts away the difference between the loading mechanisms on different platforms (Khronos Group, 1997).
- **freetype:** open-source font rasterization engine used for rendering text onto bitmaps. It supports a number of font formats and font-related operations (Turner, Wilhelm, & Lemberg, 1996).

- **assimp (Open Asset Import Library):** an import library for a wide range of file formats. It aims to provide a common API for importing different formats. (Gessler, Schulze, & Kulling, 2006).

3.1.2 Packages within the Engine Layer

Within the engine's framework, the code is organised into classes, which are grouped into a number of packages depending on the functionality they provide.

The Core package (shown in Figure 3.2) contains the functionality to initialise the application, interfaces and layers, as well as configure an updating simulation loop.

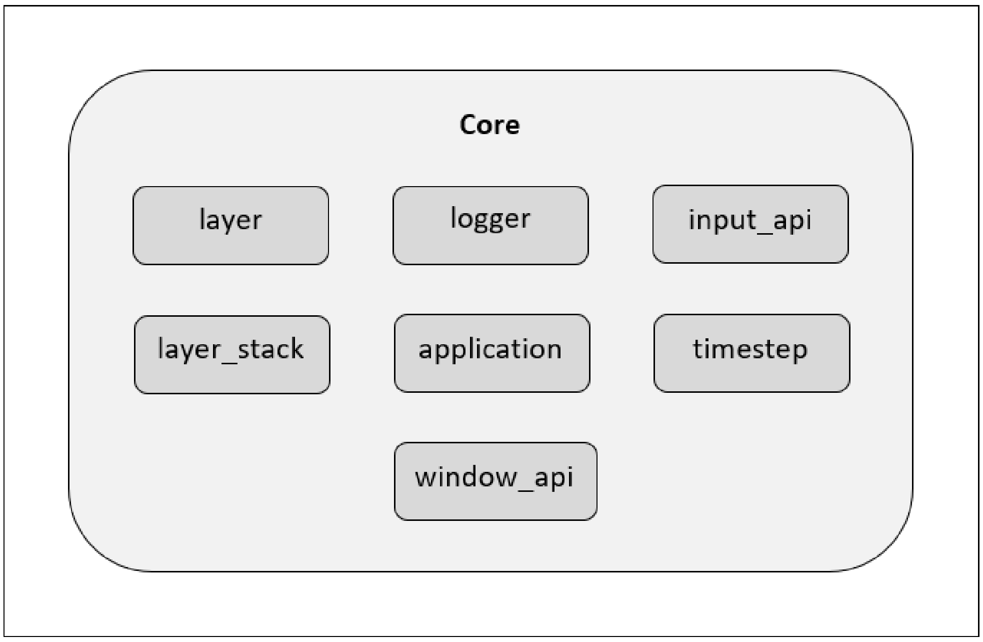


Figure 3.2: Classes contained within the Core package of the Engine

The simulation loop consists of an update method, which is responsible for calculating the timestep (time elapsed since last update call), taking any user input, processing the event calls and stepping through the physical simulation; and render method, which is responsible for configuring the hardware for specific rendering tasks and drawing the simulation on screen. The functionality for individual tasks (rendering, handling events, handling simulation) is contained within each respective package or application layer logic. Core package is responsible for the overall framework of the application.

Platform package (shown in Figure 3.3) contains platform specific functionality for communicating with the system software and connecting to the input and output APIs in order to use the underlying hardware. `gl_renderer_api` has been repeatedly modified during the project to contain the required methods for drawing lines, triangles and patches on the GPU.

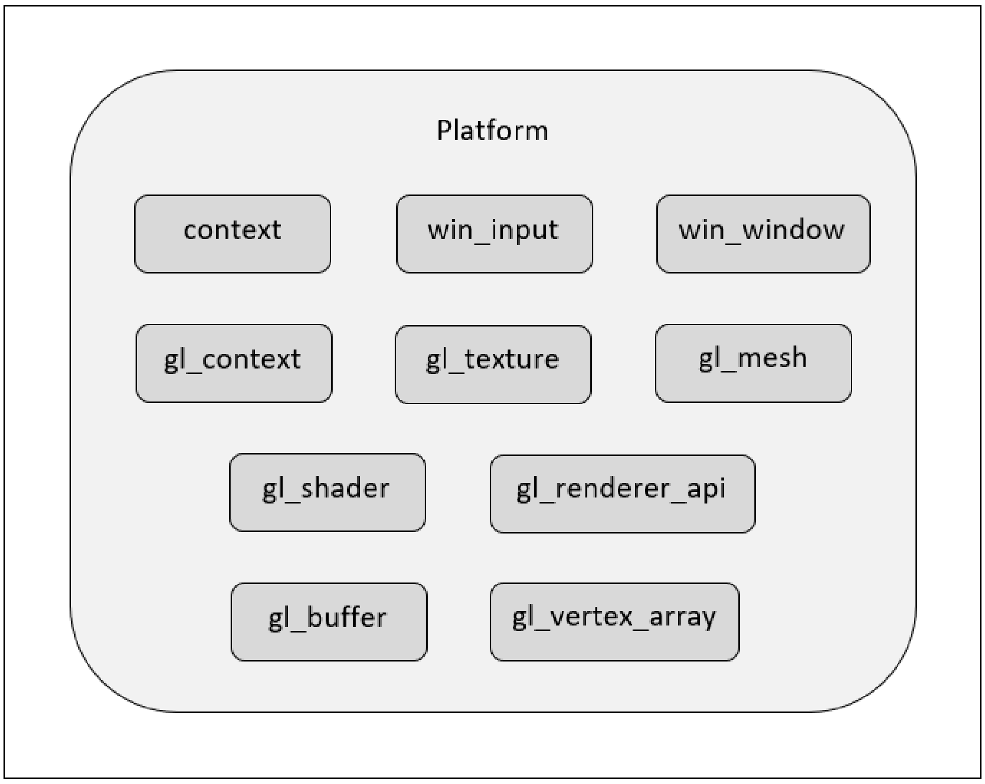


Figure 3.3: Classes contained within the Platform package of the Engine

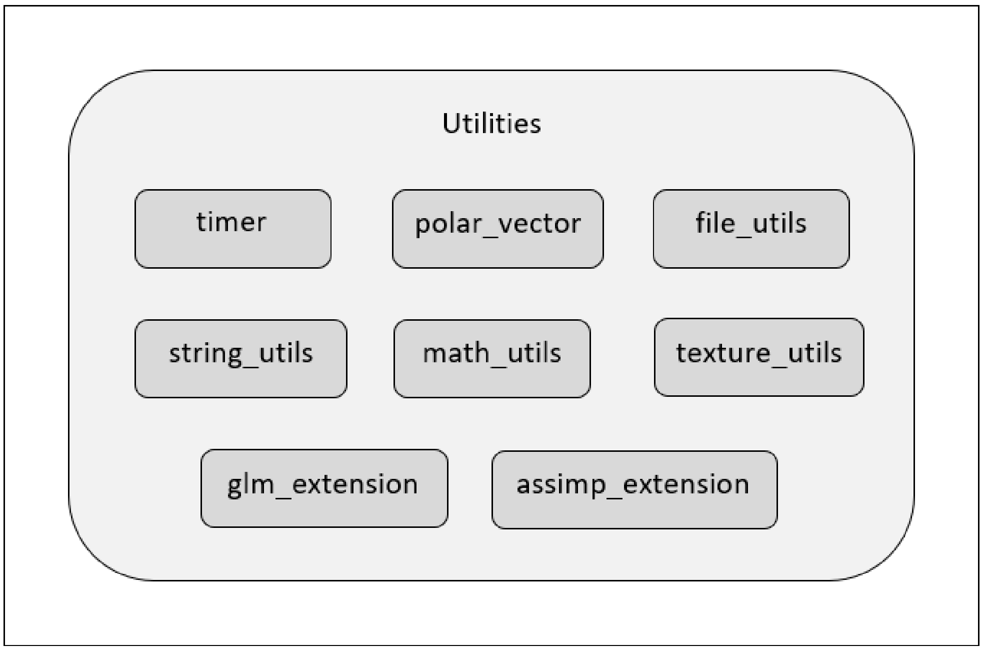


Figure 3.4: Classes contained within the Utilities package of the Engine

Utilities package (shown in Figure 3.4) contain various utility methods including additional mathematic functions, notably, a definition of a polar vector used throughout the project. This class has been created to contain all operations using the polar coordinate system (section 3.2).

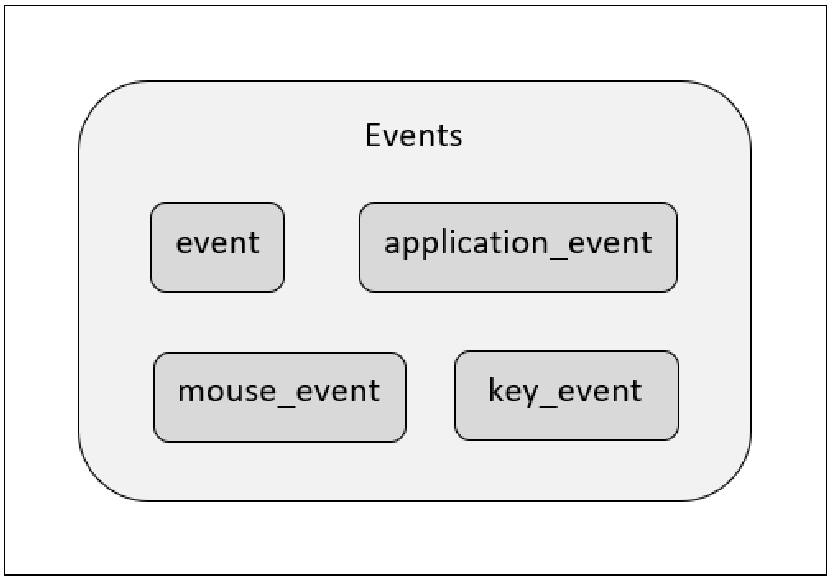


Figure 3.5: Classes contained within the Events package of the Engine

Events package (shown in Figure 3.5) contains classes and methods for listening to and handling input and application events. These include listening to the keyboard and mouse inputs, converting the input events received into game events and notifying the game event listeners. The classes in this package do not require specific modifications to work in non-Euclidean environment.

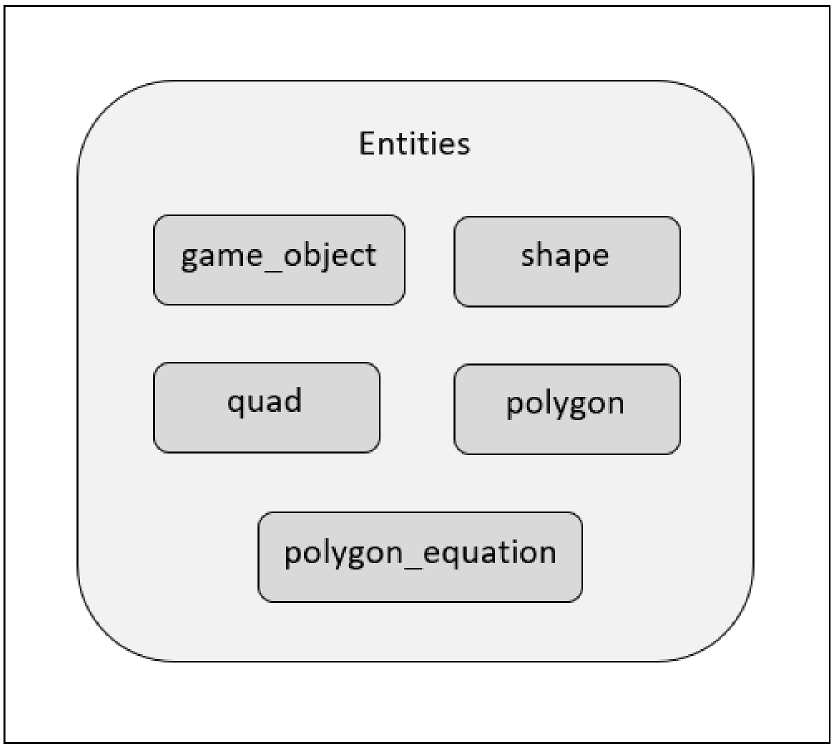


Figure 3.6: Classes contained within the Entities package of the Engine

The entities package (shown in Figure 3.6) contains the code for recording shape and object properties in the simulated environment. `game_object` contains methods for calculating object movement in curved space (chapter 5); `polygon` contains methods for tessellating shapes using spherical and hyperbolic trigonometry (chapter 4); `polygon_equation` contains methods for tessellating shapes using alternative tessellation approaches (chapter 6).

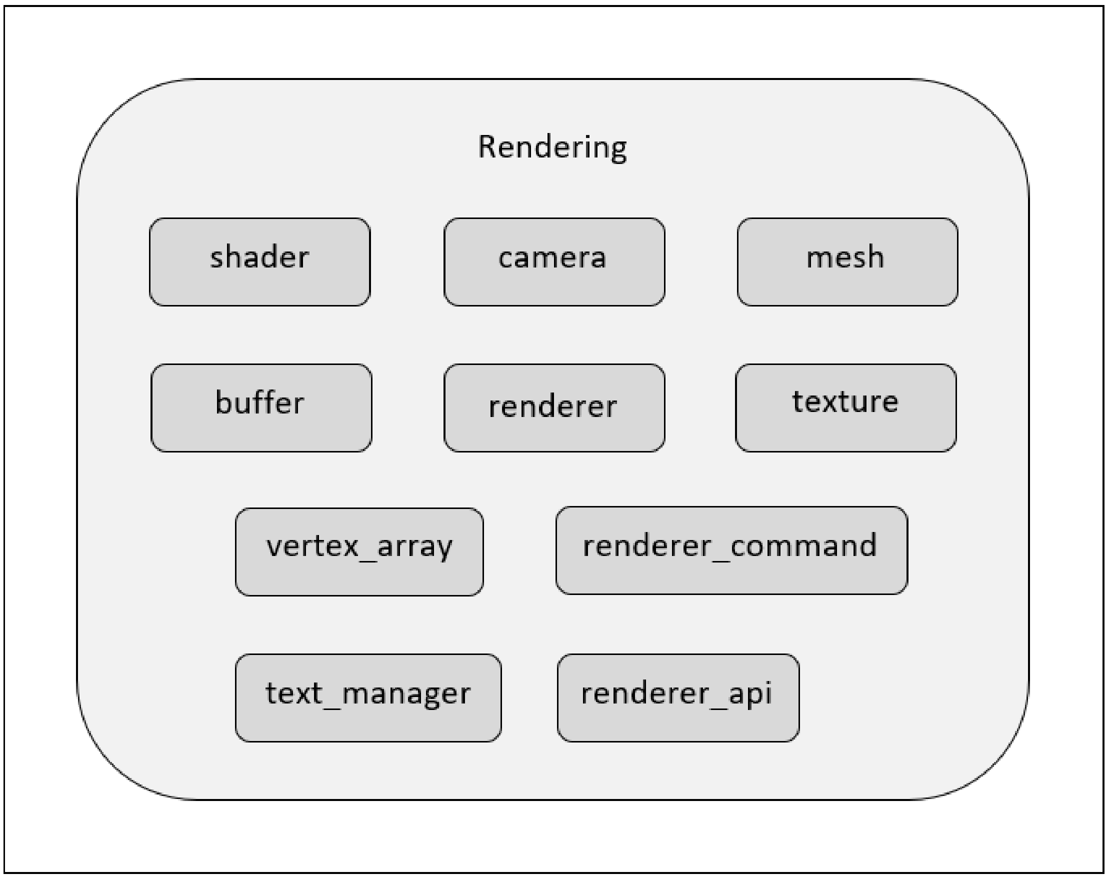


Figure 3.7: Classes contained within the Rendering package of the Engine

Rendering package (shown in Figure 3.7) has classes and functions for storing rendering information (meshes, textures, shaders, buffers and vertex arrays); initialising and updating camera views; setting up the scene for rendering and submitting commands to the OpenGL rendering pipeline. `renderer` class contains the functions to submit additional parameters for rendering shapes in curved space (e.g. curvature, tessellation parameters, etc.)

3.2 Polar Coordinate System

A polar coordinate system of the form $(\boldsymbol{r}, \boldsymbol{\theta})$ is used in this model for all of the object positions and calculations instead of Cartesian coordinates of the form $(\boldsymbol{x}, \boldsymbol{y})$.

The centre of the of the projection is taken as a reference point (analogous to the origin point in Cartesian coordinates) for the distance coordinate, $\boldsymbol{r}$, while eastbound direction is set to be the reference direction for the bearing coordinate, $\boldsymbol{\theta}$.

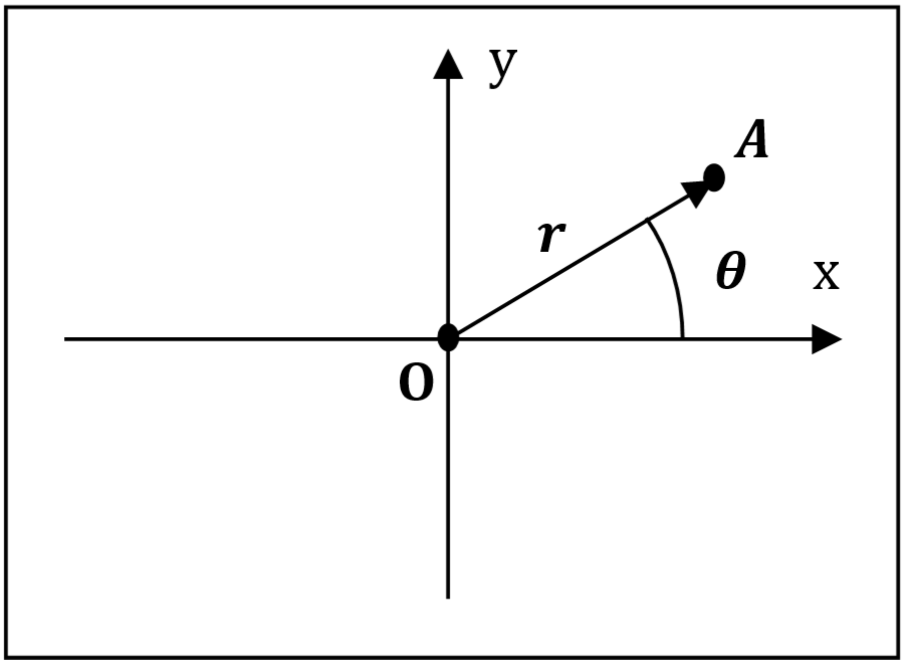


Figure 3.8: Position of a point A in polar coordinates. It has distance $\boldsymbol{r}$ and bearing $\boldsymbol{\theta}$ from the reference point $\boldsymbol{O}$.

Cartesian coordinates need to be adapted in order to be used for constant negative and positive curvature spaces, as the parallel lines, which are essential to pinpoint the location in Cartesian space, are fundamentally different in non-Euclidean space.

On the other hand, polar coordinates work just as well in any space of constant curvature, as the distance from a reference point and bearing from a reference direction still exist. This principle will help unify the coordinate system for any curvature of the space and ultimately allow the real-time change of curvature without any complications.

3.3 Screen-limited Space

To make the space represented in the engine be more useable for testing and rendering interactive scenes, a cut-off distance has been implemented. This distance is set at a distance of $\boldsymbol{r} = \boldsymbol{N}$ from the reference point of the global polar coordinate system. $\boldsymbol{N}$ is set to be half the screen size used in the application and $\boldsymbol{N} = \mathbf{500}$ is used for all of the screenshots in this thesis.

When the object's centre point moves further away from the origin than the distance $\boldsymbol{N}$, it is teleported to the antipodal point of the limiting circle. This happens in Euclidean and hyperbolic space, as well as spherical space when $\boldsymbol{K} < \mathbf{1}$. When $\boldsymbol{K} > \mathbf{1}$ the centre of the shape never goes past distance $\boldsymbol{N}$, as the whole spherical space is contained within this radius.

There are different ways to accomplish this limitation. In this engine, only the object's position is changed, while orientation, rotation and velocity are unchanged, as illustrated in Figure 3.9. This has some advantages: it is easy to implement programmatically and is similar in appearance to what is done in classic 2D games, like Asteroids and Pacman. If the object moves further away from the origin than the distance $\boldsymbol{N}$, it is teleported onto the screen from the antipodal point at the same distance. Because of the coordinate system used, it is easy to set or lift this limiting distance: the object's theta coordinate is increased by $\boldsymbol{\pi}$ and then standardised to be in range $\mathbf{0}$ to $\mathbf{2\pi}$. This makes it appear on the antipodal point of the limit circle with preserved velocity and orientation. This can be seen in the hyperbolic movement time-lapse images (Figure 5.4 and Figure 5.5).

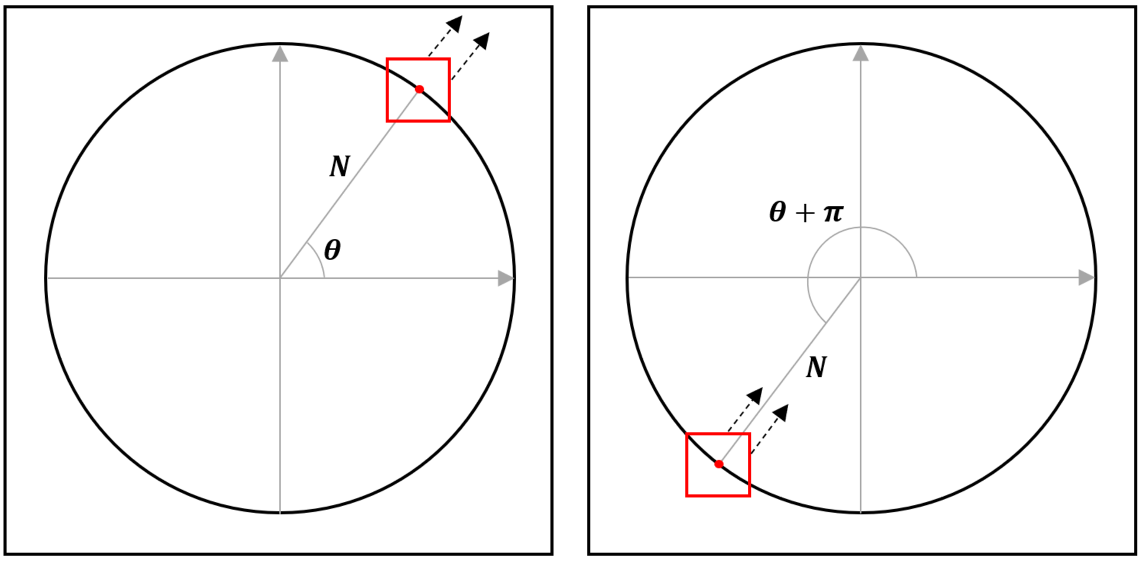


Figure 3.9: A quadrilateral object moves further than distance $\boldsymbol{N}$ *away from the reference point and teleports to the antipodal point maintaining the orientation, velocity and rotation values.*

However, this way of imposing the distance limit is not fully mathematically coherent, as the world is not a closed manifold. As an example, Asteroids game is set in a torus topological space, which is a simple closed manifold space (illustrated in Figure 3.10). It is mathematically coherent, but also quite simple to represent programmatically. Due to the nature of the curved space within this engine torus topological space would not work, as the shapes get radially stretched or compressed on the screen as they move away from the centre of the projection.

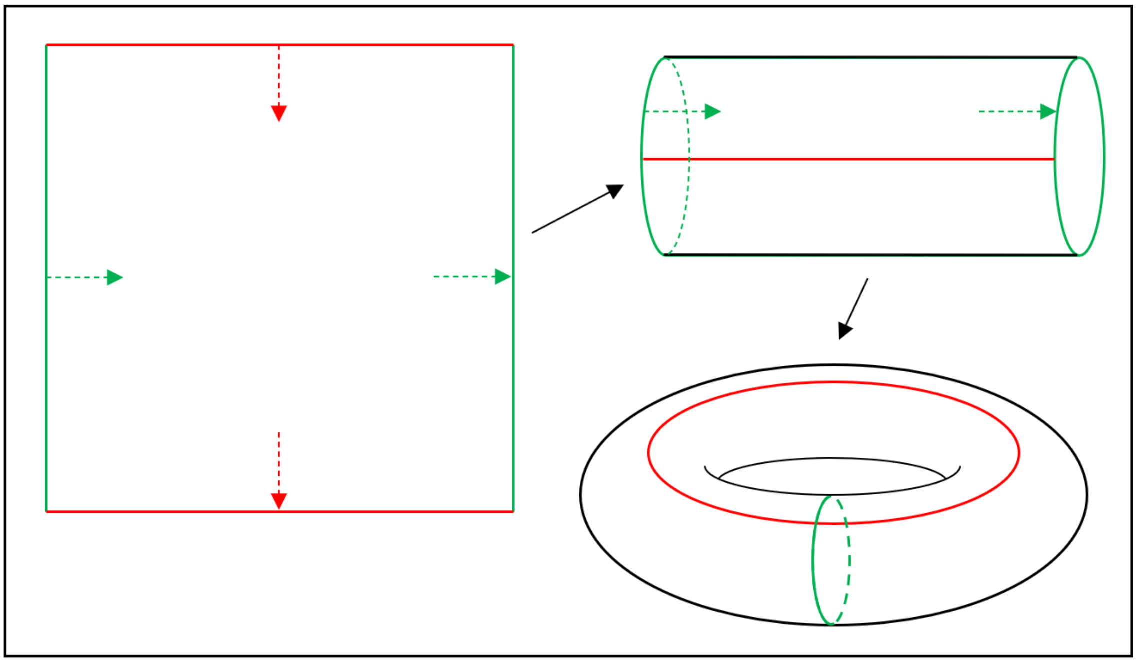


Figure 3.10: Folding of the torus topological space used for many 2D games. When an object moves off the bottom part of the screen, it moves on from the top and vice versa. When an object moves off the left part of the screen, it moves on from the right and vice versa.

If a circular space is treated as a closed manifold, then when any part of the object passes the limit distance $\boldsymbol{N}$, it reappears at the antipodal point of the limit circle. This results in the orientation and velocity vectors of the object being reflected with respect to the centre of the projection, as shown in Figure 3.11. This means that the world becomes non-oriented, which might confuse the players more than the approach used. Additionally, it would be more computationally intensive, because when part of the object is on the one side of the limit circle and the rest of the object is on the opposite side, the object would have to be rendered as two separate objects on the screen.

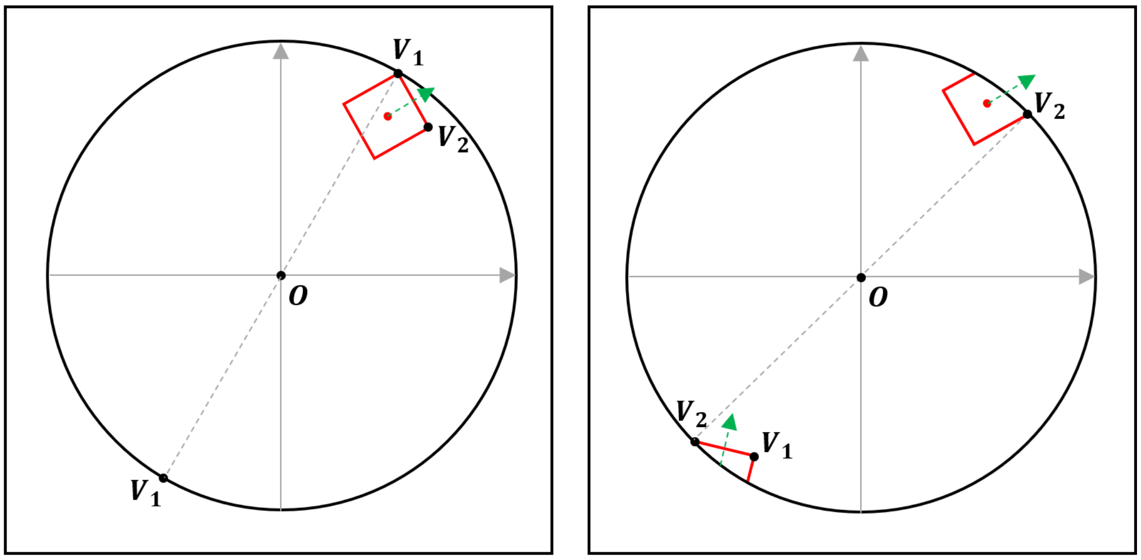


Figure 3.11: A quadrilateral object moves from one side of the projection to the other in a closed manifold space. The orientation and velocity vector of the object get reflected with respect to the centre of the projection.

3.4 Summary

This chapter has covered a high-level overview of the engine developed in this project. The chosen structure, in particular, the layered architecture pattern, has been explained. The function of each package within the engine as well as the polar coordinate system used throughout the engine have been described.

4. Method I: Rendering 2D Shapes using Spherical and Hyperbolic Trigonometry

This chapter is original research, which introduces a method for finding intermediate points along an object's edge used to tessellate and subsequently render the objects in curved space. It describes the trigonometric theorems used in this method and explains the method in three consecutive steps: determining global coordinates of the object's vertex; calculating preliminaries for the object's edge tessellation; and finding a number of intermediate points along the edge.

Note: angle magnitudes in each diagram and equation within this section are normalised to be in the range $[\mathbf{0}, \mathbf{2\pi}]$.

4.1 Law of Cosines in Spherical and Hyperbolic Trigonometry

In order to render the shape, global coordinates have to be calculated for all of the vertices of the shape. This model is using spherical and hyperbolic trigonometry in order to calculate these values. Let $\boldsymbol{K} \subset \mathbb{R}$ s.t.

$\boldsymbol{K} = 0 \rightarrow$ Euclidean Geometry

$\boldsymbol{K} > 0 \rightarrow$ Spherical Geometry, $\boldsymbol{r} = \frac{\mathbf{1}}{\sqrt{\mathbf{K}}}$, where $\boldsymbol{r}$ is the radius of the sphere with Gaussian curvature $\boldsymbol{K}$.

$\boldsymbol{K} < 0 \rightarrow$ Hyperbolic Geometry, $\boldsymbol{k} = \frac{\mathbf{1}}{\sqrt{\mathbf{K}}}$, where $\boldsymbol{k}$ is a pseudo-radius of a pseudosphere with Gaussian curvature $\boldsymbol{K}$.

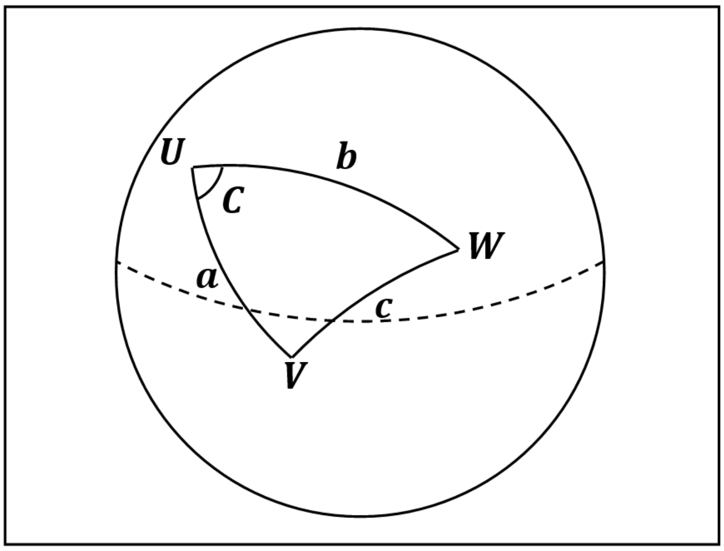


Figure 4.1: Spherical triangle with vertices $\boldsymbol{U}$, $\boldsymbol{V}$ and $\boldsymbol{W}$ on the surface of a sphere; $\overline{\boldsymbol{UV}} = \boldsymbol{a}$; $\overline{\boldsymbol{UW}} = \boldsymbol{b}$; $\overline{\boldsymbol{VW}} = \boldsymbol{c}$; $\angle \boldsymbol{VUW} = \boldsymbol{C}$

For a sphere of radius $\boldsymbol{r}$ and hence Gaussian curvature, $\boldsymbol{K} = \frac{\boldsymbol{1}}{\boldsymbol{r^2}}$, as well as a spherical triangle on its surface described by points $\boldsymbol{U}$, $\boldsymbol{V}$ and $\boldsymbol{W}$, connected by great circles that form the edges $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$ (interpreted as subtended angles) and an angle $\boldsymbol{C}$ (See Figure 4.1), the spherical law of cosines states (Gellert, 1989):

$$\boldsymbol{\cos\frac{c}{r} = \cos\frac{a}{r}\cos\frac{b}{r} + \sin\frac{a}{r}\sin\frac{b}{r}\cos C} \tag{4.1}$$

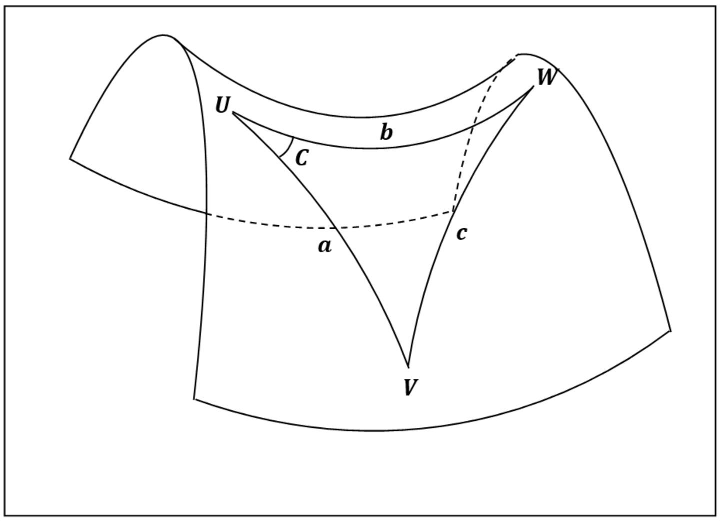


Figure 4.2: Hyperbolic triangle with vertices $\boldsymbol{U}$, $\boldsymbol{V}$ and $\boldsymbol{W}$ on the surface of a hyperbolic plane;
$\overline{\boldsymbol{UV}} = \boldsymbol{a}$*;* $\overline{\boldsymbol{UW}} = \boldsymbol{b}$*;* $\overline{\boldsymbol{VW}} = \boldsymbol{c}$*;* $\angle \boldsymbol{VUW} = \boldsymbol{C}$

For a hyperbolic plane with Gaussian curvature, $\boldsymbol{K} = \frac{\boldsymbol{1}}{\boldsymbol{k^2}}$, and a hyperbolic triangle on its surface described by points $\boldsymbol{U}$, $\boldsymbol{V}$ and $\boldsymbol{W}$, connected by geodesics that form the edges a, b and c, as well as an angle C (See Figure 4.2), the hyperbolic law of cosines states (Gray, 1979):

$$\boldsymbol{\cosh\frac{c}{k} = \cosh\frac{a}{k}\cosh\frac{b}{k} + \sinh\frac{a}{k}\sinh\frac{b}{k}\cos C} \tag{4.2}$$

4.2 Global Vertex Coordinates

The first objective is to determine the global vertex coordinates of each of the vertices of a shape being rendered. As described in section 3.2, shapes are described by the position of the centre, $\boldsymbol{C(r_c, \theta_c)}$; rotation of the shape, $\boldsymbol{\alpha}$; and an array of $\boldsymbol{n}$ vertex coordinates in local polar coordinates, $\boldsymbol{V_n(r_v, \theta_v)}$.

Using spherical and hyperbolic cosine rules, the global positions are found by considering the angle $\boldsymbol{\varDelta OCV}$ for each of the vertices individually, as shown in Figure 4.3. $\boldsymbol{O}$ is the reference point of the global polar coordinate system.

Note: in order to simplify the equations below, all of the lengths will be divided by $\boldsymbol{r}$ or $\boldsymbol{k}$ depending on the value of $\boldsymbol{K}$. Then the lengths that we are searching for will be multiplied by $\boldsymbol{r}$ or $\boldsymbol{k}$ to get the final answer, where $\boldsymbol{r = \frac{1}{\sqrt{K}}}$ when $\boldsymbol{K > 0}$ and $\boldsymbol{k = \frac{1}{\sqrt{K}}}$ when $\boldsymbol{K < 0}$.

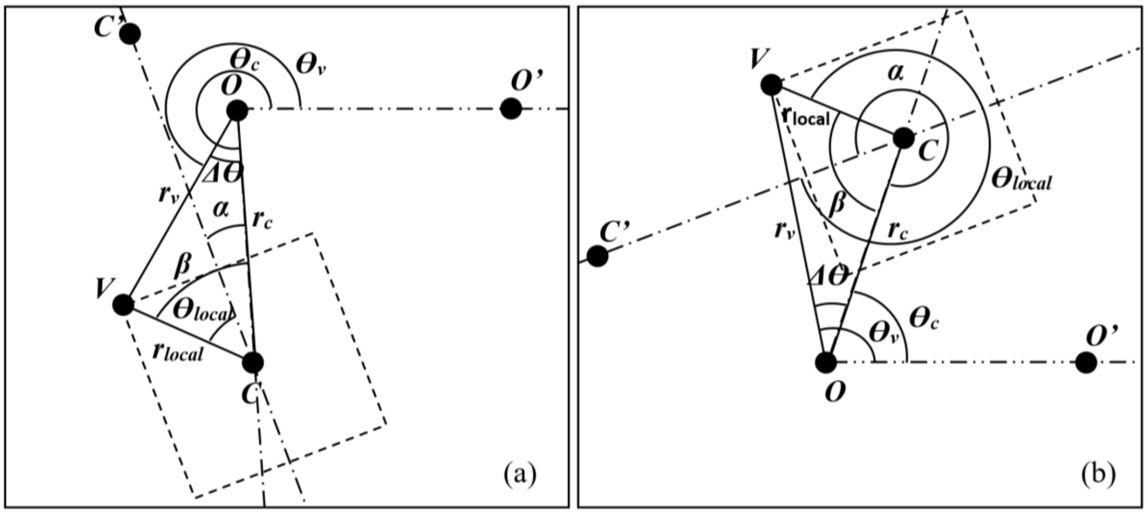


Figure 4.3: Finding the coordinates of an object's vertex $\boldsymbol{V(r_v, \theta_v)}$ *from a triangle* $\boldsymbol{\varDelta OCV}$. $\boldsymbol{O(0,0)}$, *reference point of a polar coordinate system;* $\boldsymbol{C(r_c, \theta_c)}$, *reference point of a local coordinate system;* $\boldsymbol{V(r_{local}, \theta_{local})}$, *local coordinates of the object's vertex;* $\boldsymbol{\overline{OV} = r_v}$, $\boldsymbol{\overline{OC} = r_c}$, $\boldsymbol{\overline{CV} = r_{local}}$; $\boldsymbol{\angle VOO' = \theta_v}$, $\boldsymbol{\angle VOC = \Delta\theta_v}$, $\boldsymbol{\angle COO' = \theta_c}$, $\boldsymbol{\angle VCC' = \theta_{local}}$, $\boldsymbol{\angle OCC' = \alpha}$, *object's angle of rotation,* $\boldsymbol{\angle OCV = \beta}$, *angle between* $\boldsymbol{r_{local}}$ *and* $\boldsymbol{r_c}$; *Case (a):* $\boldsymbol{\theta_{local} + \alpha} < \pi$; *case (b):* $\boldsymbol{\theta_{local} + \alpha} > \pi$.

Given: $\boldsymbol{O(0,0)}$, $\boldsymbol{C(r_c, \theta_c)}$, $\boldsymbol{V(r_v, \theta_v)}$, $\boldsymbol{\overline{OC} = r_c}$, $\boldsymbol{\overline{CV} = r_{local}}$, $\boldsymbol{\angle COO' = \theta_c}$, $\boldsymbol{\angle OCC' = \alpha}$, $\boldsymbol{\angle VCC' = \theta_{local}}$

Find: $\boldsymbol{r_v}$, $\boldsymbol{\theta_v}$

If $\boldsymbol{K} > 0$, then:

$$\boldsymbol{r_v} = \mathbf{cos^{-1}}(\mathbf{cos}\,\boldsymbol{r_c}\,\mathbf{cos}\,\boldsymbol{r_{local}} + \mathbf{sin}\,\boldsymbol{r_c}\,\mathbf{sin}\,\boldsymbol{r_{local}}\,\mathbf{cos}\,\boldsymbol{\beta}) \tag{4.3}$$

$$\boldsymbol{\Delta\theta_v} = \mathbf{cos^{-1}}\left(\frac{\mathbf{cos}\,\boldsymbol{r_{local}} - \mathbf{cos}\,\boldsymbol{r_c}\,\mathbf{cos}\,\boldsymbol{r_v}}{\mathbf{sin}\,\boldsymbol{r_c}\,\mathbf{sin}\,\boldsymbol{r_v}}\right) \tag{4.4}$$

If $\boldsymbol{K} < 0$, then:

$$\boldsymbol{r_v} = \mathbf{cosh^{-1}}(\mathbf{cosh}\,\boldsymbol{r_c}\,\mathbf{cosh}\,\boldsymbol{r_{local}} + \mathbf{sinh}\,\boldsymbol{r_c}\,\mathbf{sinh}\,\boldsymbol{r_{local}}\,\mathbf{cos}\,\boldsymbol{\beta}) \tag{4.5}$$

$$\boldsymbol{\Delta\theta_v} = \mathbf{cos^{-1}}\left(\frac{\mathbf{cosh}\,\boldsymbol{r_c}\,\mathbf{cosh}\,\boldsymbol{r_v} - \mathbf{cosh}\,\boldsymbol{r_{local}}}{\mathbf{sinh}\,\boldsymbol{r_c}\,\mathbf{sinh}\,\boldsymbol{r_v}}\right) \tag{4.6}$$

To find $\boldsymbol{r_v}$, first find $\angle\boldsymbol{OCV} = \boldsymbol{\beta}$. $\boldsymbol{\beta} = \boldsymbol{\alpha} + \boldsymbol{\theta_{local}}$; however, if $\boldsymbol{\pi} < \beta < 2\boldsymbol{\pi}$, use the explementary angle of $\boldsymbol{\beta}$ instead. This is done to determine to which side of $\overline{\boldsymbol{OC}}$ the triangle lies, which will be used to find $\boldsymbol{\theta_v}$. To find $\boldsymbol{\theta_v}$, we calculate $\angle\boldsymbol{VOC} = \boldsymbol{\Delta\theta_v}$, the angular difference between $\boldsymbol{\theta_v}$ and $\boldsymbol{\theta_c}$, which is then added to or subtracted from $\boldsymbol{\theta_c}$, depending on whether the explementary angle of $\boldsymbol{\beta}$ was taken or not. This is illustrated on Figure 4.3.

Note: Equations *(4.3)* and *(4.4)* could become unstable when the angles are small. To make calculations more precise the haversine formula (Korn & Korn, 2000) could be used instead for spherical space. For small angles instead of equations *(4.3)* and *(4.4)*, use the following two equations respectively:

$$\boldsymbol{r_v} = \mathbf{hav^{-1}}(\mathbf{hav}(\boldsymbol{r_c} - \boldsymbol{r_{local}}) + \mathbf{sin}\,\boldsymbol{r_c}\,\mathbf{sin}\,\boldsymbol{r_{local}}\,\mathbf{hav}\,\boldsymbol{\beta}) \tag{4.7}$$

$$\boldsymbol{\Delta\theta_v} = \mathbf{hav^{-1}}\left(\frac{\mathbf{hav}\,\boldsymbol{r_{local}} - \mathbf{hav}(\boldsymbol{r_c} - \boldsymbol{r_v})}{\mathbf{sin}\,\boldsymbol{r_c}\,\mathbf{sin}\,\boldsymbol{r_v}}\right) \tag{4.8}$$

In the engine, these two formulae have been used within the vertex shader, described in section 7.1.1, due to the calculations becoming less stable on the GPU than on the CPU when $\boldsymbol{\beta} < \frac{\boldsymbol{\pi}}{\mathbf{60}}$.

4.3 Intermediate Points Preliminaries

The method described in section 0 has to be used repeatedly to find every vertex of an object. However, that is not enough information to draw curved lines onto a screen. After all of the global vertex coordinates have been found, to proceed with rendering, a geodesic between the two vertices should be tessellated into smaller straight lines, which could be rendered on the screen.

Hence intermediate points should be found along the lines connecting the vertices of the shape. For that, some preliminaries have to be found first: the length of the edge between the two vertices as well as the total angle between the position vectors of the two vertices (illustrated in Figure 4.4).

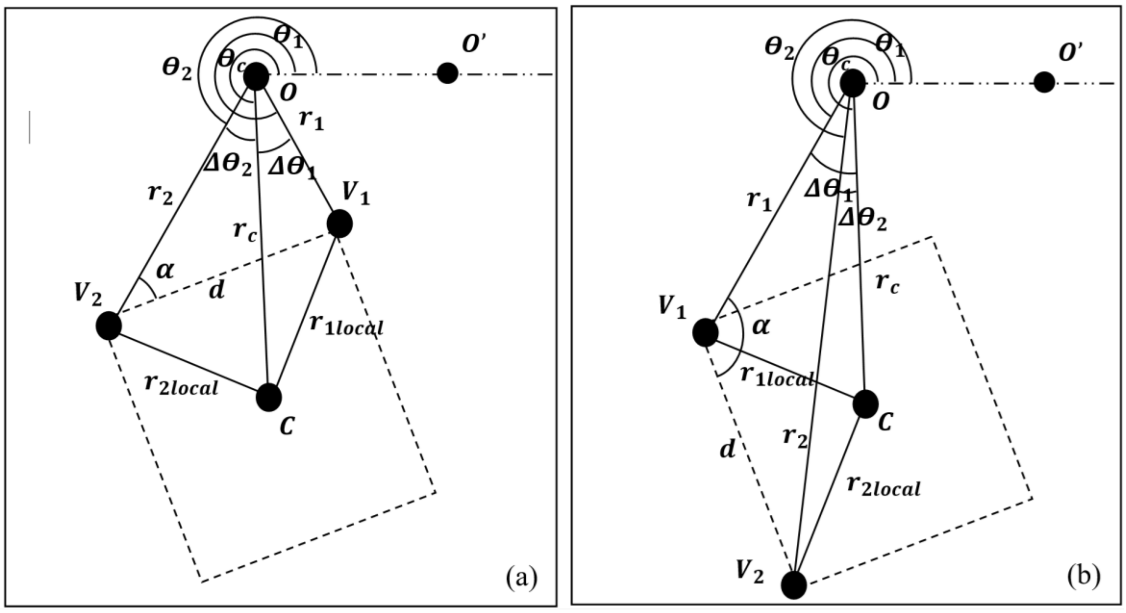


Figure 4.4: Finding preliminaries ($\boldsymbol{d}$, length of the edge $\overline{V_1V_2}$; $\boldsymbol{\Delta\theta}$, angle between $\overline{OV_1}$ and $\overline{OV_2}$) to calculate intermediate points on the edge $\overline{V_1V_2}$. Case (a): the angles $\Delta\theta_1$ and $\Delta\theta_2$ are diverging; case (b): the angles $\Delta\theta_1$ and $\Delta\theta_2$ are converging. $O(0,0)$, reference point of the global coordinate system; $C(r_c,\theta_c)$, position of the object; $V_1(r_1,\theta_1)$, $V_2(r_2,\theta_2)$, vertices of the object; $\overline{OC}=r_c$, $\overline{OV_1}=r_1$, $\overline{OV_2}=r_2$, $\overline{CV_1}=r_{1local}$, $\overline{CV_2}=r_{2local}$, $\angle COO'=\theta_c$, $\angle V_1OO'=\theta_1$, $\angle V_2OO'=\theta_2$, $\angle V_1OC=\Delta\theta_1$, $\angle V_2OC=\Delta\theta_2$, $\angle OV_1V_2=\alpha$

Given: $O(0,0)$, $C(r_c,\theta_c)$, $V_1(r_1,\theta_1)$, $V_2(r_2,\theta_2)$, $\overline{OC}=r_c$, $\overline{OV_1}=r_1$, $\overline{OV_2}=r_2$, $\overline{CV_1}=r_{1local}$, $\overline{CV_2}=r_{2local}$, $\angle COO'=\theta_c$, $\angle V_1OO'=\theta_1$, $\angle V_2OO'=\theta_2$

Find: d, $\Delta\theta$

In order to find $\boldsymbol{\Delta\theta}$, we first find $\boldsymbol{\Delta\theta_1}$ and $\boldsymbol{\Delta\theta_2}$:

$$\boldsymbol{\Delta\theta_1 = \theta_c - \theta_1} \tag{4.9}$$

$$\boldsymbol{\Delta\theta_2 = \theta_c - \theta_2} \tag{4.10}$$

Here we could have 2 cases: diverging angles and converging angles (see Figure 4.4).

Angles diverge if $\boldsymbol{\Delta\theta_1} < 0$, $\boldsymbol{\Delta\theta_2} > 0$ or $\boldsymbol{\Delta\theta_1} > 0$, $\boldsymbol{\Delta\theta_2} < 0$. Then:

$$\boldsymbol{\Delta\theta} = |\boldsymbol{\Delta\theta_1}| + |\boldsymbol{\Delta\theta_2}| \tag{4.11}$$

Angles converge if $\boldsymbol{\Delta\theta_1} < 0$, $\boldsymbol{\Delta\theta_2} < 0$ or $\boldsymbol{\Delta\theta_1} > 0$, $\boldsymbol{\Delta\theta_2} > 0$. Then:

$$\boldsymbol{\Delta\theta} = |\boldsymbol{\Delta\theta_1} - \boldsymbol{\Delta\theta_2}| \tag{4.12}$$

In order to find $\boldsymbol{d}$, consider $\triangle \boldsymbol{OV_1V_2}$.

If $\boldsymbol{K} > 0$, then:

$$\boldsymbol{d = \cos^{-1}}(\boldsymbol{\cos r_1 \cos r_2 + \sin r_1 \sin r_2 \cos \Delta\theta}) \tag{4.13}$$

If $\boldsymbol{K} < 0$, then:

$$\boldsymbol{d = \cosh^{-1}}(\boldsymbol{\cosh r_1 \cosh r_2 + \sinh r_1 \sinh r_2 \cos \Delta\theta}) \tag{4.14}$$

Note: We also need to record which of $\boldsymbol{\Delta\theta_1}$ and $\boldsymbol{\Delta\theta_2}$ is the greater angle, as that determines the direction of the edge $\boldsymbol{d}$ used in the next step.

Last preliminary needed is $\boldsymbol{\alpha}$, the angle between $\boldsymbol{r_1}$, the position vector of $\boldsymbol{V_1}$, and the edge $\boldsymbol{d}$.

If $\boldsymbol{K} > 0$, then:

$$\boldsymbol{\alpha = \cos^{-1}\left(\frac{\cos r_2 - \cos r_1 \cos d}{\sin r_1 \sin d}\right)} \tag{4.15}$$

If $\boldsymbol{K} < 0$, then:

$$\boldsymbol{\alpha = \cos^{-1}\left(\frac{\cosh r_1 \cosh d - \cosh r_2}{\sinh r_1 \sinh d}\right)} \tag{4.16}$$

Note: Equations *(4.13)* and *(4.15)* could become unstable when the angles are small. To make calculations more precise the haversine formula (Korn & Korn, 2000) could be used instead for spherical space. For small angles instead of equations *(4.13)* and *(4.15)*, use the following two equations respectively:

$$\boldsymbol{d} = \mathbf{hav}^{-1}(\mathbf{hav}(\boldsymbol{r_1} - \boldsymbol{r_2}) + \sin \boldsymbol{r_1} \sin \boldsymbol{r_2}\, \mathbf{hav}\, \boldsymbol{\Delta\theta}) \tag{4.17}$$

$$\boldsymbol{\alpha} = \mathbf{hav}^{-1}\left(\frac{\cos \boldsymbol{r_2} - \mathbf{hav}(\boldsymbol{r_1} - \boldsymbol{d})}{\sin \boldsymbol{r_1} \sin \boldsymbol{d}}\right) \tag{4.18}$$

In the engine, these two formulae have been used within the tessellation control shader, described in section 7.1.2, due to the calculations becoming less stable on the GPU than on the CPU when $\boldsymbol{\beta} < \frac{\boldsymbol{\pi}}{\mathbf{10}}$.

4.4 Coordinates of the Intermediate Points

The last step in rendering shapes in curved space is to find the global polar coordinates of the intermediate points. The number of intermediate points, $\boldsymbol{d_i}(\boldsymbol{r_i},\ \boldsymbol{\theta_i})$ along each edge is determined by the tessellation parameter. The $\boldsymbol{r}$ and $\boldsymbol{\theta}$ coordinates of $\boldsymbol{d_i}$ are found from the $\triangle \boldsymbol{OV_1V_i}$ as illustrated in Figure 4.5.

Angle $\boldsymbol{\alpha}$ was calculated as an in-between step to find the angle opposite $\boldsymbol{r_i}$ to apply the rule of cosines. Then $\boldsymbol{r_i}$ and subsequently $\boldsymbol{\Delta\theta_i}$ can be found using the cosine rule. To calculate the distance $\boldsymbol{r_i}$, the triangle $\overline{\boldsymbol{V_1V_i}}$ is considered. Using the previously found angle $\boldsymbol{\alpha}$ as well as the known lengths $\boldsymbol{r_1}$ and $\overline{\boldsymbol{V_1V_i}}$ *($\boldsymbol{d_i}$)* a hyperbolic/spherical cosine rule can be applied (illustrated on Figure 4.5).

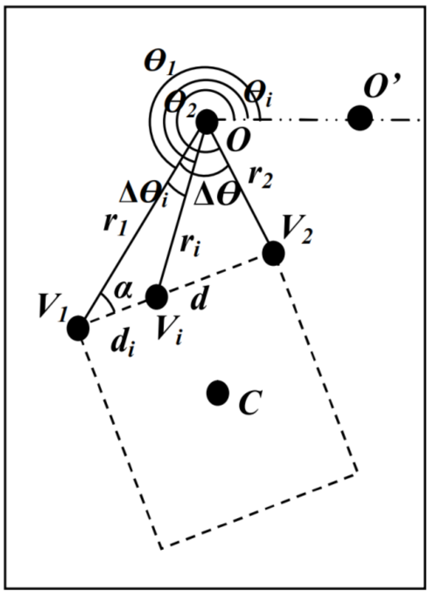


Figure 4.5: Finding intermediate points in order to render the edge $\overline{\boldsymbol{V_1V_2}}$. $\boldsymbol{O(0,0)}$, reference point of the global coordinate system; $\boldsymbol{C(r_c,\theta_c)}$, position of the object; $\boldsymbol{V_1(r_1,\theta_1)}$, $\boldsymbol{V_2(r_2,\theta_2)}$, vertices of the object; $\boldsymbol{d}$, the length of the edge $\overline{\boldsymbol{V_1V_2}}$; $\boldsymbol{V_i(r_i,\theta_i)}$, point on the edge $\overline{\boldsymbol{V_1V_2}}$; $\boldsymbol{\Delta\theta_i}$, angle between $\overline{\boldsymbol{OV_1}}$ and $\overline{\boldsymbol{OV_i}}$; $\boldsymbol{\Delta\theta}$, angle between $\overline{\boldsymbol{OV_1}}$ and $\overline{\boldsymbol{OV_2}}$.

Note: distance $\boldsymbol{d}$ is divided into a number of equal parts in order to find the distance $\boldsymbol{d_i}$ for each of the points on the edge $\overline{\boldsymbol{V_1V_2}}$. The number of segments depends on the object tessellation variable. Tessellation variable is a variable used to specify the number of segments the line has to be subdivided into.

Given: $\boldsymbol{O(0,0)}$, $\boldsymbol{C(r_c, \theta_c)}$, $\boldsymbol{V_1(r_1, \theta_1)}$, $\boldsymbol{V_2(r_2, \theta_2)}$, $\boldsymbol{V_i(r_i, \theta_i)}$, $\boldsymbol{\overline{OC} = r_c}$, $\boldsymbol{\overline{OV_1} = r_1}$, $\boldsymbol{\overline{OV_2} = r_2}$, $\boldsymbol{\overline{V_1V_2} = d}$, $\boldsymbol{\overline{V_1V_i} = d_i}$, $\boldsymbol{\angle COO' = \theta_c}$, $\boldsymbol{\angle V_1OO' = \theta_1}$, $\boldsymbol{\angle V_2OO' = \theta_2}$, $\boldsymbol{\angle V_1OV_2 = \Delta\theta}$

Find: $\boldsymbol{r_i}$, $\boldsymbol{\theta_i}$

If $\boldsymbol{K} > 0$, then:

$$\boldsymbol{r_i = \cos^{-1}(\cos r_1 \cos d_i + \sin r_1 \sin d_i \cos \alpha)} \tag{4.19}$$

$$\boldsymbol{\Delta\theta_i = \cos^{-1}\left(\frac{\cos d_i - \cos r_1 \cos r_i}{\sin r_1 \sin r_i}\right)} \tag{4.20}$$

If $\boldsymbol{K} < 0$, then:

$$\boldsymbol{r_i = \cosh^{-1}(\cosh r_1 \cosh d_i - \sinh r_1 \sinh d_i \cos \alpha)} \tag{4.21}$$

$$\boldsymbol{\Delta\theta_i = \cos^{-1}\left(\frac{\cosh r_1 \cosh r_i - \cosh d_i}{\sinh r_1 \sinh r_i}\right)} \tag{4.22}$$

Then to find actual coordinates of the point $\boldsymbol{V_i}$, $\boldsymbol{r_i}$ should be multiplied by $\boldsymbol{r}$ or $\boldsymbol{k}$ depending on the value of $\boldsymbol{K}$; $\boldsymbol{\Delta\theta_i}$ should be added to or subtracted from angle $\boldsymbol{\theta_1}$, depending on the direction of the edge $\boldsymbol{d}$, determined previously.

Note: Equations *(4.19)* and *(4.20)* could become unstable when the angles are small. To make calculations more precise the haversine formula (Korn & Korn, 2000) could be used instead for spherical space. For small angles instead of equations *(4.19)* and *(4.20)*, use the following two equations respectively:

$$\boldsymbol{r_i = \mathrm{hav}^{-1}(\mathrm{hav}(r_1 - d) + \sin r_1 \sin d \,\mathrm{hav}\, \alpha)} \tag{4.23}$$

$$\boldsymbol{\Delta\theta_i = \mathrm{hav}^{-1}\left(\frac{\cos d - \mathrm{hav}(r_1 - r_i)}{\sin r_1 \sin r_i}\right)} \tag{4.24}$$

In the engine, these two formulae have been used within the tessellation evaluation shader, described in section 7.1.3, due to the calculations becoming less stable on the GPU than on the CPU when $\boldsymbol{\beta < \frac{\pi}{10}}$.

4.5 Results

Using the method described in this chapter and OpenGL, software was created that has a capability to calculate and render the objects using the vector graphics in curved space.

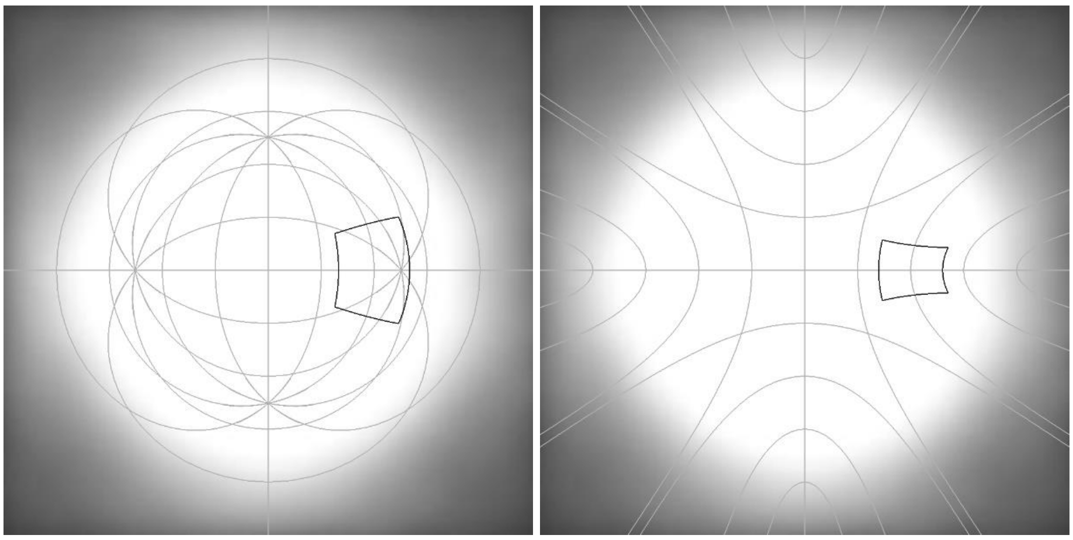


Figure 4.6: A regular quadrilateral with the centre at position $(\mathbf{200}\ \boldsymbol{px}, \mathbf{0}\ \boldsymbol{rad})$ *and rotation of* $\mathbf{0}\ \boldsymbol{rad}$*. Rendered with spherical trigonometry (left) and hyperbolic trigonometry (right). The white circle has the radius of* $\mathbf{500}\ \boldsymbol{px}$*.*

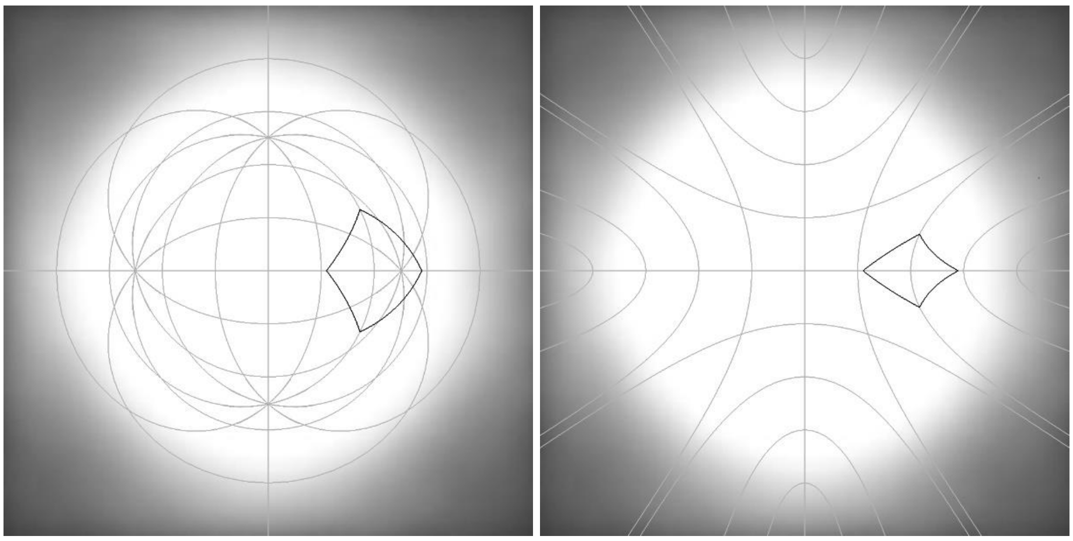


Figure 4.7: A regular quadrilateral with the centre at position $(\mathbf{200}\ \boldsymbol{px}, \mathbf{0}\ \boldsymbol{rad})$ *and rotation of* $\frac{\boldsymbol{\pi}}{\mathbf{4}}\ \boldsymbol{rad}$*. Rendered with spherical trigonometry (left) and hyperbolic trigonometry (right). The white circle has the radius of* $\mathbf{500}\ \boldsymbol{px}$*.*

Figure 4.6, Figure 4.7, Figure 4.8 and Figure 4.9 show a single regular quadrilateral rendered in a space with positive curvature on ($\boldsymbol{K} = \mathbf{1}$) the left and negative curvature ($\boldsymbol{K} = -\mathbf{1}$) on the right. Note that the grid-lines have been created and rendered as separate game objects, hence there is no need to recalculate them manually when the curvature changes.

Figure 4.6 and Figure 4.7 show the quadrilateral positioned at a point $(\mathbf{200\ px}, \mathbf{0\ rad})$ in a polar coordinate system of a form $(\boldsymbol{r}, \boldsymbol{\theta})$ and show the correct transformation of the shape in an Azimuthal equidistant projection in spherical and hyperbolic space. In Figure 4.6 the shape is not rotated, while in Figure 4.7 the shape is rotated by 45 degrees.

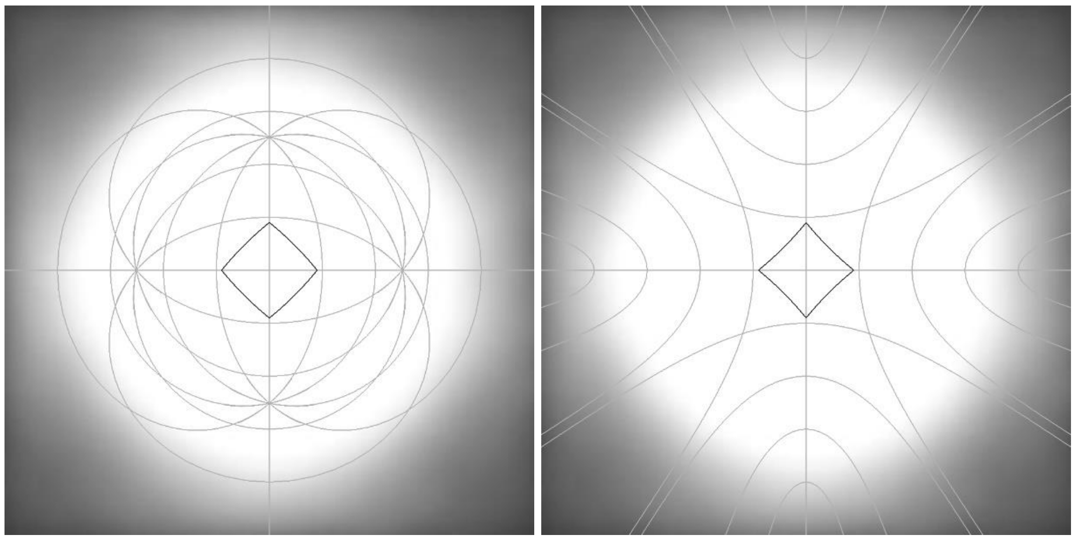


Figure 4.8: A regular quadrilateral with the centre at position $(\boldsymbol{0\ px}, \boldsymbol{0\ rad})$ *and rotation of* $\boldsymbol{\frac{\pi}{4}\ rad}$*. Rendered with spherical trigonometry (left) and hyperbolic trigonometry (right). The white circle has the radius of* $\boldsymbol{500\ px}$*.*

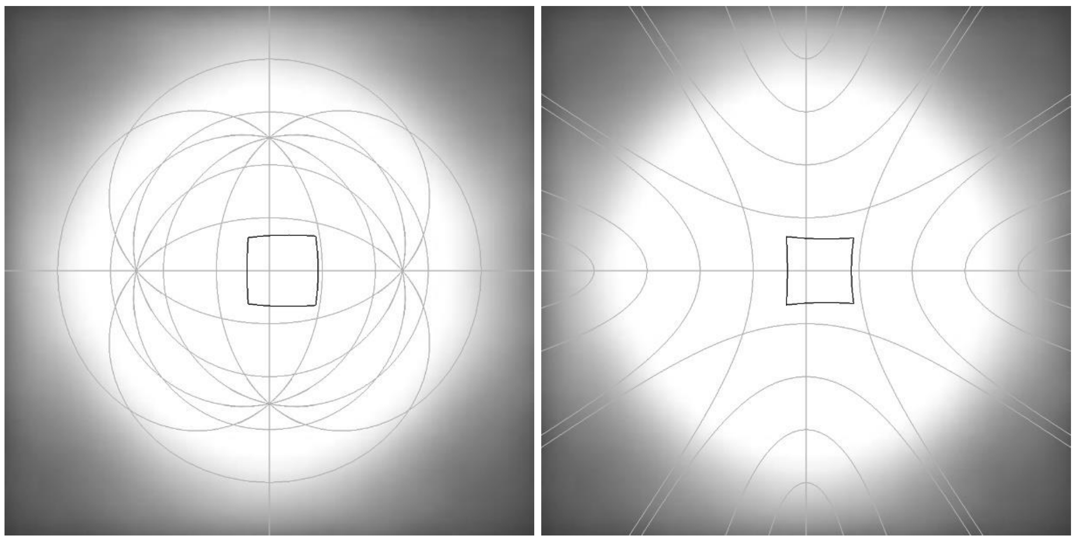


Figure 4.9: A regular quadrilateral with the centre at position $(\boldsymbol{25\ px}, \boldsymbol{0\ rad})$ *and rotation of* $\boldsymbol{0\ rad}$*. Rendered with spherical trigonometry (left) and hyperbolic trigonometry (right). The white circle has the radius of* $\boldsymbol{500\ px}$*.*

Figure 4.8 show the quadrilateral positioned at the centre of projection, point $(\mathbf{0\ px}, \mathbf{0\ rad})$, rotated by 25 45 degrees. Figure 4.9, shows the shape not rotated, but horizontally displaced from the centre of projection to the point $(\mathbf{25\ px}, \mathbf{0\ rad})$.

4.6 Summary

This chapter has described a method for rendering shapes in curved space in three steps using spherical and hyperbolic trigonometry. First, the global vertex coordinates of a 2D shape are calculated given the shape's position, rotation and its local vertex coordinates. Once the global vertex coordinates are found, the edges of the shape can be tessellated. For each edge, preliminaries for tessellation have to be calculated based on the global vertex coordinates of the endpoints of the respective edge. These preliminaries are the length of the edge as well as the angle between the edge and the position vector of vertex 1. Once these preliminaries are found, the intermediate points along the edge are calculated. The chapter also illustrates the renderings created using the software developed in this project.

5. Method II: Object Movement in a non-Euclidean Environment

This chapter is original research, which introduces a method developed for calculating object movement in real time in non-Euclidean space using spherical (Gellert, 1989) and hyperbolic trigonometry (Gray, 1979). This method works by translating the object to its position at the next time instance (frame in the engine). The trajectory of movement in non-Euclidean space is a geodesic and the object has to keep the same orientation with respect to it when moving.

Note: angle magnitudes in each diagram and equation within this section are normalised to be in the range $[\mathbf{0}, \mathbf{2\pi}]$.

5.1 Method

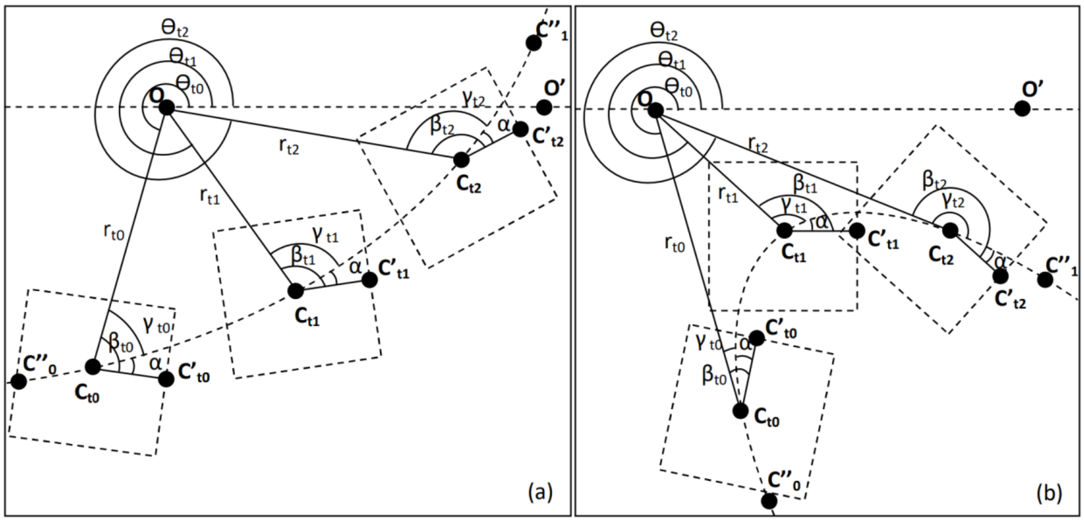


Figure 5.1: Movement of the object along a geodesic in Spherical (a) and Hyperbolic (b) space. Orientation with respect to the geodesic (angle $\boldsymbol{\alpha}$) is kept constant if the object is not rotating. $\boldsymbol{O}(\mathbf{0},\mathbf{0})$, reference point of the global coordinate system with; $\boldsymbol{C_{tx}}(\boldsymbol{r_{tx}}, \boldsymbol{\theta_{tx}})$, position of the object at time $\boldsymbol{x}$; $\boldsymbol{\beta_{tx}}$, object's rotation angle at time $\boldsymbol{x}$; $\boldsymbol{\gamma_{tx}}$, object's velocity vector direction at time $\boldsymbol{x}$.

Note: In addition to updating the global position vector of an object, the velocity vector and rotation angle have to be updated accordingly in order to keep their orientation towards the geodesic consistent (for a non-rotating object; for a rotating object, extra rotation over time should be added after the position of the object was recalculated).

Given: $\mathbf{O}(\mathbf{0},\mathbf{0})$, $\mathbf{C_{t0}}(\mathbf{r_{t0}}, \boldsymbol{\theta}_{\mathbf{t0}})$, $\mathbf{C_{t1}}(\mathbf{r_{t1}}, \boldsymbol{\theta}_{\mathbf{t1}})$, $\overline{\boldsymbol{OC_{t0}}} = \boldsymbol{r_{t0}}$, $\overline{\boldsymbol{C_{t0}C_{t1}}} = \mathbf{r_p}$, $\angle \boldsymbol{C_{t0}OO'} = \boldsymbol{\theta_{t0}}$, $\angle \boldsymbol{OC_{t0}O''} = \boldsymbol{\gamma_{t0}}$, $\angle \boldsymbol{OC_{t0}C'_{t0}} = \boldsymbol{\beta_{t0}}$

Find: $\mathbf{r_{t1}}, \mathbf{\theta_{t1}}, \mathbf{\beta_{t1}}, \mathbf{\gamma_{t1}}$

$\boldsymbol{\gamma_{t0}}$ should be in the range $\mathbf{0}$ to $\boldsymbol{\pi}$, take explementary angle if $\boldsymbol{\gamma_{t0}} > \pi$. This will determine the direction of the movement with respect to the reference point (needed to calculate $\boldsymbol{\theta_{t1}}$).

Let $\angle \boldsymbol{OC_{t1}C_{t0}} = \boldsymbol{\gamma'_{t1}}$

If $\boldsymbol{K} > 0$, then:

$$\boldsymbol{r_{t1} = \cos^{-1}\left(\cos r_{t0} \cos r_p + \sin r_{t0} \sin r_p \cos \gamma_{t0}\right)} \tag{5.1}$$

$$\boldsymbol{\Delta\theta = \cos^{-1}\left(\frac{\cos r_p - \cos r_{t0} \cos r_{t1}}{\sin r_{t0} \sin r_{t1}}\right)} \tag{5.2}$$

$$\boldsymbol{\gamma'_{t1} = \cos^{-1}\left(\frac{\cos r_{t0} - \cos r_p \cos r_{t1}}{\sin r_p \sin r_{t1}}\right)} \tag{5.3}$$

If $\boldsymbol{K} < 0$, then:

$$\boldsymbol{r_{t1} = \cosh^{-1}\left(\cosh r_{t0} \cosh r_p - \sinh r_{t0} \sinh r_p \cos \gamma_{t0}\right)} \tag{5.4}$$

$$\boldsymbol{\Delta\theta = \cos^{-1}\left(\frac{\cosh r_{t0} \cosh r_{t1} - \cosh r_p}{\sinh r_{t0} \sinh r_{t1}}\right)} \tag{5.5}$$

$$\boldsymbol{\gamma'_{t1} = \cos^{-1}\left(\frac{\cosh r_p \cosh r_{t1} - \cosh r_{t0}}{\sinh r_p \sinh r_{t1}}\right)} \tag{5.6}$$

Angle $\boldsymbol{\alpha}$ is the difference between object's local reference direction and its geodesic of movement ($\overline{\boldsymbol{C''_{t0}C''_{t1}}}$), which has to stay constant if the object is not rotating over time. The value is found using object's parameters at time point 0:

$$\boldsymbol{\alpha = \beta_{t0} - \gamma_{t0}} \tag{5.7}$$

Hence, at time point 1:

$$\boldsymbol{\beta_{t1} = \gamma_{t1} + \alpha} \tag{5.8}$$

$\boldsymbol{\gamma_{t1}}$ and $\boldsymbol{\gamma'_{t1}}$ are supplementary angles, so:

$$\boldsymbol{\gamma_{t1} = \pi - \gamma'_{t1}} \tag{5.9}$$

To find the $\boldsymbol{\theta}$ coordinate, either subtract or add $\boldsymbol{\Delta\theta}$ to the $\boldsymbol{\theta_c}$ depending on whether the angle $\boldsymbol{\alpha}$ or its explementary angle is used for the subsequent calculation.

5.2 Results

The movement of an object can be shown dynamically by creating several time-lapse images collated from multiple screenshots of the game screen one over the other. These are shown in the figures below. They show movement through different geodesics at different curvatures.

The software can calculate the object flying in arbitrary direction with arbitrary speed as well as starting from arbitrary position in the space. A tessellation parameter of 30 has been used to render all shapes in this section.

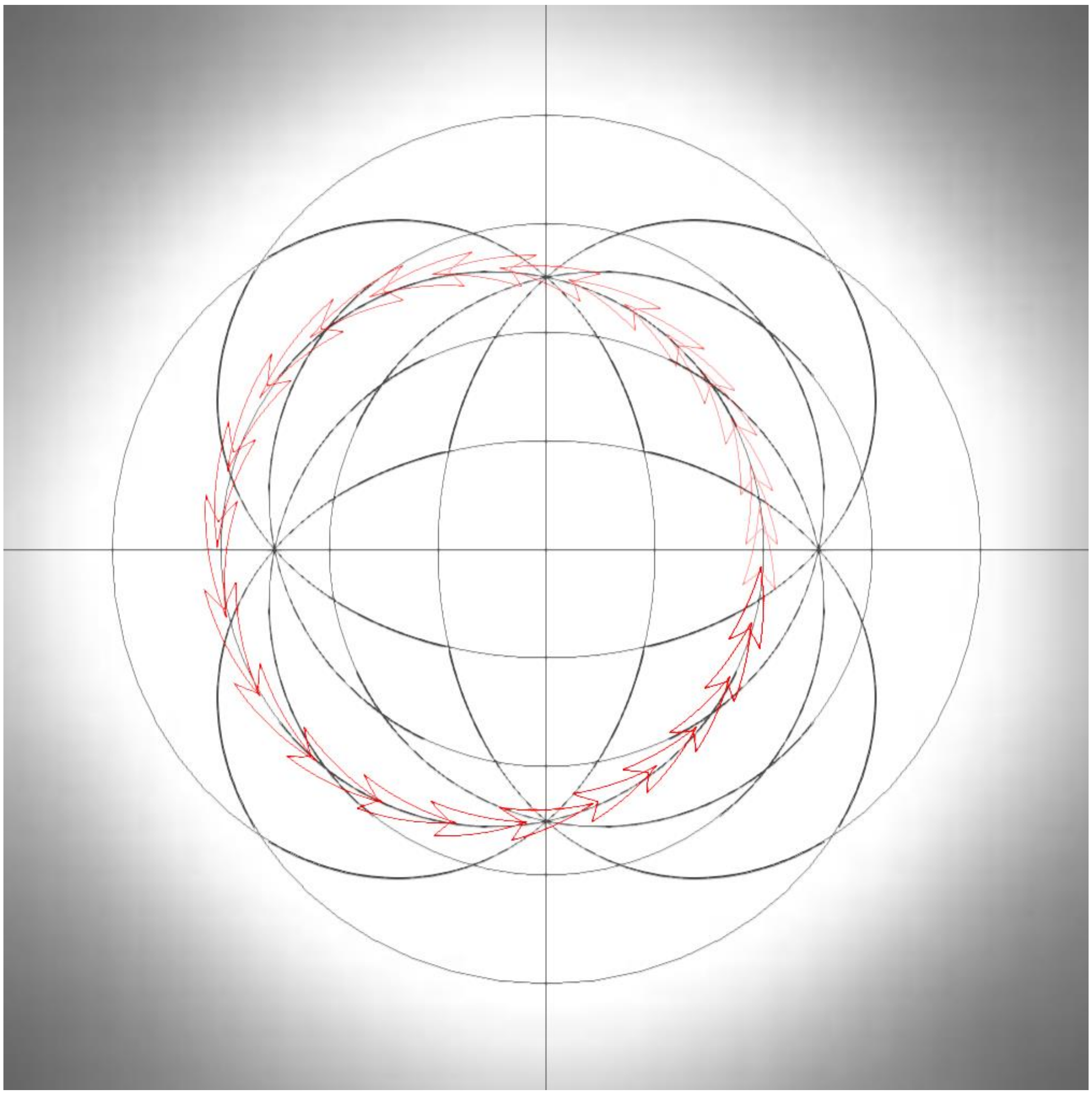


Figure 5.2: Object's movement in spherical space along the geodesic passing through the point ($\mathbf{200\ px, 0\ rad}$) at $\frac{\pi}{2}$ $\mathbf{rad}$ angle. The white circle has the radius of $\mathbf{500\ px}$.

Figure 5.2 and Figure 5.3 show object movement in spherical space ($\boldsymbol{K = 1}$). The object is a red arrow shape, it starts movement in the same position, but different direction of movement. In Figure 5.2 the object starts to move orthogonally to the horizontal axis, while in Figure 5.3, the object's starting position is rotated 45 degrees towards the origin.

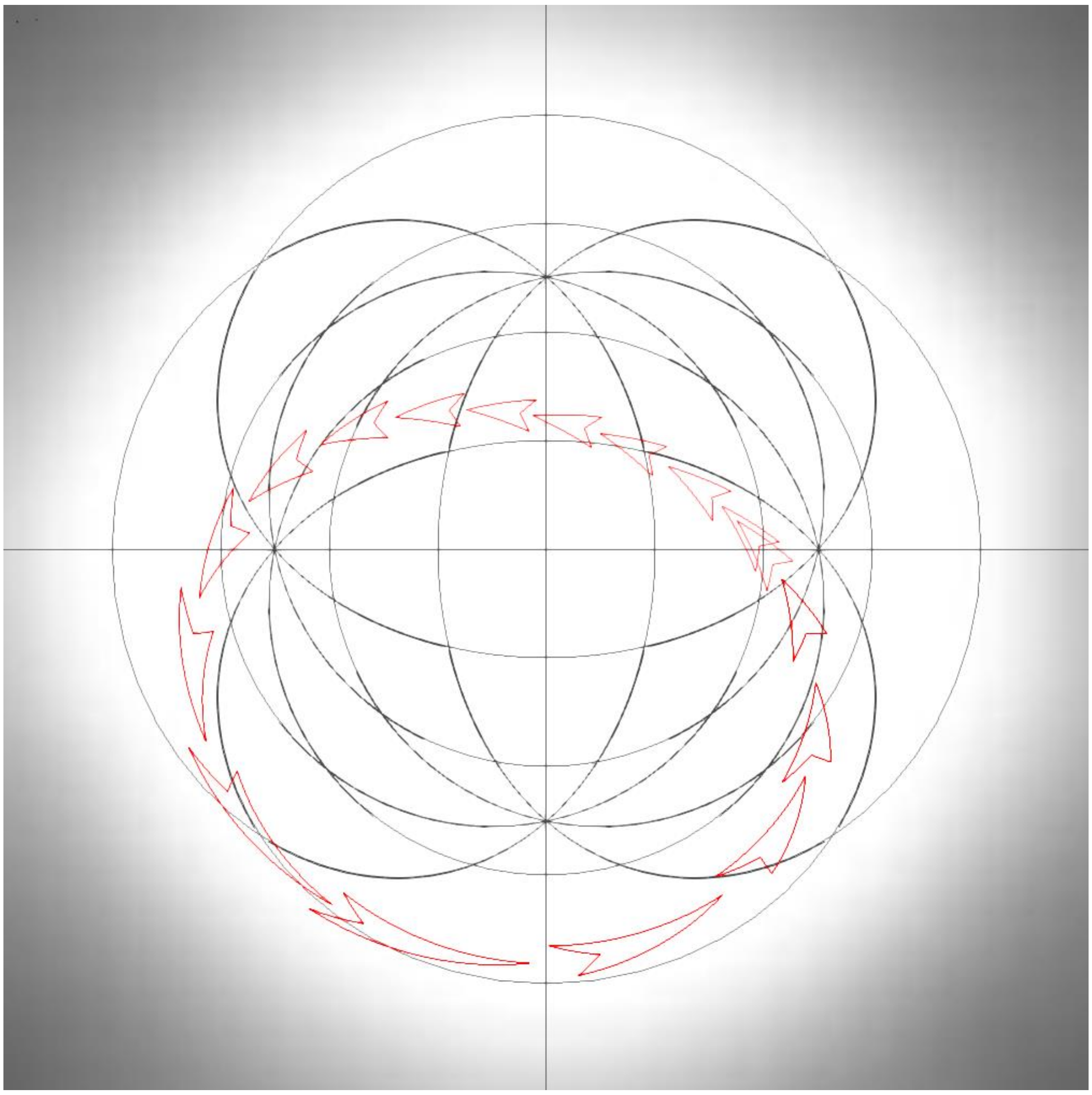


Figure 5.3: Object's movement in spherical space along the geodesic passing through the point ($\boldsymbol{200\,px, 0\,rad}$*) at* $\boldsymbol{\frac{\pi}{4}\,rad}$ *angle. The white circle has the radius of* $\boldsymbol{500\,px}$*.*

In Spherical Space, using azimuthal equidistant projection, objects can never fly off screen with continuous movement. This is because the geodesics of movement in this space are great circles and the space is finite as it folds in on itself. As such the geodesics are visible in their entirety in this projection.

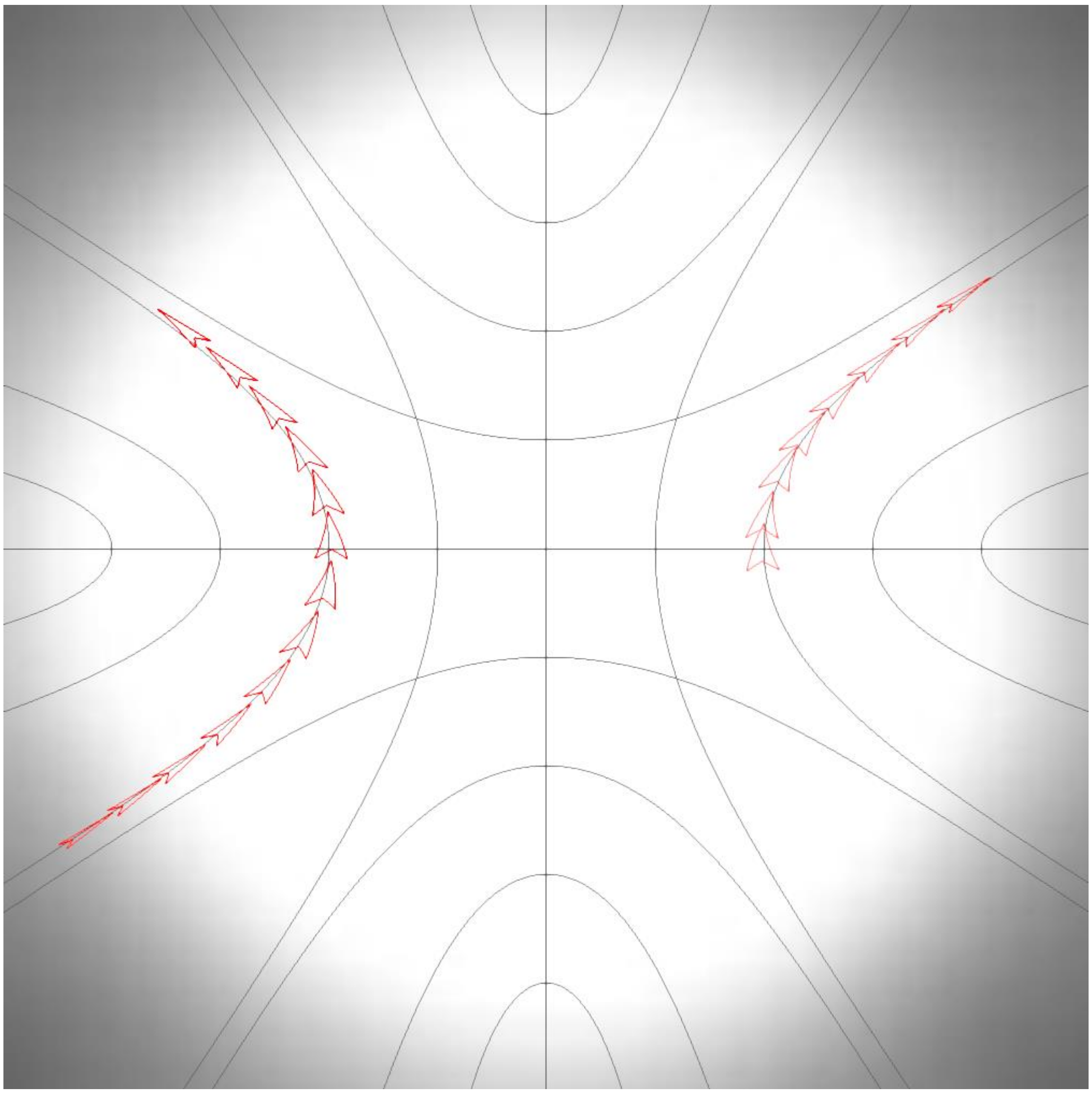


Figure 5.4: Object's movement in hyperbolic space along the geodesic passing through the point ($\boldsymbol{200\ px, 0\ rad}$) at $\boldsymbol{\frac{\pi}{2}\ rad}$ angle. The white circle has the radius of $\boldsymbol{500\ px}$.

In hyperbolic space the above does not hold, because the space is infinite, so the objects could fly off the screen. Thus, to make the space more useable for testing and rendering interactive scenes, a feature has been added to aid demonstration of the space. A cut-off distance has been implemented, which wraps the world around at a distance $\boldsymbol{r = N}$ from the centre of the screen. If the object moves further away from the origin than the specified distance, it moves onto the screen from the antipodal point at the same distance. $\boldsymbol{N}$ is set to be half the screen size used in the application; $\boldsymbol{N = 500}$ is used for all of the screenshots in this thesis. Because of the coordinate system used, it is easy to set or lift this limiting distance: the object's theta coordinate is increased by $\boldsymbol{\pi}$ and then standardised to be in range $\boldsymbol{0}$ to $\boldsymbol{2\pi}$. This makes it appear on the antipodal point of the limit circle with preserved velocity and orientation. This can be seen in the hyperbolic movement time-lapse images (Figure 5.4 and Figure 5.5). Visually, as the object crosses into the shaded area of the world, it is immediately reset on the antipodal point of the white limit circle.

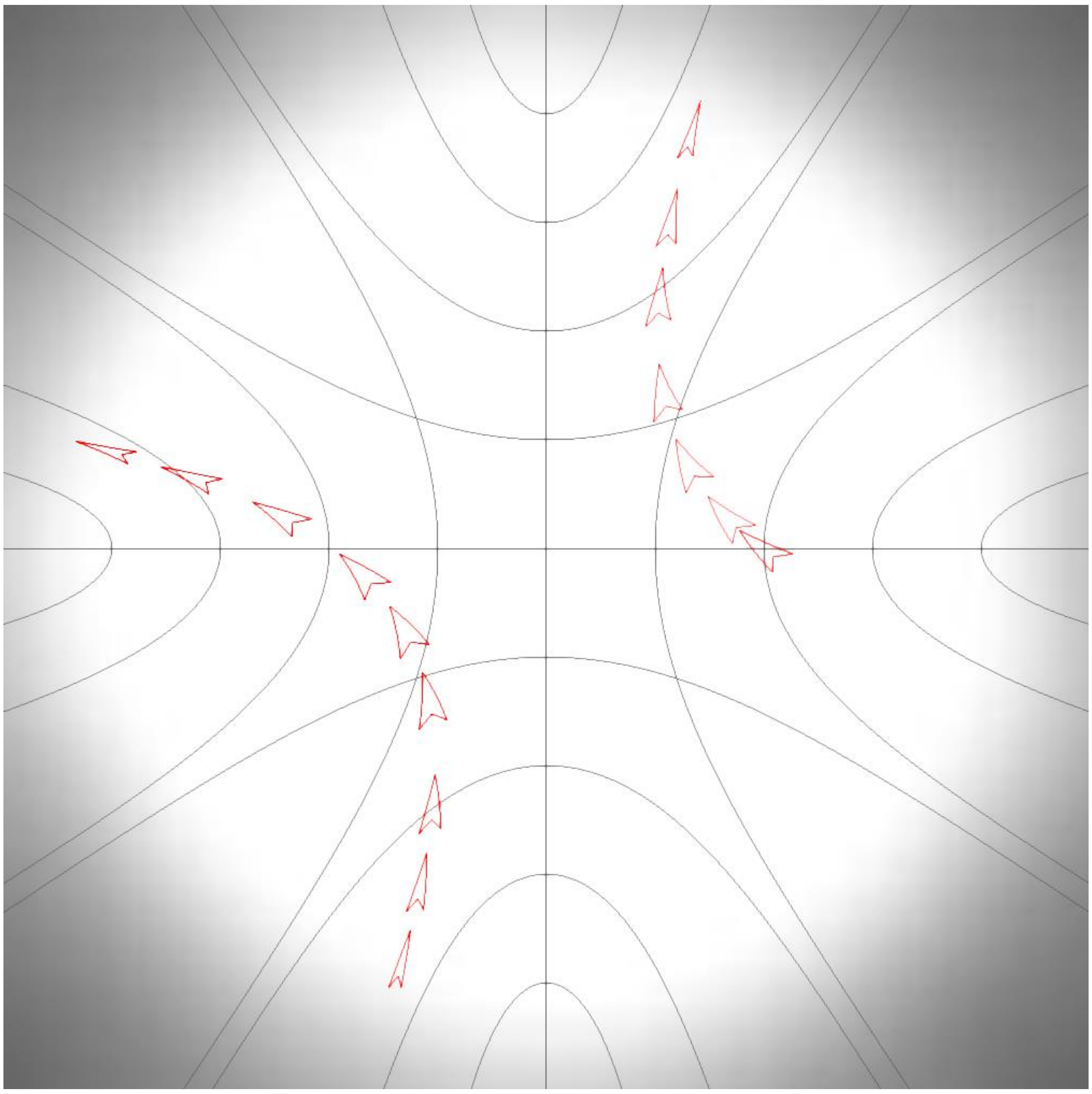


Figure 5.5: Object's movement in hyperbolic space along the geodesic passing through the point ($\boldsymbol{200\ px, 0\ rad}$) at $\frac{\boldsymbol{\pi}}{\boldsymbol{4}}$ $\boldsymbol{rad}$ angle. The white circle has the radius of $\boldsymbol{500\ px}$.

Figure 5.4 and Figure 5.5 show object movement in spherical space ($\boldsymbol{K = 1}$). The object is a red arrow shape, it starts movement in the same position, but different direction of movement. In Figure 5.4 the object starts to move orthogonally to the horizontal axis. This is the same position and initial direction as the object illustrated on Figure 5.2 in Spherical Space.

However, in Figure 5.5, the object's starting position is rotated 45 degrees towards the origin. This is the same position and initial direction as the object illustrated on Figure 5.3 in Spherical Space.

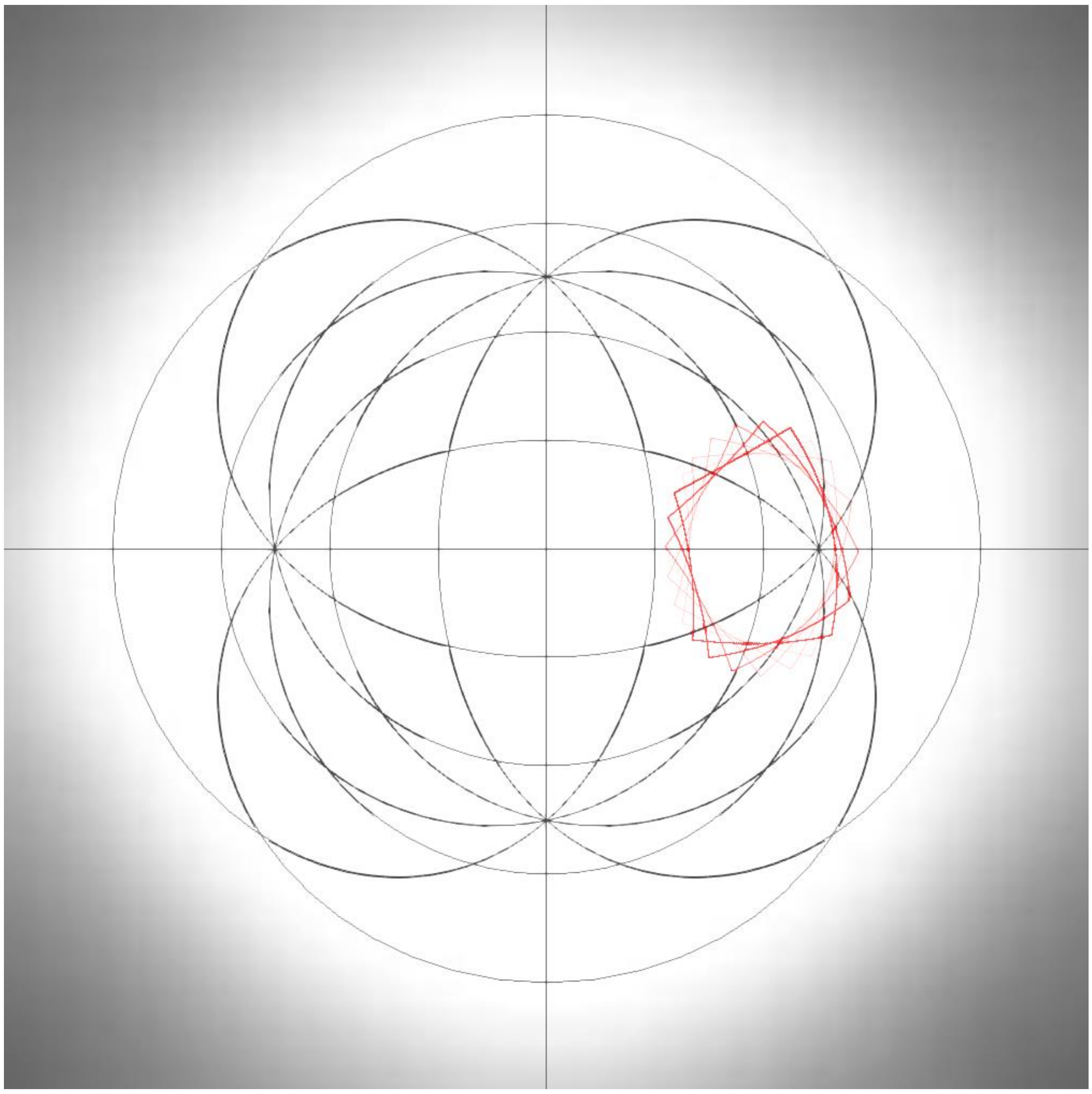


Figure 5.6: Time-lapse of the object's rotation around its centre point in spherical space ($\boldsymbol{K} = \boldsymbol{1}$). The white circle has the radius of $\boldsymbol{500\ px}$.

In addition to movement through curved space a time-lapse images of rotation of a square is shown (in this case, a square is described as a quadrilateral that has 4 vertices equidistant from the centre of the object in local coordinates as well as being equally spaced out around the local reference point). Figure 5.6 shows a rotation of this object in spherical space.

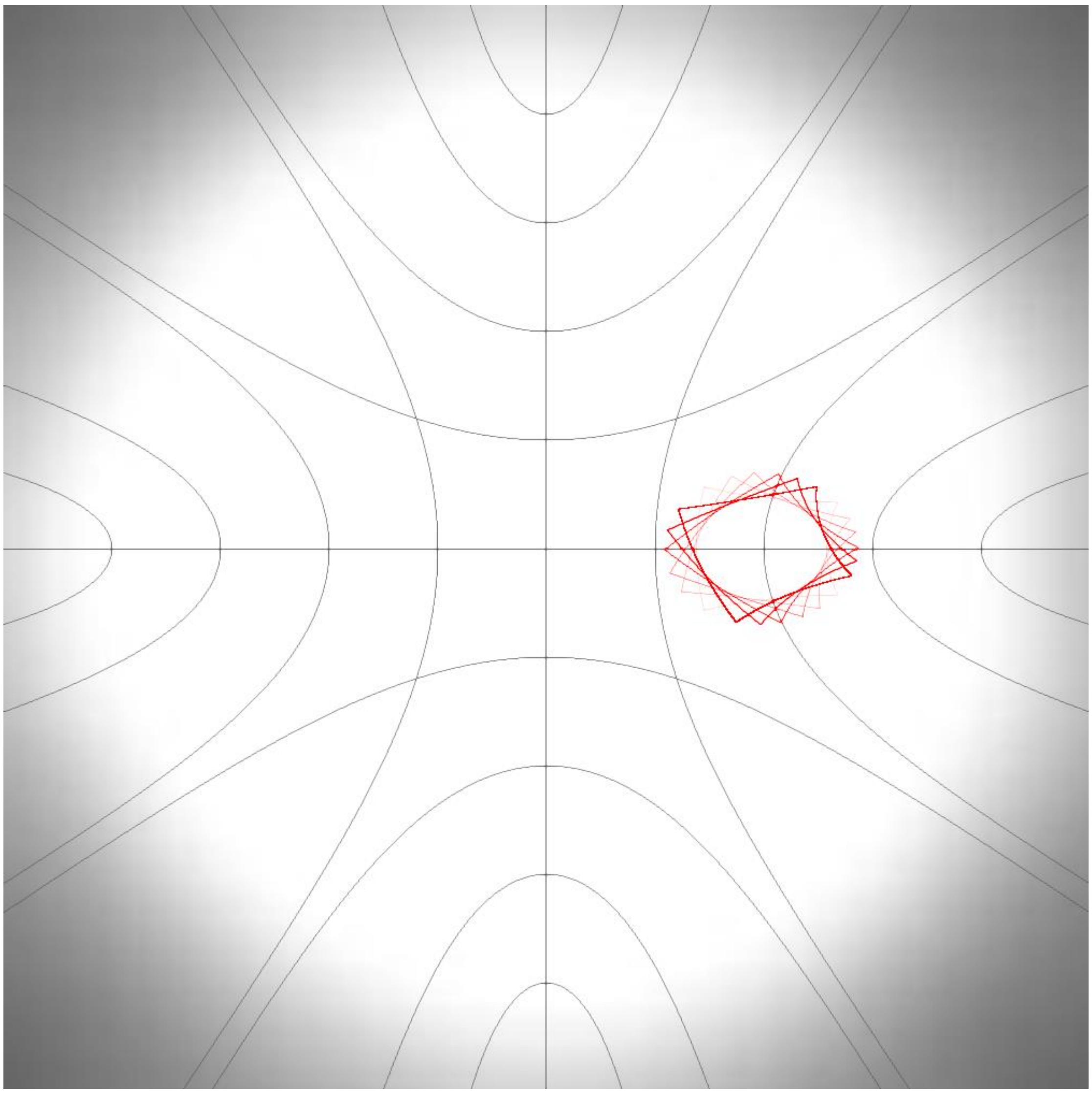


Figure 5.7: Time-lapse of the object's rotation around its centre point in hyperbolic space $(\boldsymbol{K} = -\mathbf{1})$*. The white circle has the radius of* $\mathbf{500}\ \boldsymbol{px}$*.*

Figure 5.7 shows a rotation of this object in hyperbolic space. The quadrilateral is described by the following local vertex coordinates:

$$\left[\left(\mathbf{90\ px}, \frac{\boldsymbol{\pi}}{\mathbf{4}}\ \mathbf{rad}\right), \left(\mathbf{90}\ \boldsymbol{px}, \frac{\mathbf{3}\boldsymbol{\pi}}{\mathbf{4}}\ \mathbf{rad}\right), \left(\mathbf{90}\ \boldsymbol{px}, \frac{\mathbf{5}\boldsymbol{\pi}}{\mathbf{4}}\ \mathbf{rad}\right), \left(\mathbf{90}\ \boldsymbol{px}, \frac{\mathbf{7}\boldsymbol{\pi}}{\mathbf{4}}\ \mathbf{rad}\right)\right]$$

Note that the object is described by the same position and local vertex coordinates in hyperbolic and spherical spaces. These time-lapses consist of 6 images with the square being rotated by 15 degrees between the successive images of the time-lapse.

The inner contour that the square traces, tends towards a circle. Note that the circle that would fit this contour is elliptic and is stretched when the curvature of the world is positive (shown in

Figure 5.6) and is compressed when the curvature of the world is negative (shown in Figure 5.7).

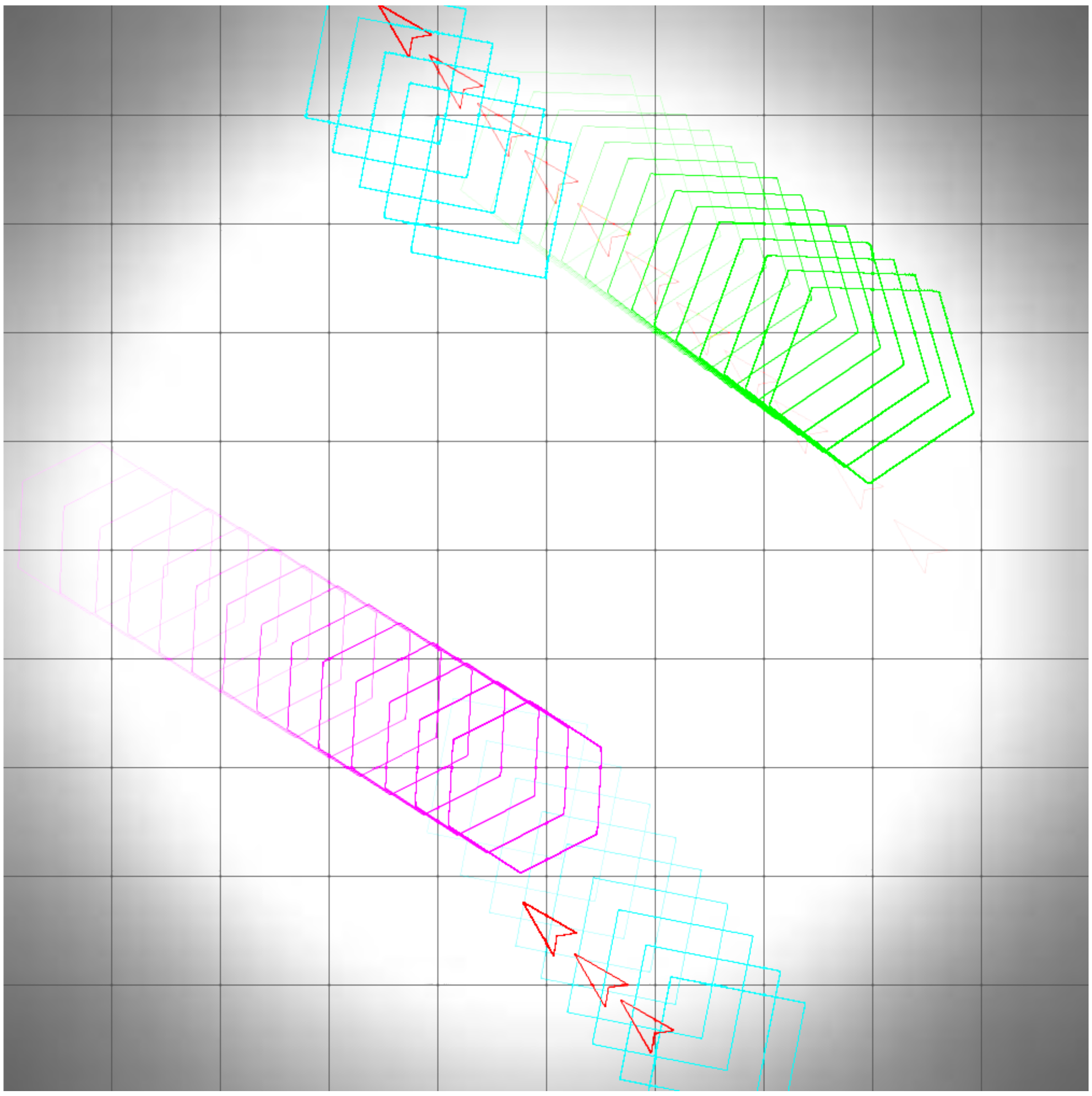


Figure 5.8: Time-lapse showing movement of multiple objects in planar space ($\boldsymbol{K = 0}$). A regular rectangle (cyan), a regular pentagon (green), a regular hexagon (magenta) and an arrow shape (red). The white circle has the radius of $\boldsymbol{500\ px}$.

Figures below show the time-lapse results of the engine rendering multiple objects in planar (Figure 5.8), spherical (Figure 5.9) and hyperbolic (Figure 5.10) geometries. The resulting engine can be built upon for multiple purposes, such as creating video games or animations in non-Euclidean environment, help visualise the mathematics of non-Euclidean space or create tools for use in different areas of research.

As shown in Figure 5.8, these time-lapse images display the movement of four different shapes: a square (or regular quadrilateral), a regular pentagon, a regular hexagon and a concave quadrilateral (or spaceship).

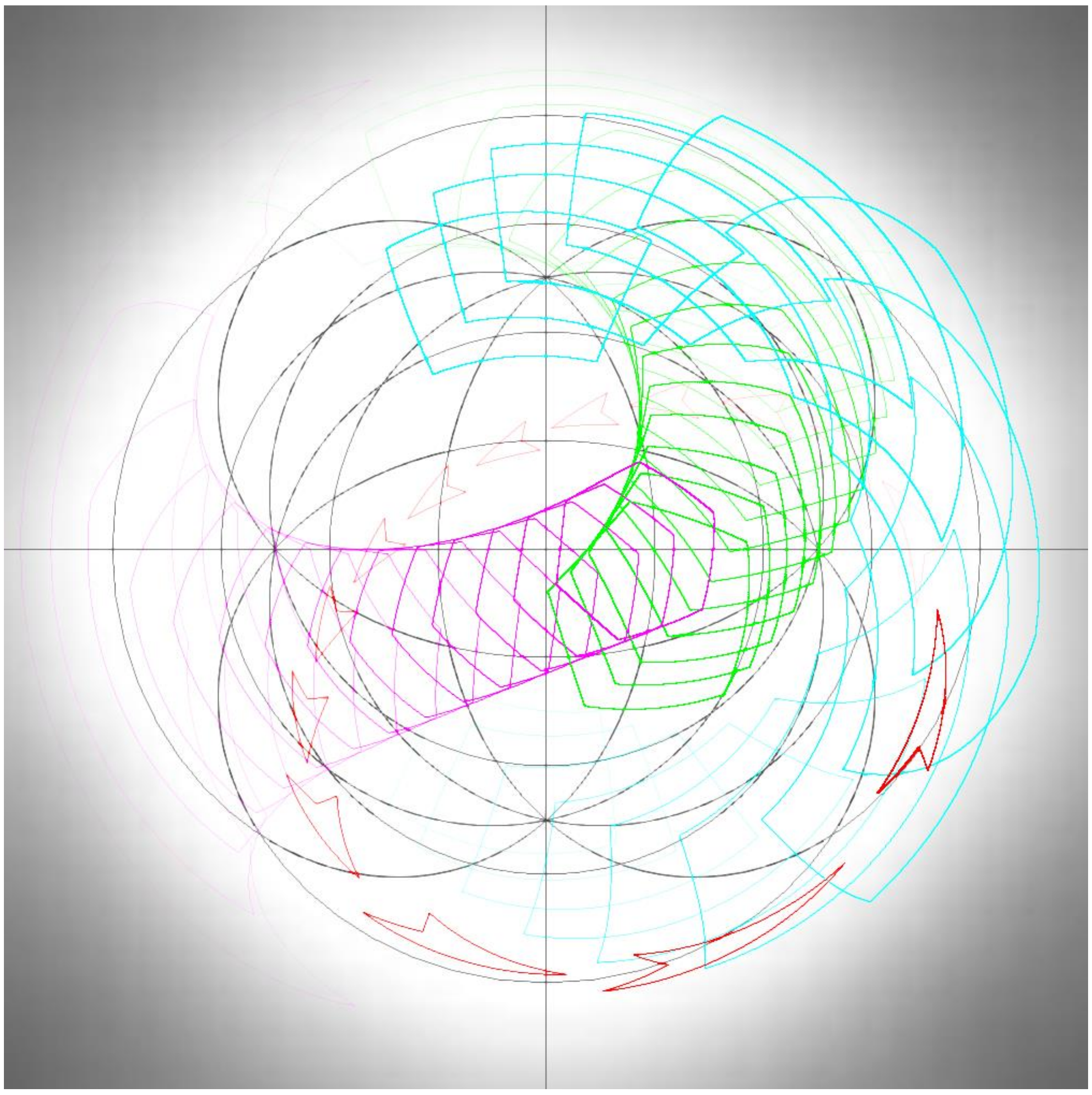


Figure 5.9: Time-lapse showing movement of multiple objects in spherical space ($\boldsymbol{K = 1}$*). A regular quadrilateral (cyan), a regular pentagon (green), a regular hexagon (magenta) and an arrow shape (red). The white circle has the radius of* $\boldsymbol{500\ px}$.

The regular polygons have different sizes set by the local r-coordinate of the respective object's vertices. Square has this property set to 45, pentagon has it set to 50 and hexagon has it set to 40. The spaceship (or the concave quadrilateral) is described by the following local vertex coordinate values:

$$\left[(\mathbf{30\ px}, \mathbf{0\ rad}), \left(\mathbf{30\ px}, \frac{\mathbf{5}\boldsymbol{\pi}}{\mathbf{6}}\ \mathbf{rad}\right), (\mathbf{15\ px}, \boldsymbol{\pi}\ \mathbf{rad}), \left(\mathbf{30\ px}, \frac{\mathbf{7}\boldsymbol{\pi}}{\mathbf{6}}\ \mathbf{rad}\right)\right]$$

The four shapes have the same starting azimuthal equidistant coordinates and the same velocities in each simulation irrespective of the curvature value. The objects have different velocities from each other: the square moves 10 units between the successive timelapse images, the pentagon moves 6 units, the hexagon moves 8 units and the spaceship moves 15 units.

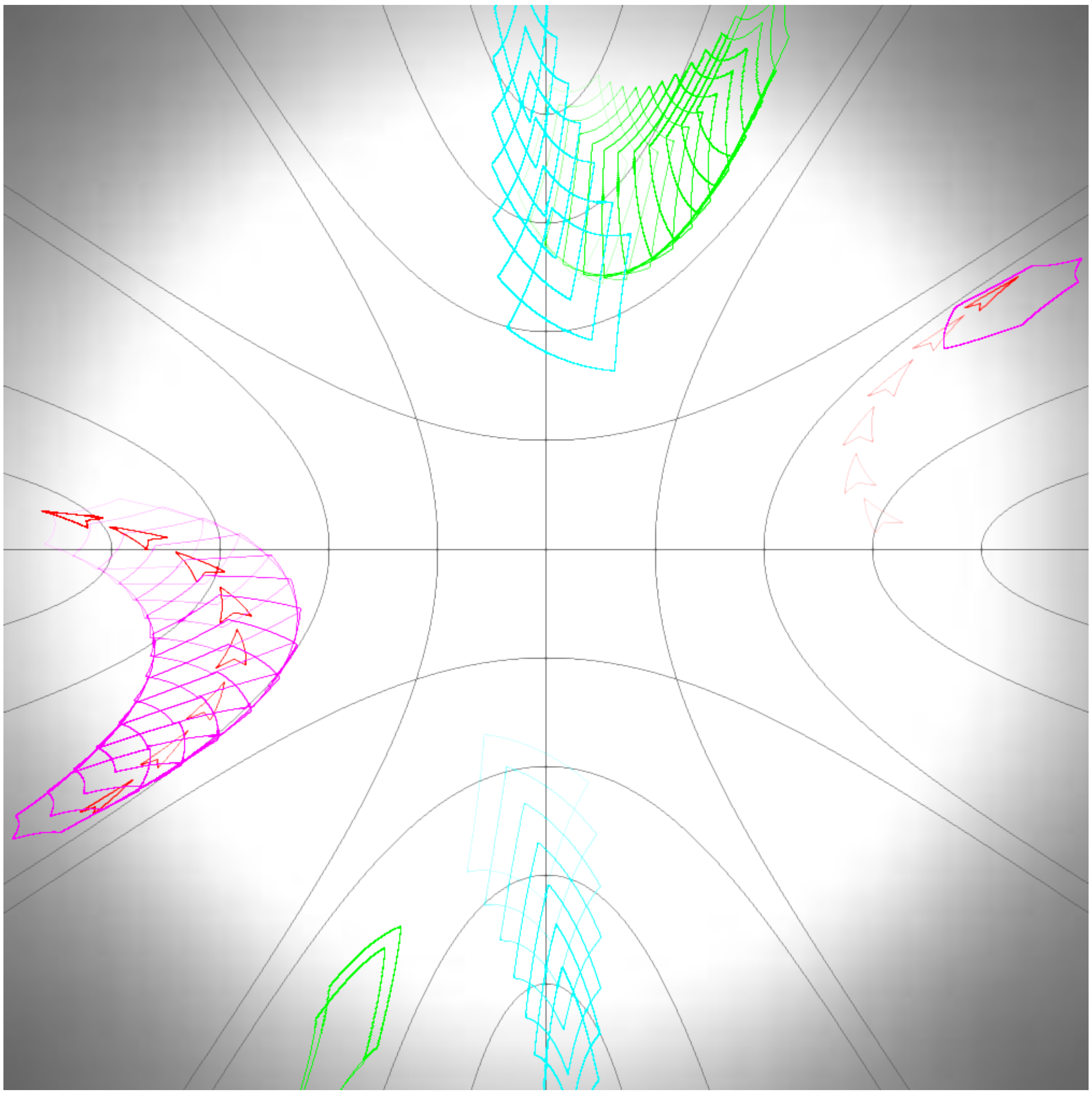


Figure 5.10: Time-lapse showing movement of multiple objects in hyperbolic space ($\boldsymbol{K = -1}$). A regular quadrilateral (cyan), a regular pentagon (green), a regular hexagon (magenta) and an arrow shape (red). The white circle has the radius of $\boldsymbol{500\ px}$.

Even though the starting positions and velocities are the same, the trajectories vary depending on the curvature of the space, due to the objects following the geodesics through the respective curved space. These geodesics are straight lines in Euclidean space, but they appear to curve towards each other in spherical space and away from each other in hyperbolic space. After 14 images of the timelapse, the shapes end the simulation in vastly different positions in space. Such simulations can help visualise and explain the nature of curved space.

Note: In Figure 5.8 and Figure 5.10 some shapes are drawn partially outside the boundary circle. This is because in Euclidean and Hyperbolic space the engine will only teleport the shapes to the antipodal point when the centre of the shape passes the boundary circle. If the world was a manifold that would wrap around, this would be incorrect behaviour. Instead this has been done by design to make the process of limiting the playable area easier programmatically.

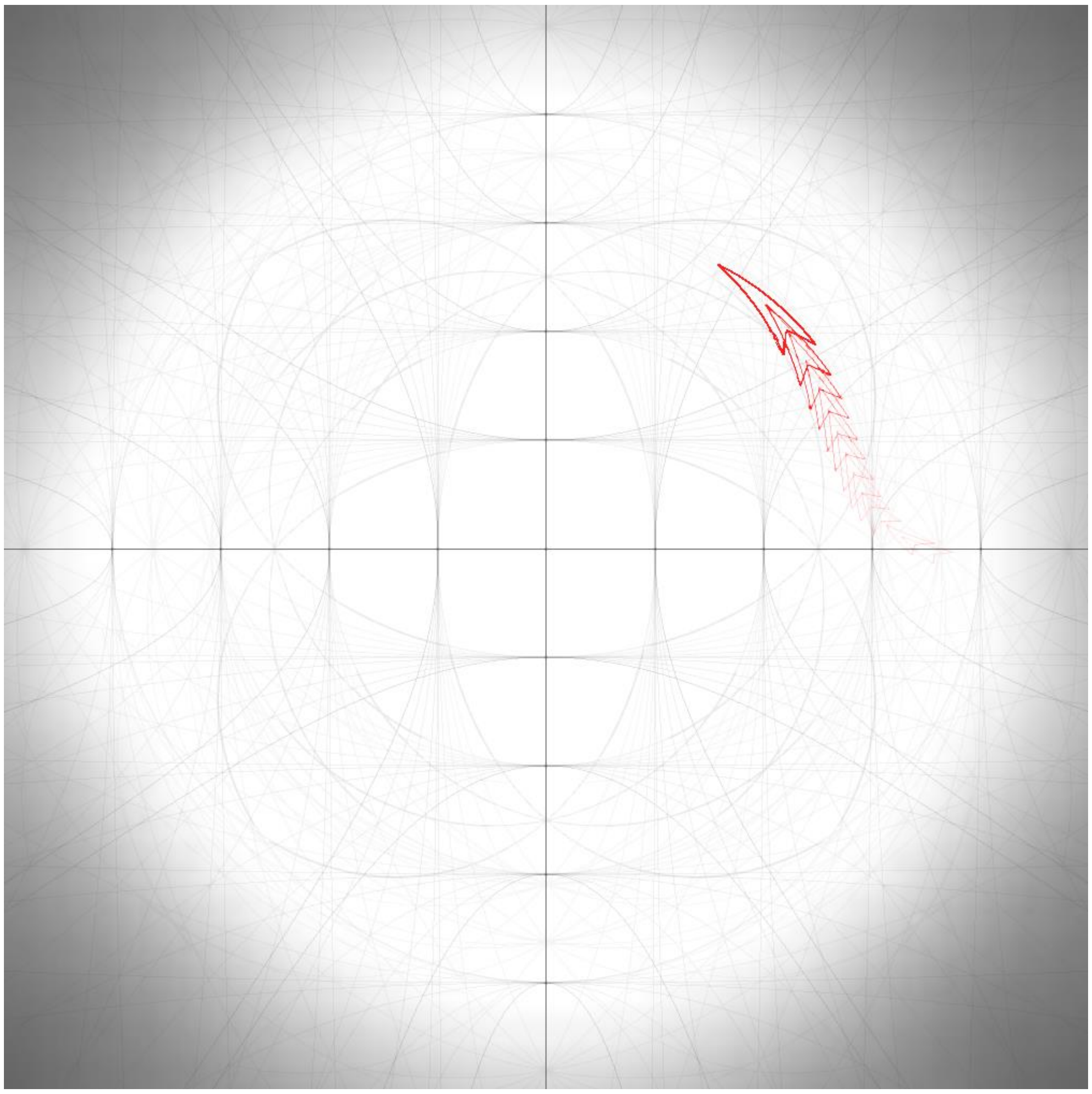


Figure 5.11: Time-lapse of a spaceship (concave quadrilateral) movement through a space with dynamically changing curvature. Curvature started at $\boldsymbol{K = -1}$ *when the first image of the time-lapse was taken and then it increased with time until its value reached* $\boldsymbol{K = 1}$, when the last image of the time-lapse was taken. *The white circle has the radius of* $\boldsymbol{500\ px}$.

The software created in this project allows for the curvature of the world to be changed in real-time using keyboard inputs in a similar manner to controlling the object's acceleration and orientation. The curvature can be switched between several pre-set values (i.e. $\mathbf{K} = \mathbf{0}$, $\mathbf{K} = \mathbf{1}$ and $\mathbf{K} = -\mathbf{1}$) or changed continuously by adding or subtracting a small number from the current curvature variable every time the corresponding button is pressed.

In the time-lapse image shown in Figure 5.11, this feature is demonstrated by starting the object's movement in hyperbolic space and then gradually increasing the value of $\mathbf{K}$ in order to progress from a hyperbolic space through flat space into spherical space. This can be observed by not only the movement of the spaceship (concave red quadrilateral), but also the change in the way the grid lines are curved.

This feature allows for flexibility in the use of the engine. The gradual change in curvature can be used to explain the concept of curvature much more easily than the standard methods of imagining the hyperbolic space or a flat projection of a sphere.

5.3 Summary

This chapter described an approach for calculating object movement through curved space in real-time using spherical and hyperbolic trigonometry. The object’s position in the next frame is calculated based on the time difference between frames, object’s previous position, velocity and trajectory. Object is moved along the geodesic of its trajectory keeping a constant orientation with respect to it. Renderings of this movement have been illustrated in this chapter to give an example of the output produced by the described software.

6. Alternative Approaches for Line Tessellation

This chapter is original research, which introduces alternative methods for tessellating shapes in non-Euclidean space. The main motivation for developing alternate tessellation approaches is to minimise the number of calculations involved in finding intermediate points along a geodesic, thus improving the performance of the application. Two different approaches have been developed for the spherical tessellation. The first one uses great circle navigation and the second uses orthogonal vectors; additionally, a single approach has been developed for hyperbolic tessellation, which uses Poincaré disc in order to find intermediate points.

Note: angle magnitudes in each diagram and equation within this section are normalised to be in the range $[\mathbf{0}, \mathbf{2\pi}]$.

6.1 Great Circle Path Approach

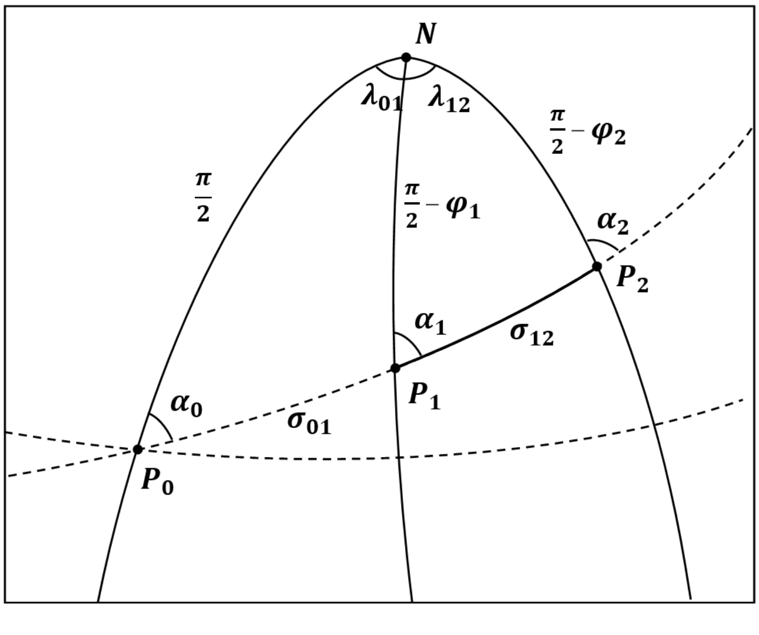


Figure 6.1: Great Circle Navigation. Finding waypoints along a great circle path between points $\boldsymbol{P_1(\phi_1, \lambda_1)}$ *and* $\boldsymbol{P_2(\phi_2, \lambda_2)}$. $\boldsymbol{N(\frac{\pi}{2}, 0)}$, *North Pole of the latitude and longitude coordinate system;* $\boldsymbol{P_0(0, \lambda_0)}$, *point of intersection of the great circle with equator geodesic;* $\boldsymbol{\alpha_0, \alpha_1, \alpha_2}$, *azimuths to the great circle path at points* $\boldsymbol{P_0, P_1, P_2}$ *respectively;* $\boldsymbol{\sigma_{01} = \overline{P_0P_1}}$; $\boldsymbol{\sigma_{12} = \overline{P_1P_2}}$; $\boldsymbol{\lambda_{01} = \lambda_1 - \lambda_0}$; $\boldsymbol{\lambda_{12} = \lambda_2 - \lambda_1}$

Finding waypoints on great circle geodesics has been used for naval and later aircraft navigation since XVII century (Cotter, 1976) (Weintrit & Neumann, 2011). Great circle navigation uses a latitude and longitude coordinate system $(\boldsymbol{\phi}, \boldsymbol{\lambda})$ instead of a polar coordinate system $(\boldsymbol{r}, \boldsymbol{\theta})$ for all calculations, so the coordinates of both points should be converted accordingly.

The theta element of the polar coordinate is angular distance from a reference direction in the range of $\mathbf{0}$ to $\mathbf{2\pi}$. Longitude is also angular distance from reference direction, but the range is $-\boldsymbol{\pi}$ to $\boldsymbol{\pi}$. $\boldsymbol{\theta}$-coordinate of the polar coordinate system is converted into longitude using:

$$\boldsymbol{\lambda = \mathrm{mod}(\theta + \pi, 2\pi) - \pi} \tag{6.1}$$

The scaled to unit sphere $\boldsymbol{r}$-coordinate of the polar coordinate system is analogous to colatitude ($\boldsymbol{\delta}$), which can be converted to latitude via:

$$\boldsymbol{\phi = \frac{\pi}{2} - \delta} \tag{6.2}$$

In order to find the points along the line $\overline{\boldsymbol{P_1 P_2}}$, the geodesic is to be extended until it crosses the equator geodesic (Figure 6.1), which has coordinates $(\mathbf{0}, \boldsymbol{\lambda})$. In particular, the coordinates of the point of intersection, $\mathbf{P_0}$, should be found. This is done by first finding the azimuth $\boldsymbol{\alpha_1}$ at $\mathbf{P_1}$ from the triangle $\mathbf{NP_1P_2}$. The tangent of the azimuth is given by:

$$\boldsymbol{\tan \alpha_1 = \frac{\cos \phi_2 \sin \lambda_{12}}{\cos \phi_1 \sin \phi_2 - \sin \phi_1 \cos \phi_2 \cos \lambda_{12}}} \tag{6.3}$$

Where $\boldsymbol{\lambda_{12}}$ is the angle between $\mathbf{P_1}$ and $\mathbf{P_2}$ at the origin of the spherical polar coordinate system (i.e. the North Pole, $\mathbf{N}$). The arctangent of the resulting value is found. This is used to find the azimuth $\boldsymbol{\alpha_0}$ at the point of intersection with the equator geodesic, $\mathbf{P_0}$. The value of its tangent is given by:

$$\boldsymbol{\tan \alpha_0 = \frac{\sin \alpha_1 \cos \phi_1}{\sqrt{\cos^2 \alpha_1 + \sin^2 \alpha_1 \sin^2 \phi_1}}} \tag{6.4}$$

As previously, the arctangent is taken. Azimuth at $\mathbf{P_0}$ is one of the preliminaries, the other is the longitude coordinate of $\mathbf{P_0}$. In order to find it, first $\boldsymbol{\sigma_{01}}$, the distance between $\mathbf{P_0}$ and $\mathbf{P_1}$ should be found. $\boldsymbol{\tan \sigma_{01}}$ is found using:

$$\tan \boldsymbol{\sigma}_{01} = \frac{\tan \boldsymbol{\phi}_1}{\cos \boldsymbol{\alpha}_1} \qquad (6.5)$$

Arctangent is subsequently calculated.

After that, the angular distance between $\mathbf{P_0}$ and $\mathbf{P_1}$ at the North Pole ($\boldsymbol{\lambda}_{01}$) should be found. Tangent is given by:

$$\tan \boldsymbol{\lambda}_{01} = \frac{\tan \boldsymbol{\alpha}_0 \sin \boldsymbol{\sigma}_{01}}{\cos \boldsymbol{\sigma}_{01}} \qquad (6.6)$$

Arctangent is calculated and $\boldsymbol{\lambda}_0$ is found by subtracting $\boldsymbol{\lambda}_{01}$ from $\boldsymbol{\lambda}_1$.

Finally, the following parametric equation is used to calculate the latitude coordinate of a point on a great circle given the two preliminaries and the longitude coordinate of the desired point. In order to parameterise the longitude coordinate $\boldsymbol{\lambda}_i$, it is incremented starting at $\boldsymbol{\lambda}_1$ and finishing with $\boldsymbol{\lambda}_1$ + $\boldsymbol{\lambda}_{12}$. The increment depends on the tessellation parameter.

$$\boldsymbol{\lambda}_i = \boldsymbol{\lambda}_1 + \frac{\boldsymbol{i}\,\boldsymbol{\lambda}_{12}}{\boldsymbol{tess\_param}} \qquad (6.7)$$

$$\tan \boldsymbol{\phi}_i = \cot \boldsymbol{\alpha}_0 \sin(\boldsymbol{\lambda}_i - \boldsymbol{\lambda}_0) \qquad (6.8)$$

Note: this formula can not be used if the geodesic passes through or close to the North Pole **N**,. This is avoided programmatically by drawing a straight line between the two vertices when the edge passes through **N**.

This process produces a series of points along the given geodesic in a geographical coordinate system. The final step is to convert them to Polar Spherical coordinates which the engine uses via:

$$\boldsymbol{\theta}_i = \mathbf{mod}(\boldsymbol{\lambda}_i, \mathbf{2}\boldsymbol{\pi}) \qquad (6.9)$$

$$\boldsymbol{r}_i = \boldsymbol{\delta}_i = \boldsymbol{\pi} - \boldsymbol{\phi}_i \qquad (6.10)$$

6.2 Orthogonal Vectors Approach

Any vector plane in vector space $\mathbf{R^3}$ can be described in terms of two orthogonal vectors (Kriz, 2010), which create a local 2D Cartesian coordinate system, illustrated on Figure 6.2. This local coordinate system is used to construct a parametric equation of the geodesic in question.

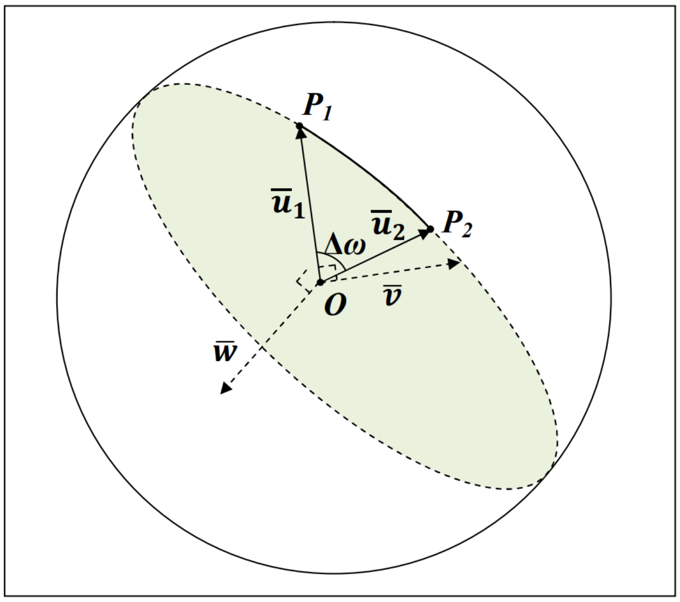


Figure 6.2: Finding the orthogonal vectors lying on the plane of a great circle given points $\boldsymbol{P_1(x_1, y_1, z_1)}$ *and* $\boldsymbol{P_2(x_2, y_2, z_2)}$ *which passes through.* $\boldsymbol{O(0,0,0)}$, *origin of the Cartesian coordinate system;* $\boldsymbol{\overline{u}_1 = \overline{OP}_1}$; $\boldsymbol{\overline{u}_2 = \overline{OP}_2}$; $\boldsymbol{\overline{v}, \overline{u}_1, \overline{u}_2 \in OP_1P_2}$; $\boldsymbol{\overline{w} \perp OP_1P_2}$; $\boldsymbol{\overline{v} \perp \overline{u}_1}$; $\boldsymbol{\Delta\omega = \angle P_1OP_2}$

First, in order to find the orthogonal vectors, for each of the two points comprising the line in question, known 2D polar coordinates have to be converted to Cartesian 3D coordinates with an origin at the centre of the sphere:

$$\boldsymbol{x = \cos\theta \sin r} \tag{6.11}$$

$$\boldsymbol{y = \sin\theta \sin r} \tag{6.12}$$

$$\boldsymbol{z = \cos r} \tag{6.13}$$

The equation of the circle can be determined by using two orthogonal vectors $\boldsymbol{\overline{u}_1}$ and $\boldsymbol{\overline{v}}$ which lie on the plane of the geodesic have to be determined. Let $\boldsymbol{\overline{u}_1}$ and $\boldsymbol{\overline{u}_2}$ be the Cartesian coordinates of points $\boldsymbol{P_1}$ and $\boldsymbol{P_2}$ respectively**.** The cross product can be taken to find $\boldsymbol{\overline{w}}$, a vector orthogonal to two known vectors, thus:

$$\overline{\boldsymbol{w}} = \overline{\boldsymbol{u}}_{\mathbf{1}} \times \overline{\boldsymbol{u}}_{\mathbf{2}} \tag{6.14}$$

This gives vector $\overline{\boldsymbol{w}}$, which has to be normalised by dividing it by its magnitude $|\overline{\boldsymbol{w}}|$. $\overline{\boldsymbol{w}}$ does not lie on the plane $\boldsymbol{OP_1P_2}$. It is in fact orthogonal to the plane, due to being orthogonal to both non parallel vectors that lie on the plane ($\overline{\boldsymbol{u}}_{\mathbf{1}}$ and $\overline{\boldsymbol{u}}_{\mathbf{2}}$). Vector $\overline{\boldsymbol{v}}$ is found by taking a cross product of $\overline{\boldsymbol{u}}_{\mathbf{1}}$ and $\overline{\boldsymbol{w}}$:

$$\overline{\boldsymbol{v}} = \overline{\boldsymbol{u}}_{\mathbf{1}} \times \overline{\boldsymbol{w}} \tag{6.15}$$

Vector $\overline{\boldsymbol{v}}$ is already normalised, because the vectors $\overline{\boldsymbol{u}}_{\mathbf{1}}$ and $\overline{\boldsymbol{w}}$ are normalised and orthogonal, so their cross product is also normalised. Then, $\boldsymbol{\Delta\omega}$ is found; it is the angle between $\boldsymbol{P_1}$ and $\boldsymbol{P_2}$ at the centre of the sphere. It is then divided into a number of parts to produce a number of angles $\boldsymbol{\Delta\omega_i}$. They are angles between $\boldsymbol{P_1}$ and a series of points $\boldsymbol{P_i}$ along the line $\overline{\boldsymbol{P_1P_2}}$:

$$\boldsymbol{\Delta\omega} = \mathbf{cos}^{-1}\frac{\overline{\boldsymbol{u}}_{\mathbf{1}} \cdot \overline{\boldsymbol{u}}_{\mathbf{2}}}{|\overline{\boldsymbol{u}}_{\mathbf{1}}||\overline{\boldsymbol{u}}_{\mathbf{2}}|} \tag{6.16}$$

$$\boldsymbol{\Delta\omega_i} = \frac{\boldsymbol{i\,\Delta\omega}}{\boldsymbol{tess\_param}} \tag{6.17}$$

Subsequently, using the vectors $\overline{\boldsymbol{u}}_{\mathbf{1}}$ and $\overline{\boldsymbol{w}}$, as well as angle $\boldsymbol{\Delta\omega_i}$, the parametric equation of the great circle on which the line lies is:

$$\boldsymbol{P_i} = \overline{\boldsymbol{u}}_{\mathbf{1}}\,\mathbf{cos}\,\boldsymbol{\Delta\omega_i} + \overline{\boldsymbol{v}}\,\mathbf{sin}\,\boldsymbol{\Delta\omega_i} \tag{6.18}$$

Finally, the coordinates of each point $\boldsymbol{P_i}$ need to be converted back to 2D polar coordinate system:

$$\boldsymbol{\theta_i} = \mathbf{tan}^{-1}\left(\frac{\boldsymbol{y_i}}{\boldsymbol{x_i}}\right) \tag{6.19}$$

$$\boldsymbol{r_i} = \mathbf{cos}^{-1}\,\boldsymbol{z_i} \tag{6.20}$$

6.3 Poincaré Disc Line Equation Approach

To find the intermediate points in this approach, the known vertex coordinates have to be converted from azimuthal equidistant projection polar coordinate system to a Poincaré disc polar coordinate system ($\mathbf{0}$ to $\mathbf{1}$ in $\boldsymbol{r}$, with $\mathbf{1}$ being an infinite distance away, and theta unchanged) (Anderson, 2006). This is achieved via:

$$\boldsymbol{r_p} = \mathbf{tanh}\left(\frac{\mathbf{r_v}}{\mathbf{2}}\right) \tag{6.21}$$

The Poincaré disc has a crucial property in that geodesics in this projection lie on circles which cross the outer edge at a right angle, illustrated on Figure 6.3. This gives a parametric equation of the geodesic of the form:

$$\boldsymbol{x^2 + y^2 + ax + by + 1 = 0} \tag{6.22}$$

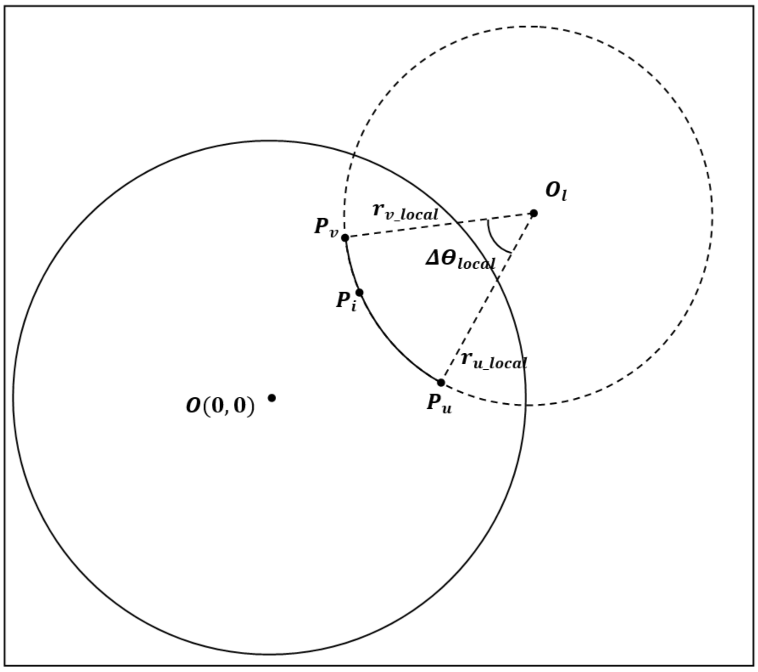


Figure 6.3: Using the Poincaré disc projection to find points between $\boldsymbol{P_v(x_v, y_v)}$ *and* $\boldsymbol{P_u(x_u, y_u)}$ *lying on a geodesic.* $\boldsymbol{O(0,0)}$*, origin of the Cartesian Poincaré coordinate system;* $\boldsymbol{O_l(x_l, y_l)}$*, origin of the local polar coordinate system;* $\boldsymbol{P_v(r_{v\_local}, \theta_{v\_local})}$ *and* $\boldsymbol{P_u(r_{u\_local}, \theta_{u\_local})}$. $\boldsymbol{\Delta\theta_{local} = \theta_{v\_local} - \theta_{v\_local}}$

In equation *(6.22)*, $\boldsymbol{a}$ *and* $\boldsymbol{b}$ are variables which can be calculated, given two points the geodesic passes through: $\boldsymbol{P_v(x_v, y_v)}$ and $\boldsymbol{P_u(x_u, y_u)}$. They can be found using:

$$\boldsymbol{a} = \frac{y_u(x_v^2 + y_v^2) - y_v(x_u^2 + y_u^2) + y_u - y_v}{x_u y_v - y_u x_v} \qquad (6.23)$$

$$\boldsymbol{b} = \frac{x_v(x_u^2 + y_u^2) - x_u(x_v^2 + y_v^2) + x_v - x_u}{x_u y_v - y_u x_v} \qquad (6.24)$$

Once the equation of the geodesic is known, several other preliminaries have to be found. The centre of the circle tracing the geodesic, $\mathbf{O_l}(\boldsymbol{x_l}, \boldsymbol{y_l})$, in Cartesian Poincaré coordinates:

$$x_l = \frac{1}{2}a \qquad (6.25)$$

$$y_l = \frac{1}{2}b \qquad (6.26)$$

The second preliminary is the radius of the circle, $\boldsymbol{r_l}$:

$$r_l = \sqrt{x_l} \qquad (6.27)$$

Intermediate points, $\mathbf{P_i}(\boldsymbol{r_{i\_local}}, \boldsymbol{\theta_{i\_local}})$, can then be found. Let $\mathbf{O_l}$ be the origin of the local polar coordinate system with local coordinates $\mathbf{O_l}(\mathbf{0}, \mathbf{0})$. Find the local coordinates for the points $\boldsymbol{P_v(r_{v\_local}, \theta_{v\_local})}$ and $\boldsymbol{P_u(r_{u\_local}, \theta_{u\_local})}$, using:

$$\tan \theta_{v\_local} = \frac{y_v - y_l}{x_v - x_l} \qquad (6.28)$$

$$\tan \theta_{u\_local} = \frac{y_u - y_l}{x_u - x_l} \qquad (6.29)$$

Local $\boldsymbol{r}$-coordinate is equal to the radius of the circle, $\boldsymbol{r_l}$, for any point lying on its circumference. The difference of the theta coordinates, $\boldsymbol{\Delta\theta_{local}}$, gives the local angular distance between $\boldsymbol{P_v}$ and $\boldsymbol{P_u}$. $\boldsymbol{\theta_{i\_local}}$ can then be found by incrementing the value of the theta coordinate from $\boldsymbol{\theta_{v\_local}}$ to $\boldsymbol{\theta_{u\_local}}$ depending on the value of tessellation parameter:

$$\boldsymbol{\theta_{i\_local} = \theta_{v\_local} + \frac{i\,\Delta\theta_{local}}{tess\_param}} \tag{6.30}$$

Note: Compared to the other methods of tessellating edges presented in this thesis, this method segments the edge using the angular distance between two vertices, rather than the actual distance along the edge. Because this is done for the circle on the Poincaré disc, this distance will not be uniform when converted to the azimuthal equidistant projection.

In order to find the global coordinates of the point $\mathbf{P_i}$, local polar coordinates of $\mathbf{P_i}$ are added to global polar coordinates of the point $\boldsymbol{O_l(r_p, \theta_p)}$, which are:

$$\boldsymbol{r_p = \sqrt{x^2 + y^2}} \tag{6.31}$$

$$\boldsymbol{\tan\theta_p = \frac{y}{x}} \tag{6.32}$$

An addition operation between two polar vectors is calculated as follows:

$$\boldsymbol{r_i = \sqrt{r_{i\_local}^2 + r_p^2 + 2r_{i\_local}r_p\cos(\theta_p - \theta_{i\_local})}} \tag{6.33}$$

$$\boldsymbol{\theta_i = \theta_{i\_local} + \tan^{-1}\left(\frac{r_p\sin(\theta_p - \theta_{i\_local})}{r_{i\_local} + r_p\cos(\theta_p - \theta_{i\_local})}\right)} \tag{6.34}$$

The result is the global coordinates of the intermediate point on a Poincaré disc. The last step is to convert these back to azimuthal equidistant projection by performing the reverse of the scaling applied in equation *(6.21)*, $\boldsymbol{\theta_v = \theta_p}$, while $\boldsymbol{r_v}$ is found using:

$$\boldsymbol{r_v = 2\tanh^{-1} r_i} \tag{6.35}$$

6.4 Summary

This chapter has described three alternative approaches for tessellating edges of 2D shapes in non-Eulidean geometry. Two of these (i.e., Great Circle Path Approach and Orthogonal Vectors approach) were developed for spherical geometry and one (i.e. Poincare Disc Line Equation approach), has been developed for hyperbolic geometry. The performance of these approaches is compared to performance of spherical and hyperbolic trigonometry approaches in chapter 8.

7. Method III: Parallelisation of the Line Rendering Approach

This chapter is original research, which introduces a method of parallelisation of line rendering approaches described in chapters 4 and 6. To accomplish this, the methods have been modified to work within the GLSL shader pipeline to perform the shape rendering on a GPU.

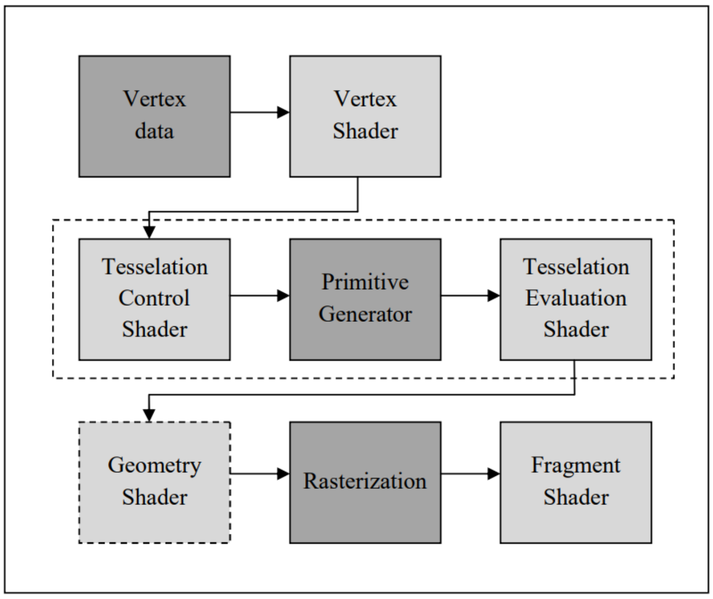


Figure 7.1: GLSL shader Pipeline. Allows for a direct control over 5 steps of the pipeline (shaded light grey). Vertex, Tessellation Control and Tessellation Evaluation shaders have been used for the project. Modifiable steps are shaded light grey, while non-modifiable are shaded dark grey. Optional steps in the pipeline are shaded light grey.

In order to achieve a better performance rendering the non-Euclidean environment, approaches can be parallelised and certain sections can be calculated simultaneously on a GPU. GPU rendering pipeline takes a set of instructions and runs them on a GPU rather than the CPU. GPUs are designed for parallel execution of a single task, they run the given set of instructions on a large number of cores, which significantly speeds up rendering process. The program only communicates with the GPU at the beginning and at the end of the pipeline.

Another use of the GPU is the GPGPU (general purpose computing on graphics processing units), which uses GPU to execute tasks outside the scope of computer graphics. However this research project will not be using GPGPU.

For this research project, GLSL was used, a shading language that offers direct control over 5 steps of the graphics pipeline (Kessenich, Baldwin, & Rost, 2017), shaded light grey in Figure

7.1. The other three steps, shaded dark grey, are set and cannot be modified. Only 3 steps will be used in order to improve the rendering time: Vertex, Tessellations Control and Tessellations Evaluation shaders.

7.1 Shader Pseudocode

Pseudocode will be used to show how the aforementioned approaches were split between the shaders. The following parameters are passed onto the shader when an object is being rendered: $\mathbf{k}$, $\mathbf{r_c}$, $\boldsymbol{\theta_c}$, $\boldsymbol{\alpha}$, tessellation parameter and vertex array of $\mathbf{r_{local}}$ and $\boldsymbol{\theta_{local}}$ coordinates, described in section 4.2.

7.1.1 Vertex Shader

In the vertex shader, spherical or hyperbolic trigonometry is used to find the position of each of the shape's vertices in the global polar coordinate system depending on the value of $\mathbf{K}$. Because calculation of the global position of the vertex depends only on that vertex's local coordinates and the global coordinates/rotation of the shape, each vertex can be computed simultaneously (depending on the capacity of the machine's GPU).

Note: Vertex shader does not change depending on which tessellation approach is used. Global vertex coordinates are found in the same way for trigonometry based approaches described in chapter 4 and alternative approaches described in chapter 6.

```
inputs: k, r_c, theta_c, alpha, vertex_array[r_local, theta_local]


initialise r_v, theta_v, delta_theta, beta;
initialise direction = 1
r_c = r_c / k
r_local = r_local / k
beta = r_local + alpha
if beta > pi
     direction = -1
     beta = 2 * pi - beta
if K > 0
     if beta > pi / 60
          use equation (4.3) to find r_v
          use equation (4.4) to find delta_theta
     else
          use equation (4.7) to find r_v
          use equation (4.8) to find delta_theta
else if K < 0
     use equation (4.5) to find r_v
     use equation (4.6) to find delta_theta
delta_theta = delta_theta * direction
theta_v = theta_c + delta_theta


outputs: cos_r_v, sin_r_v, cosh_r_v, sinh_r_v, r_v, theta_v
```

7.1.2 Tessellation Control Shader (Trigonometry Approach)

The shape is separated into patches of 2 vertices each, so that each edge of the shape is processed separately. For each edge, the tessellation shaders will produce a specified number of intermediate points which lie on the geodesic connecting the two vertices.

Preliminaries for the calculation of intermediate points are found in the tessellation control shader, and the parameters are passed onwards to the tessellation evaluation shader.

Using the trigonometry approach, the preliminaries (i.e., the length of the edge, **d**, and the angle between the edge and vertex 1, $\boldsymbol{\alpha}$ and the direction of the edge compared to the first vertex) are found following the method described in section 4.3.

```
inputs:   r_c,   theta_c,   tessellation_parameter,   array[cos_r_v],
array[sin_r_v], array[cosh_r_v], array[sinh_r_v],
vertex_array[r_v, theta_v]


initialise alpha, d
initialise direction = 1
initialise delta_theta = |theta_1 - theta_2|
if K > 0
     if delta_theta > pi / 10
          use equation (4.13) to find d
          use equation (4.15) to find alpha
     else
          use equation (4.17) to find d
          use equation (4.18) to find alpha
else if K < 0
     use equation (4.14) to find d
     use equation (4.16) to find alpha
if the triangles are converging
     direction = -1


outputs: d, alpha, cos_alpha, delta_theta, direction, array[sin_r],
array[cos_r], array[sinh_r], array[cosh_r],
vertex_array[r_v, theta_v]
```

7.1.3 Tessellation Evaluation Shader (Trigonometry Approach)

A number of intermediate point coordinates, equal to the tessellation parameter, is found following the method described in section 0.

```
inputs: d, alpha, cos_alpha, delta_theta, direction,
array[cos_r_v], array[sin_r_v], array[cosh_r_v], array[sinh_r_v],
vertex_array[r_v, theta_v]


initialise r_i, theta_i, delta_theta_i
initialise d_i = tesselation_coordinate * d
if K > 0
     if alpha > pi / 10
          use equation (4.19) to find r_i
          use equation (4.20) to find delta_theta_i
     else
          use equation (4.23) to find r_i
          use equation (4.24) to find delta_theta_i
else if K < 0
     use equation (4.21) to find r_i
     use equation (4.22) to find delta_theta_i
delta_theta_i = delta_theta_i * direction
theta_i = theta_1 + delta_theta_i
r_i = r_i * k
x_i = r_i * cos(theta_i)
y_i = r_i * sin(theta_i)


outputs: x_i, y_i
```

7.1.4 Tessellation Control Shader (Great Circle Path Approach)

Using the great circle path approach, the preliminaries (i.e., the angular distance between the vertices at the North Pole, $\boldsymbol{\lambda_{12}}$, longitude coordinate of the intersection of the edge with the equator, $\boldsymbol{\lambda_0}$, and cosine of the azimuth at the point of intersection, $\mathbf{cos}\,\boldsymbol{\alpha_0}$) are found following the method described in section 6.1.

```
inputs: r_c, theta_c, tessellation_parameter,
vertex_array[r_v, theta_v]


initialise phi_1, lambda_1, phi_2, lambda_2
use equation (6.1) to find lambda_1 and lambda_2
use equation (6.2) to find phi_1 and phi_2
initialise alpha_1, alpha_0, sigma_01, lambda_01, cot_alpha_0
initialise lambda_12 = |lambda_1 - lambda_2|
use equation (6.3) to find alpha_1
use equation (6.4) to find alpha_0
cot_alpha_0 = cot(alpha_0)
use equation (6.5) to find sigma_01
use equation (6.6) to find lambda_01
lambda_0 = |lambda_1 - lambda_01|


outputs: lambda_0, lambda_1, cot_alpha_0, lambda_12
```

7.1.5 Tessellation Evaluation Shader (Great Circle Path Approach)

A number of intermediate point longitude and latitude coordinates, equal to the tessellation parameter, is found following the method described in section 6.1. These are subsequently converted into polar coordinates of the engine.

```
inputs: k, lambda_0, lambda_1, cot_alpha_0, lambda_12


initialise phi_i, r_i, theta_i
increment = tessellation_coordinate * lambda_12
initialise lambda_i = lambda_01 + increment
use equation (6.8) to find phi_i
use equation (6.9) to find theta_i
use equation (6.10) to find r_i
r_i = r_i * k
x_i = r_i * cos(theta_i)
y_i = r_i * sin(theta_i)


outputs: x_i, y_i
```

7.1.6 Tessellation Control Shader (Orthogonal Vectors Approach)

Using the orthogonal vectors approach, the preliminaries (i.e., $\overline{\boldsymbol{u}}$ and $\overline{\boldsymbol{v}}$, orthogonal vectors which lie on the plane of the geodesic between the vertices) are found via the approach described in section 6.2.

```
inputs: r_c, theta_c, tessellation_parameter,
vertex_array[r_v, theta_v]


initialise P_1(x_1, y_1, z_1)
initialise P_2(x_2, y_2, z_2)
use equation (6.11) to find x_1 and x_2
use equation (6.12) to find y_1 and y_2
use equation (6.13) to find z_1 and z_2
initialise u, w, v
u = P_1 / |P_1|
w = cross(u, P_2)
w = w / |w|
v = cross(u, w)
initialise delta_omega
delta_omega = acos(dot(P_1, P_2) / (|P_1| * |P_2|))


outputs: v, u, delta_omega
```

Note: if $\boldsymbol{\Delta\omega}$ is small, $\mathbf{cos^{-1}}$ function can be imprecise, instead, the following formula can be used:

$$\boldsymbol{\Delta\omega} = \mathbf{tan^{-1}}\left(\frac{|\boldsymbol{P_1} \times \boldsymbol{P_2}|}{\boldsymbol{P_1} \cdot \boldsymbol{P_2}}\right) \qquad (7.1)$$

7.1.7 Tessellation Evaluation Shader (Orthogonal Vectors Approach)

A number of intermediate point 3D Cartesian coordinates, equal to the tessellation parameter, are found following the method described in section 6.2. These are subsequently converted into the polar coordinates of the engine.

```
inputs: k, u, v, delta_omega


initialise P_i(x_i, y_i, z_i)
initialise r_i, theta_1
initialise delta_omega_i = tesselation_coordinate * delta_omega
use equation (6.18) to find P_i
use equation (6.19) to find theta_i
use equation (6.20) to find r_i
r_i = r_i * k
x_i = r_i * cos(theta_i)
y_i = r_i * sin(theta_i)


outputs: x_i, y_i
```

7.1.8 Tessellation Control Shader (Poincaré Disc Line Equation Approach)

Using the Poincaré disc geodesic equation approach, the preliminaries (i.e. global coordinates of the centre of the circle, which traces the geodesic, $\boldsymbol{O_l(r_l, \theta_l)}$; radius of this circle, $\boldsymbol{r_{local}}$, and angular distance between the vertices at the centre of this circle, $\boldsymbol{\Delta\theta_{local}}$) are found following the approach described in section 6.3.

```
inputs: r_c, theta_c, tessellation_parameter,
vertex_array[r_v, theta_v]


initialise a, b, r_l, theta_u, theta_v
use equation (6.23) to find a
use equation (6.24) to find b
initialise O_l (x_l, y_l)
use equations (6.25) and (6.26) to find O_l
use equation (6.27) to find r_l
use equation (6.28) to find theta_v
use equation (6.29) to find theta_u
initialise delta_theta = |theta_v - theta_u|


outputs: O_l, r_l, delta_theta
```

7.1.9 Tessellations Evaluation Shader (Poincaré Disc Line Equation Approach)

A number of intermediate point global polar coordinates, equal to the tessellation parameter, are calculated via the method described in section 6.3.

```
inputs: k, O_l, r_l, delta_theta


initialise P_i(x_i, y_i)
initialise theta_l = theta_v + tesselation_coordinate * delta_theta
initialise P_l(r_l, theta_l)
P_i = P_l + O_l
initialise r_p, r_i, theta_p
use equation (6.31) to find r_p
use equation (6.32) to find theta_p
use equation (6.35) to find r_i
r_i = r_i * k
x_i = r_i * cos(theta_i)
y_i = r_i * sin(theta_i)


outputs: x_i, y_i
```

7.2 Summary

This chapter has described a way to improve performance of the non-Euclidean engine by parallelising the calculations using the GPU. Approaches described in chapter 4 and 6 have been split and adapted to work as part of a GLSL shader pipeline. Global vertex coordinates are found in the vertex shader given the local coordinate of the respective vertex as well as object's position and rotation parameters. Preliminaries for edge tessellation are subsequently found in the Tessellation Control Shader. Finally, the intermediate points along each of the object's edges are found in the Tessellation Evaluation Shader. The performance comparison between GPU and non-GPU approaches is detailed in chapter 8.

8. Analysis I: Line Rendering Performance

This chapter is original research, which describes the theoretical comparison of line rendering approach efficiency. It then presents an empirical test performed to further compare the approaches and analyses the data gathered from this test.

8.1 Theoretical

First, the number of calculations for each approach is calculated in order to compare their theoretical efficiency. The overall structure of each approach is the same. It is split into 3 parts: calculating global vertex coordinates, preliminaries for tessellation and intermediate point coordinates. Calculating the global vertex coordinates is done once per vertex, so the number of calculations at this step increases linearly as the total number of vertices increases (number of shapes multiplied by the number of vertices they have). Each approach uses trigonometry (spherical or hyperbolic, depending on value of $\boldsymbol{K}$) to compute the vertex coordinates, so the number of per-vertex calculations is identical for these. Next, the calculation of preliminaries for tessellation is done once for every edge the shape has, which for 2D shapes is equal to once for every consecutive pair of vertices. For the objects that are a loop, this amount is equal to the number of vertices the shape has. Lastly, the tessellation parameter specifies the number of intermediate points, the coordinates of which will be calculated. Thus, this section of the code is executed once per intermediate point and is also repeated by the number of edges.

Thus, no matter the approach the general complexity of the calculations is:

$$\boldsymbol{n_s} \times \boldsymbol{n_v} \times \boldsymbol{n_i}$$

Where $\boldsymbol{n_s}$ is the number of shapes in the world; $\boldsymbol{n_v}$ is the number of vertices the shapes have; $\boldsymbol{n_i}$ is the number of intermediate points for the tessellated lines. In the worst case this is equivalent to cubic time complexity, $\boldsymbol{O}(\boldsymbol{N^3})$ using the Big O notation (Bachmann, 1894), where $\boldsymbol{N} = \boldsymbol{max}(\boldsymbol{n_s}, \boldsymbol{n_v}, \boldsymbol{n_i})$. The number of shapes, vertices and intermediate points will rarely be equally large, so the overall complexity will heavily depend on one of these three variables.

Number of per-vertex calculations for the first step:

Spherical Trigonometry	
Trigonometric functions:	8
Algebraic operations:	14

Hyperbolic Trigonometry	
Trigonometric functions:	8
Algebraic operations:	14

These tables show how many mathematical operations each of the methods needs. They are split into more computationally expensive trigonometric functions and less costly algebraic operation. Comparing the trigonometric approaches and parametric equation approaches, both have the same number of calculations in the first step, as trigonometry is used to find the vertex coordinates.

Number of per-edge calculations for the second step:

Spherical Trigonometry	
Trigonometric functions:	3
Algebraic operations:	17

Hyperbolic Trigonometry	
Trigonometric functions:	3
Algebraic operations:	16

Great Circle Navigation	
Trigonometric functions:	16
Algebraic operations:	33

Orthogonal Vectors	
Trigonometric functions:	15
Algebraic operations:	37

Poincaré Disk Circle Equation	
Trigonometric functions:	9
Algebraic operations:	55

Number of per-intermediate point calculations for the third step:

Spherical Trigonometry	
Trigonometric functions:	5
Algebraic operations:	12

Hyperbolic Trigonometry	
Trigonometric functions:	5
Algebraic operations:	12

Great Circle Navigation	
Trigonometric functions:	2
Algebraic operations:	10

Orthogonal Vectors	
Trigonometric functions:	4
Algebraic operations:	7

Poincaré Disk Circle Equation	
Trigonometric functions:	4
Algebraic operations:	17

Parametric equation approaches have more calculations in the second step, but less expensive calculations in the third step. This would mean that if the number of times each step is executed was equal, the trigonometry approaches would be faster. The third step is executed multiple times per edge, while the second is only executed once, which means that the higher the tessellation parameter, the more efficient the parametric equation approaches are.

Figure 8.2 and Figure 8.3 show the comparative speed of the tessellation approaches. To compare the approaches, benchmark tests (shown in Figure 8.1) have been run on the Personal Computer (with specifications described in the next subsection), which empirically determined the comparative cost of each mathematical operation. These have then be scaled to the least expensive operation, addition, which has been taken as an operation unit for this comparison. Then, the comparative cost of each approach (in op units) has been calculated as such:

$$\boldsymbol{C_t} = \sum \boldsymbol{n C_o} \qquad (8.1)$$

operation	ms for test	op units
plus	0.189	1.000
minus	0.198	1.047
mult	0.199	1.051
div	0.435	2.302
sqrt	0.446	2.361
sin	1.780	9.416
asin	1.805	9.546
cos	1.834	9.699
acos	1.813	9.589
tan	2.333	12.343
atan	2.409	12.741
sinh	4.857	25.689
asinh	3.324	17.585
cosh	4.890	25.866
acosh	11.875	62.813
tanh	5.044	26.682
atanh	2.734	14.460
exp	1.195	6.322
fmod	2.466	13.045
abs	0.246	1.303

Figure 8.1: benchmark tests results for a range of mathematical operations and trigonometric functions in ms per 100,000,000 operations and operation units (taking plus operation as a single unit)

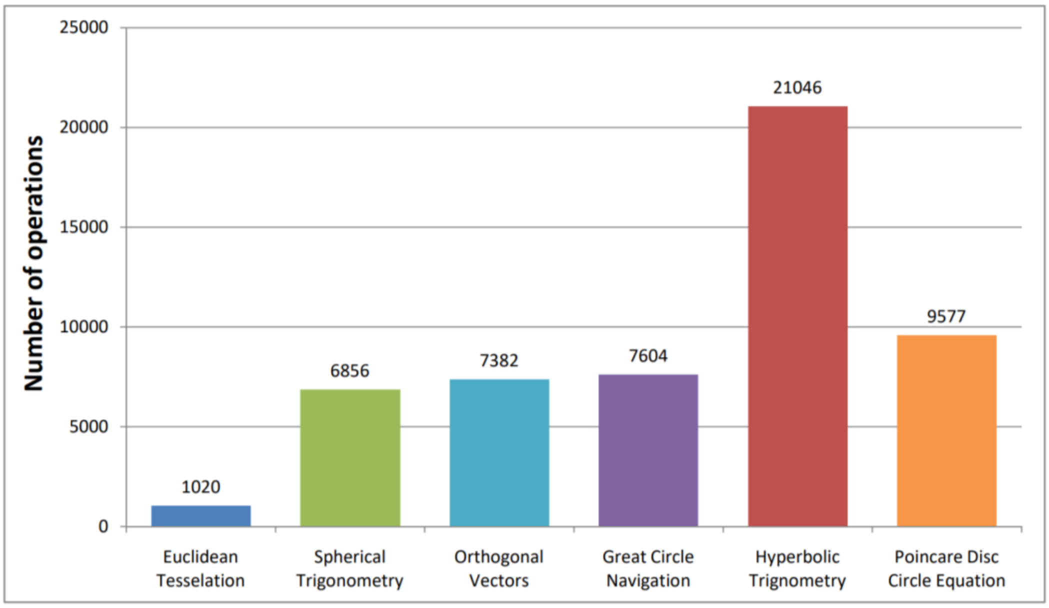


Figure 8.2: Theoretical number of operations for different approaches tessellating 1 shape with 4 vertices and a tessellation parameter of 30, based on CPU benchmarking of mathematical operations

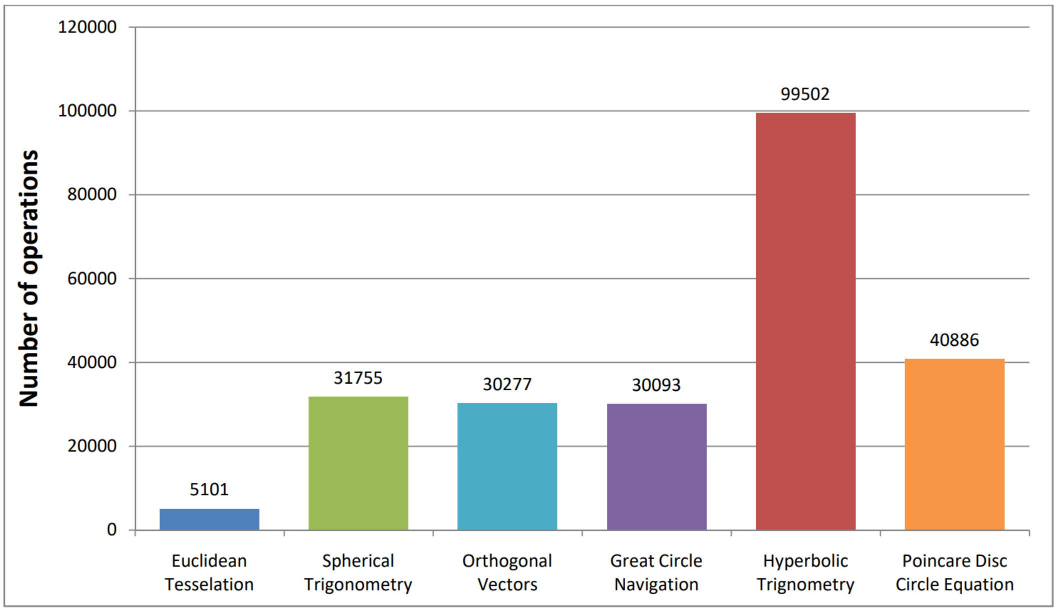


Figure 8.3: Theoretical number of operations for different approaches tessellating 1 shape with 4 vertices and a tessellation parameter of 150, based on CPU benchmarking of mathematical operations

Where C_o is a cost of an operation, n is the number of times this operation is used by the approach and C_t is the total cost of the approach. Both figures calculate the cost for calculating a single shape with 4 vertices, but Figure 8.2 assumes 30 intermediate points along each edge, while Figure 8.3 assumes 150 points.

This comparison illustrates that under low tessellation detail, spherical trigonometry outperforms both of the parametric equation approaches by a small amount, however the opposite is true when the number of intermediate points is high. Hyperbolic trigonometry looks to be disproportionately slow; this is due to hyperbolic arccosine function being more than twice the cost of other trigonometric functions in the results of our benchmark tests. The test has been run multiple times and the result is consistent. GPU implementation of this function could vary from the CPU implementation, which could explain the theoretical prediction differing from data gathered during testing described in the next section.

8.2 Empirical

Testing has been performed in order to collect the data required to analyse each described approach. The test was collecting the frames-per-second (FPS) performance of each approach with an increasing number of shapes. A regular quadrilateral has been used for all shapes and the tessellation parameter was chosen to be 30 for all of the edges of the shapes. These were chosen to be the same for all shapes to have a consistent increase of computing time with each additional shape. Shape size, rotation and position on the screen were pre-determined before the testing was begun and the same values were used for all of the tests. These were chosen to provide a spread of possible values. An unchanging resolution of 1000x1000 pixels has been used for all tests.

The number of shapes has been gradually increased from 1 to 1000 at an increasing rate (i.e. at first the increment in number of shapes is small, but the increment is gradually increased as the number of shapes grows). This was done in order to obtain a higher density of data points when the number of shapes is low, because each additional shape influences the performance a lot more when there are 5 shapes being rendered than when there are 100 shapes being rendered. This also has saved time during testing, because there is no need to have as high a density of the data points when the number of shapes is large, thus higher values for number of shapes could be tested by gradually increasing the increment between successive tests. The engine can operate with any Gaussian curvature ($\boldsymbol{K}$), however the testing has been performed at three different curvature values: $\mathbf{0}$, $\mathbf{1}$ and $\mathbf{-1}$.

The test was conducted on a Personal Computer with the following technical specifications. CPU: Intel(R) Quad-Core(TM) i7-6700K @ 4.00GHz; RAM: 16GB; GPU: NVIDIA GeForce GTX 980; Operating System: 64-bit, Windows 10 Pro.

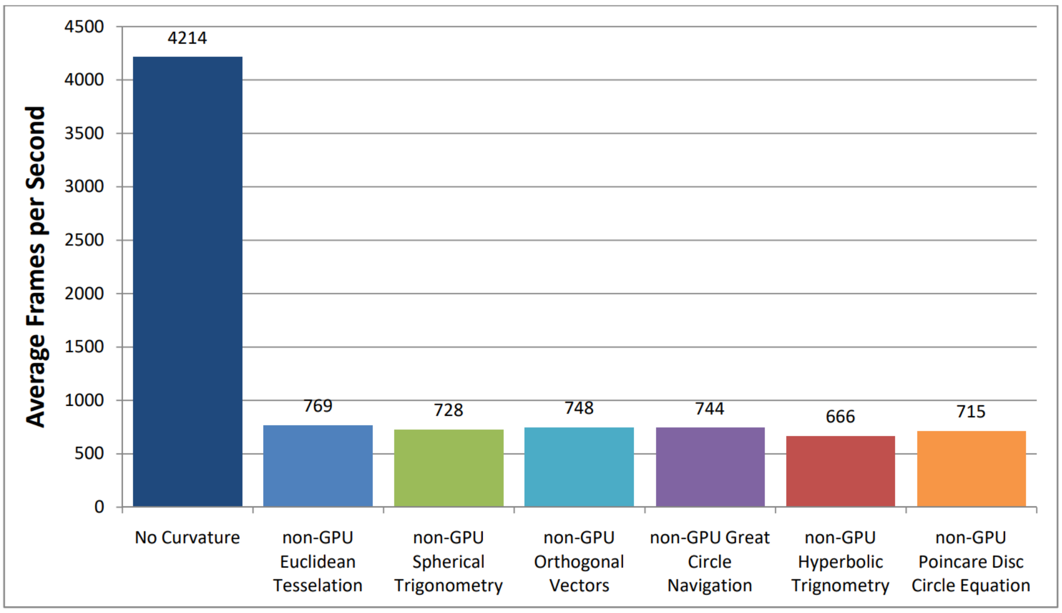


Figure 8.4: Average FPS values of non-GPU approaches (and non-curved baseline) from a test with progressively increasing number of shapes.

In order to eliminate any sudden spikes and drops in FPS, all background processes were disabled. In addition, the test was run multiple times in a loop. Each individual test was conducted for 2 seconds and was progressed sequentially with an increasing number of shapes for each approach before the loop would start again, this would repeat 15 times.

The data collected shows that on average, for non-GPU approaches, the parametric equation approaches performed better by 2-4% (shown in Figure 8.4). This is in line with the prediction made from theoretical analysis of the approaches. The tessellation parameter chosen for the test was 30, which means that step 3 was executed 30 times per edge, so despite the increased number of calculations at step 2, the overall number of expensive computations decreased, thus improving the performance of the program.

The data also shows that spherical trigonometry is more efficient to calculate compared to hyperbolic trigonometry. Two baseline tests were conducted for Euclidean geometry: in the first baseline test, Euclidean No Tessellation, the shapes are not being tessellated, just rendered; in the second baseline test, non-GPU Euclidean Tessellation, the edges of each shape are tessellated into the same number of points as the non-Euclidean tests.

The data from the second baseline test shows that the tessellation with no curvature calculations is more efficient, but the overall differences in performance between non-GPU tessellation tests are small compared to the Euclidean no tessellation rendering.

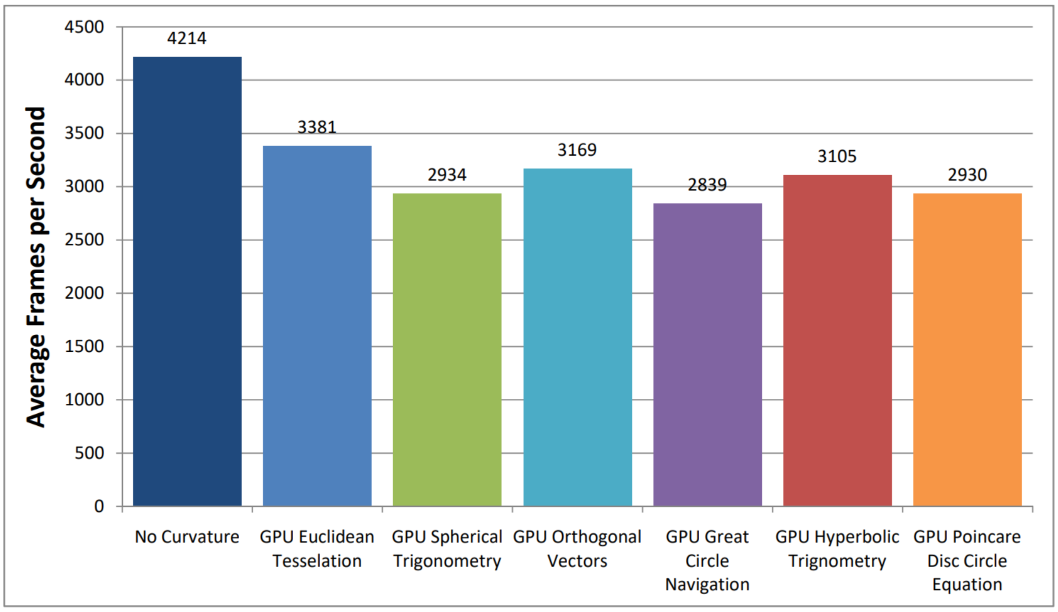


Figure 8.5: Average FPS values of GPU approaches (and non-curved baseline) from a test with progressively increasing number of shapes

Figure 8.5 compares the performance of GPU approaches against two baseline tests: in the first baseline test, Euclidean No Tessellation, the shapes are not being tessellated, just rendered; in the second baseline test, GPU Euclidean Tessellation, the edges of each shape are tessellated into the same number of points as the non-Euclidean tests. The difference in performance between non-GPU and GPU approaches is on average a 5-fold increase. The comparison of GPU approaches reveals that some trends don't change: Euclidean tessellation is still more efficient than non-Euclidean tessellation by a small margin; spherical trigonometry is on average faster to calculate than hyperbolic, however it does depend on the approach used. The difference between approaches is greater than for non-GPU approaches. The surprising find is that while the Orthogonal vectors approach is the most efficient for $\boldsymbol{K = 1}$, the spherical trigonometry approach outperforms Great Circle Navigation; and for $\boldsymbol{K = -1}$, the hyperbolic trigonometry outperforms Poincaré disc approach. Due to the parallelisation of calculations, the small decrease of trigonometric operations performed in Tessellation Evaluation shader does not compensate for the increased cost in the Tessellation Control shader. For example, this could happen when $\boldsymbol{n_i}$ is small, then the tessellation evaluation shader completes the $\boldsymbol{n_s n_v n_i}$ operations and spends the rest of the time waiting for tessellation control shader to complete the $\boldsymbol{n_s n_v}$ operations.

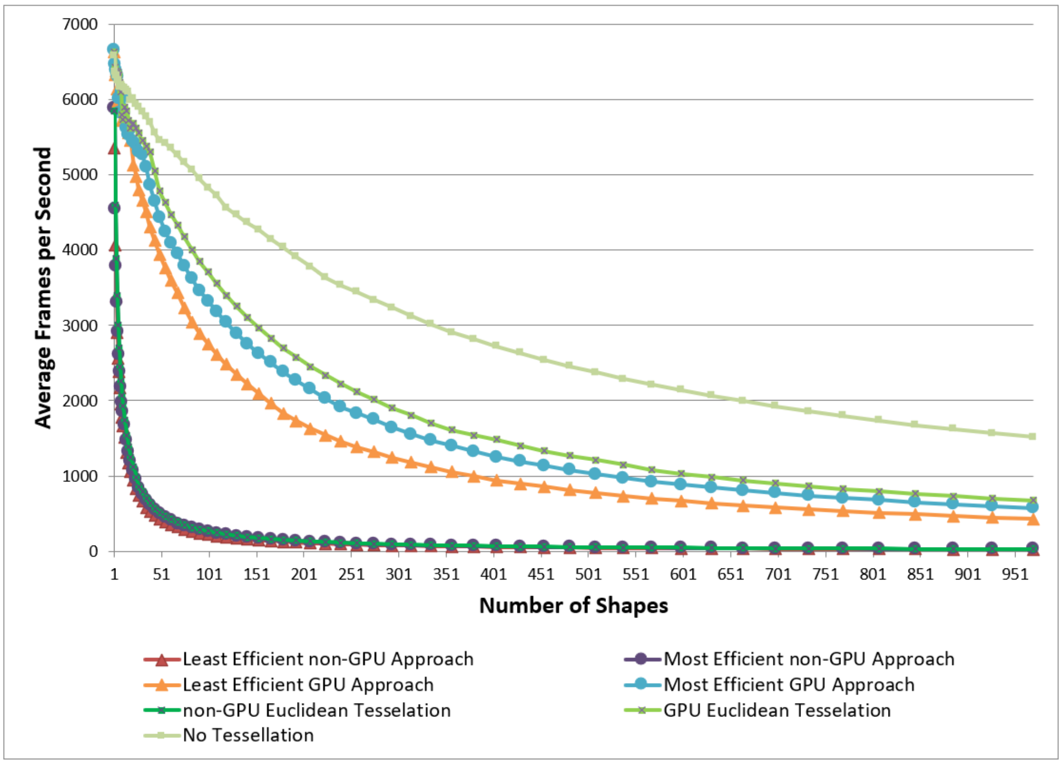


Figure 8.6: Line graph constructed from the data gathered in a test with a progressively increasing number of shapes from 1 to 1000. The graph only shows the best fitting lines for the most and the least efficient approaches for GPU and non-GPU rendering as well as three baseline tests: no tessellation, non-GPU Euclidean Tessellation and GPU Euclidean Tessellation.

Average FPS value is the measure of how many times an approach would render the specified number of shapes in one second, thus the more calculations needed for an approach to render a shape, the lower the FPS value is. This means that it would be expected for values on Figure 8.4 and Figure 8.5 to be the inverse of the values in Figure 8.2 and Figure 8.3, as the testing was performed with a tessellation parameter of 30. Theoretical predictions are roughly confirmed for non-GPU approaches with the difference between spherical tessellation and parametric equation approaches being small; non-GPU hyperbolic trigonometry approach is also slower than the Poincaré disc approach, however the margin is not as large. For GPU approaches, Poincaré disc does not perform as well as hyperbolic trigonometry. This is likely due to parallelisation of the third step of calculations, so the increased number of calculations for the second step is not offset by the reduced number of calculations needed for the third.

Figure 8.6 is a line graph showing the Frames-per-Second for each data point (number of shapes). Each line represents one approach. These lines overlap considerably, but the bottom- and middle-line groups consist of 2 approaches and a baseline test each: the bottom ones are non-GPU approaches and middle ones are GPU approaches. This graph illustrates the performance of each approach with increasing number of shapes being rendered.

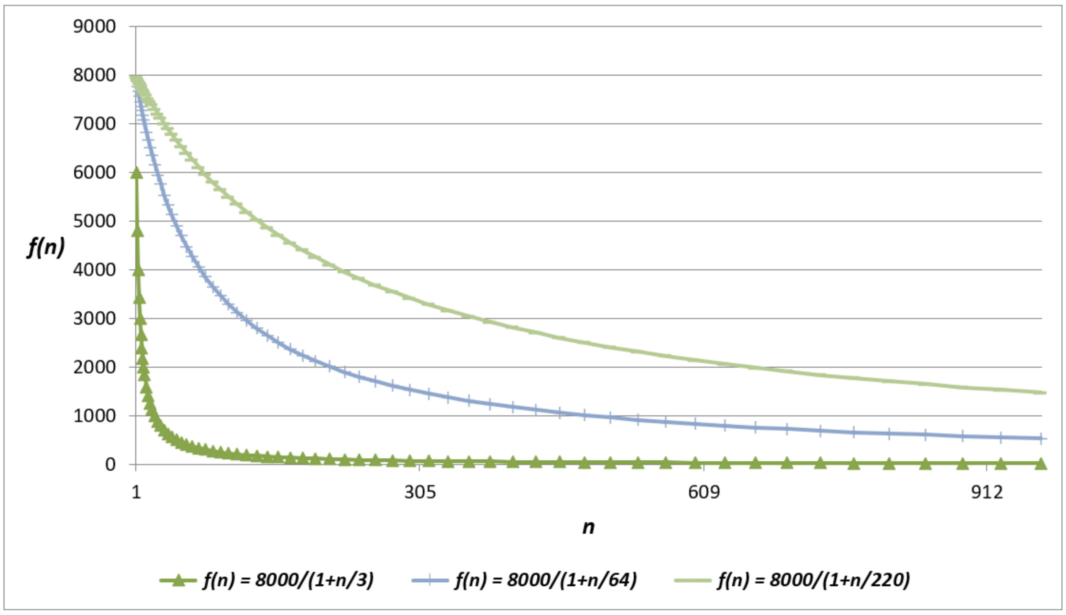


Figure 8.7: Number series of the general form $\boldsymbol{f(n)} = \frac{\mathbf{1}}{\boldsymbol{n}}$ *scaled to approximately fit the shapes of graphs from Figure 8.6.*

Figure 8.7 plots number series of the form $\boldsymbol{f(n)} = \mathbf{8000}/(\mathbf{1} + \frac{\boldsymbol{n}}{\boldsymbol{i}})$, where $\mathbf{8000}$ is chosen to match the baseline of the data collected for the test with $\mathbf{1}$ shape, $\boldsymbol{i}$ is the number of shapes and $\boldsymbol{n}$ is a scaling required to approximate the trends in the collected data (the values of $\boldsymbol{n}$ that fit the scaling are $\mathbf{3}$, $\mathbf{64}$ and $\mathbf{220}$). From this approximation it is clear that firstly, overall performance is inversely proportional to the number of shapes being rendered and the decrease is a scaled $\frac{\mathbf{1}}{\boldsymbol{n}}$ trend; secondly, the GPU approaches are an order of magnitude faster than the non-GPU approaches.

Overall, the implementation of the methods on the GPU has improved performance of the application several fold enabling the engine to render complex simulations with a large number of objects in a non-Euclidean environment in real-time. The improvement of the algorithm efficiency was less impactful however. The 2-4% increase in performance on the CPU is notable but is small in comparison with the increase from parallelisation. Furthermore, only one of the parametric equation approaches performed better than trigonometry approaches on the GPU. On the one hand any increase in application performance is worth pursuing, but keeping trigonometry approach for $\boldsymbol{K} = -\mathbf{1}$, while adopting orthogonal vectors approach for $\boldsymbol{K} = \mathbf{1}$ could be detrimental to further developments, because the approaches differ significantly, while spherical and hyperbolic trigonometry approaches follow the same structure.

The FPS achieved with the GPU based approaches is well suited for real-time game applications, which should run at a stable 60 FPS minimum. In recent years, a lot of games have moved to higher frame rates due to most displays supporting 144Hz format. The GPU approaches exceed both of these limits even when a large number of objects are rendered on the screen. The improved performance of the application has created an opportunity for further developments. Specifically, this has made texturing of shapes in a non-Euclidean environment possible in real-time. Keeping the spherical and hyperbolic trigonometry approaches has been beneficial to implementing texturing, as it allowed for a more precise determination of each of the intermediate points along the edge, which has been useful for computing texture coordinates in non-Euclidean environment. This is further detailed in chapter 9.

8.3 Summary

This chapter has described the comparison between different approaches described in chapters 4, 6 and 7. The approaches have been compared theoretically by looking at the number of calculations required to render a single shape at different values of the tessellation parameter. Subsequently the results of empirical testing have been detailed and analysed. The parallelisation of the approaches has proven to improve the performance several fold. On the other hand, line-equation based approaches have been shown to be comparable in performance to spherical and hyperbolic trigonometry approaches. Only one approach has yielded a marginal improvement over the trigonometry approaches, but in comparison with parallelisation it was negligible. Furthermore, keeping the spherical and hyperbolic trigonometry approaches has been preferable for implementing texturing, as it allowed for a more precise determination of each of the intermediate points and their texture coordinates in non-Euclidean environment, further described in chapter 9.

9. Method for Texturing 2D Shapes using Spherical and Hyperbolic Trigonometry

This chapter is original research, which introduces a method for finding intermediate points throughout a shape in order to render textured shapes; it also explains the modifications made in the shader pipeline to find texture coordinates for the curved shape.

Note: angle magnitudes in each diagram and equation within this section are normalised to be in the range $[\mathbf{0}, \mathbf{2\pi}]$.

9.1 Splitting Object into Patches

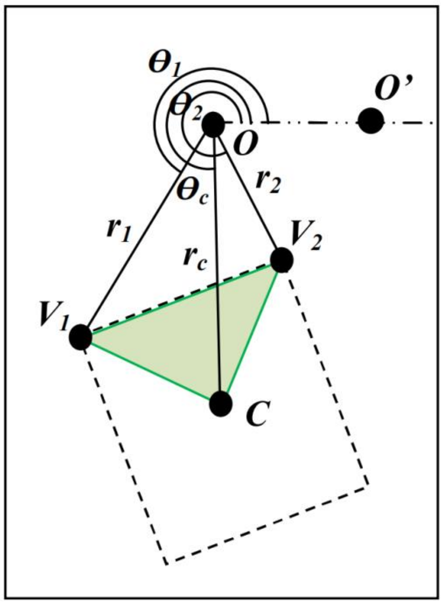


Figure 9.1: Texturing a shape by subdividing it into patches, one patch per edge. Patch $\boldsymbol{V_1V_2C}$ *is highlighted.* $\boldsymbol{O(0,0)}$*, reference point of a polar coordinate system;* $\boldsymbol{V_1(r_1,\theta_1)}$*,* $\boldsymbol{V_2(r_2,\theta_2)}$*, vertices of the object;* $\boldsymbol{C(r_c,\theta_c)}$*, object's position;* △*, area of the shape rendered by the patch* $\boldsymbol{V_1V_2C}$*;* $\boldsymbol{\overline{OV_1} = r_1}$*;* $\boldsymbol{\overline{OV_2} = r_2}$*;* $\boldsymbol{\overline{OC} = r_c}$*;* $\boldsymbol{\angle V_1OO' = \theta_1}$*;* $\boldsymbol{\angle V_2OO' = \theta_2}$*;* $\boldsymbol{\angle COO' = \theta_c}$

Previously for the method described in chapter 4, `GL_LINES` rendering mode has been used to render shapes, it draws lines between the pairs of vertices. This has been changed to use `GL_PATCHES` instead to implement the GPU tessellation described in chapter 7. Pairs of vertices were passed in as a patch and intermediate points along the edge were found within the shader. However, to render textures in curved space, intermediate points would also need to be found throughout the surface of the shape. Thus, it was chosen to use the shape's position, $\mathbf{C(r_c, \theta_c)}$, as a third point for the patch, so a patch for the edge $\overline{\mathbf{V_1V_2}}$ would be $\mathbf{V_1V_2C}$

(illustrated on Figure 9.1). Using such subdivision, the entire area of the shape would be tessellated.

9.2 Intermediate Vertex Coordinates

Once the patches have been set up, a method for finding intermediate points' texture coordinates has been developed. It is split into two steps: finding intermediate vertices and finding intermediate points using these vertices.

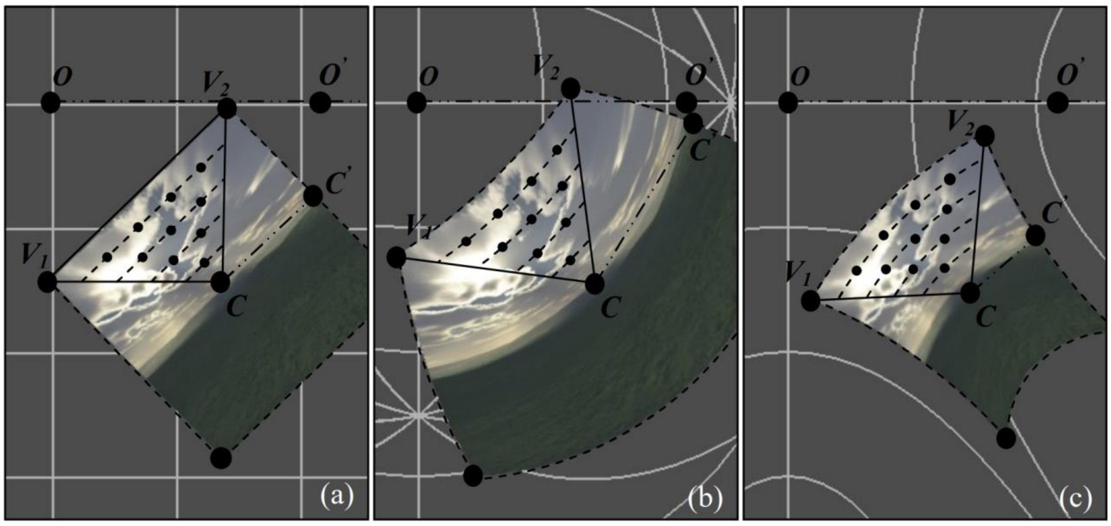


Figure 9.2: Position of the texture coordinates within a patch $\boldsymbol{V_1 V_2 C}$ *displayed in (a) spherical, (b) planar and (c) hyperbolic space.* $\boldsymbol{O}$*, reference point of the global coordinate system;* $\boldsymbol{OO'}$*, reference direction of the global coordinate system;* $\boldsymbol{C}$*, reference point of the local coordinate system;* $\boldsymbol{CC'}$*, reference direction of local coordinate system;* $\boldsymbol{V_1}$*,* $\boldsymbol{V_1}$*, object's vertices; -•- , object's texture coordinates*

To structure the calculation required to find intermediate points it was decided to use a number of smaller similar triangles sharing the vertex $\boldsymbol{C}$ for each patch, dependant on the outer tessellation level. Figure 9.2 shows an overview of how this method works to render a shape in different curvatures. The diagram is overlaid onto the rendering produced by the developed software.

Once the coordinates of each vertex within the edge are found using the method described in Chapter 0, instead of tessellating the edges as described in Chapters 4.3 and 0, intermediate vertices are found.

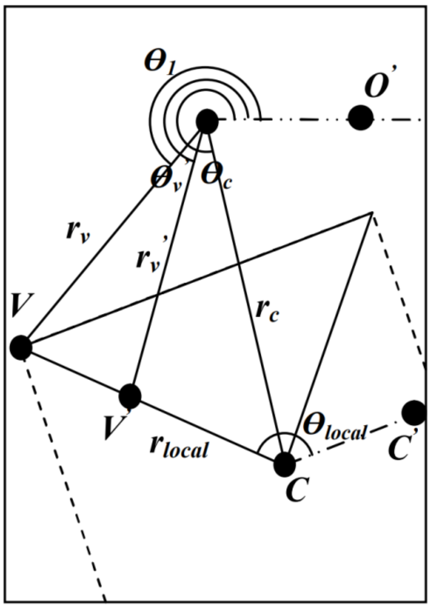


Figure 9.3: Finding an intermediate vertex $\boldsymbol{V'}\,(\boldsymbol{r'_v}, \boldsymbol{\theta'_v})$ along the line $\overline{\boldsymbol{VC}}$. $\boldsymbol{O}(\mathbf{0}, \mathbf{0})$, reference point of the global coordinate system; $\boldsymbol{OO'}$, reference direction of the global coordinate system; $\boldsymbol{C}(\boldsymbol{r_c}, \boldsymbol{\theta_c})$, object's position and reference point of local coordinate system; $\boldsymbol{CC'}$, reference direction of local coordinate system; $\boldsymbol{V}(\boldsymbol{r_v}, \boldsymbol{\theta_v})$, object's vertex with local coordinates $\boldsymbol{V}(\boldsymbol{r_{local}}, \boldsymbol{\theta_{local}})$; $\overline{\boldsymbol{OV}} = \boldsymbol{r_v}$; $\overline{\boldsymbol{OV}}' = \boldsymbol{r'_v}$; $\overline{\boldsymbol{CV}} = \boldsymbol{r_{local}}$; $\overline{\boldsymbol{OC}} = \boldsymbol{r_c}$; $\angle \boldsymbol{VOO'} = \boldsymbol{\theta_v}$; $\angle \boldsymbol{V'OO'} = \boldsymbol{\theta'_v}$; $\angle \boldsymbol{COO'} = \boldsymbol{\theta_c}$; $\angle \boldsymbol{VCC'} = \boldsymbol{\theta_{local}}$

First the intermediate vertex $\mathbf{V'}$ is found for each vertex in the patch (illustrated on Figure 9.3). Vertex $\mathbf{V'}$ is located along the line $\overline{\mathbf{VC}}$ and it can be found using the similar method as described in Chapter 0. Intermediate vertex $\mathbf{V'}$ shares the local polar coordinates with vertex $\mathbf{V}$, except the $\mathbf{r_{local}}$ coordinate is scaled by the internal tessellation coordinate of the current point to find $\mathbf{r'_{local}}$. Then using $\boldsymbol{\beta}$ found as shown in Chapter 0 for vertex $\mathbf{V}$, $\mathbf{r'_v}$ and $\boldsymbol{\theta'_v}$ coordinates are found via the following equations. These are different to equations :

If $\mathbf{K} > 0$, then:

$$\boldsymbol{r'_v} = \mathbf{cos^{-1}}(\mathbf{cos}\, \boldsymbol{r_c} \,\mathbf{cos}\, \boldsymbol{r'_{local}} + \mathbf{sin}\, \boldsymbol{r_c} \,\mathbf{sin}\, \boldsymbol{r'_{local}} \,\mathbf{cos}\, \boldsymbol{\beta}) \tag{9.1}$$

$$\boldsymbol{\Delta\theta'_v} = \mathbf{cos^{-1}}\left(\frac{\mathbf{cos}\, \boldsymbol{r'_{local}} - \mathbf{cos}\, \boldsymbol{r_c} \,\mathbf{cos}\, \boldsymbol{r'_v}}{\mathbf{sin}\, \boldsymbol{r_c} \,\mathbf{sin}\, \boldsymbol{r'_v}}\right) \tag{9.2}$$

If $\mathbf{K} > 0$, then:

$$\boldsymbol{r'_v} = \mathbf{cosh^{-1}}(\mathbf{cosh}\, \boldsymbol{r_c}\, \mathbf{cosh}\, \boldsymbol{r'_{local}} - \mathbf{sinh}\, \boldsymbol{r_c}\, \mathbf{sinh}\, \boldsymbol{r'_{local}}\, \mathbf{cos}\, \boldsymbol{\beta}) \qquad (9.3)$$

$$\boldsymbol{\Delta\theta'_v} = \mathbf{cos^{-1}}\left(\frac{\mathbf{cosh}\, \boldsymbol{r'_v}\, \mathbf{cosh}\, \boldsymbol{r_c} - \mathbf{cosh}\, \boldsymbol{r'_{local}}}{\mathbf{sinh}\, \boldsymbol{r_c}\, \mathbf{sinh}\, \boldsymbol{r'_v}}\right) \qquad (9.4)$$

The $\boldsymbol{\Delta\theta'}$ is then added to or subtracted from $\boldsymbol{\theta_c}$ to find $\boldsymbol{\theta'_v}$ depending on how the $\boldsymbol{\theta_v}$ was found as in Chapter 0.

Note: Equations *(9.1)* and *(9.2)* could become unstable when the angles are small. To make calculations more precise the haversine formula (Korn & Korn, 2000) could be used instead for spherical space. For small angles instead of equations *(9.1)* and *(9.2)*, use the following two equations respectively:

$$\boldsymbol{r'_v} = \mathbf{hav^{-1}}(\mathbf{hav}(\boldsymbol{r_c} - \boldsymbol{r'_{local}}) + \mathbf{sin}\, \boldsymbol{r_c}\, \mathbf{sin}\, \boldsymbol{r'_{local}}\, \mathbf{hav}\, \boldsymbol{\beta}) \qquad (9.5)$$

$$\boldsymbol{\Delta\theta'_v} = \mathbf{hav^{-1}}\left(\frac{\mathbf{hav}\, \boldsymbol{r'_{local}} - \mathbf{hav}(\boldsymbol{r_c} - \boldsymbol{r'_v})}{\mathbf{sin}\, \boldsymbol{r_c}\, \mathbf{sin}\, \boldsymbol{r'_v}}\right) \qquad (9.6)$$

In the engine, these two formulae have been used within the vertex shader, described in section 9.4.1, due to the calculations becoming less stable on the GPU than on the CPU when $\boldsymbol{\beta} < \frac{\boldsymbol{\pi}}{\mathbf{60}}$.

9.3 Intermediate Point Coordinates

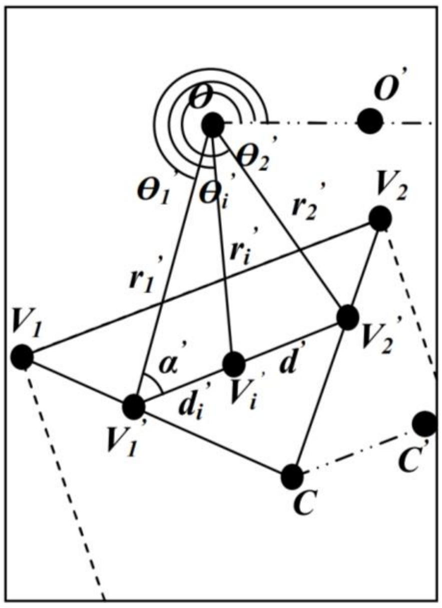


Figure 9.4: Finding a series of intermediate points $\boldsymbol{V'_i(r'_i, \theta'_i)}$ *along the intermediate edge* $\overline{\boldsymbol{V'_1V'_2}}$. $\boldsymbol{O(0,0)}$, *reference point of the global coordinate system;* $\boldsymbol{OO'}$, *reference direction of the global coordinate system;* $\boldsymbol{C}$, *object's position;* $\boldsymbol{V_1}$, $\boldsymbol{V_2}$, *object's vertices;* $\boldsymbol{V'_1(r'_1, \theta'_1)}$, $\boldsymbol{V'_2(r'_2, \theta'_2)}$, *object's intermediate vertices;* $\overline{\boldsymbol{OV'_1}} = \boldsymbol{r'_1}$; $\overline{\boldsymbol{OV'_2}} = \boldsymbol{r'_2}$; $\overline{\boldsymbol{OV'_i}} = \boldsymbol{r'_i}$; $\overline{\boldsymbol{V'_1V'_2}} = \boldsymbol{d'}$; $\overline{\boldsymbol{V'_1V'_i}} = \boldsymbol{d'_i}$; $\angle\boldsymbol{V'_1OO'} = \boldsymbol{\theta'_1}$; $\angle\boldsymbol{V'_2OO'} = \boldsymbol{\theta'_2}$; $\angle\boldsymbol{V'_iOO'} = \boldsymbol{\theta'_i}$; $\angle\boldsymbol{OV'_1V'_2} = \boldsymbol{\alpha'}$

After finding the coordinates of both intermediate vertices $\boldsymbol{V'_1}$ and $\boldsymbol{V'_2}$, the preliminaries for finding intermediate points need to be found. Firstly, $\boldsymbol{d'}$, a geodesic distance between $\boldsymbol{V'_1V'_2}$ is found using the cosine law. In order to do that, the angle $\angle\boldsymbol{V'_1OV'_2}$ ($\boldsymbol{\Delta\theta'}$) must be determined. Like in Chapter 4.3 there are 2 cases depending on whether the angles converge or diverge.

If the angles diverge, $\boldsymbol{\Delta\theta'} = \boldsymbol{\theta'_1} + \boldsymbol{\theta'_2}$; if the angles converge, $\boldsymbol{\Delta\theta'} = |\boldsymbol{\theta'_1} - \boldsymbol{\theta'_2}|$. Once $\boldsymbol{\Delta\theta'}$ is found, $\boldsymbol{d'}$ can be found using the following formulas (illustrated on Figure 9.4):

If $\mathbf{K} > 0$, then:

$$\boldsymbol{d'} = \cos^{-1}(\cos r'_1 \cos r'_2 + \sin r'_1 \sin r'_2 \cos \Delta\theta') \qquad (9.7)$$

If $\mathbf{K} < 0$, then:

$$\boldsymbol{d'} = \cosh^{-1}(\cosh r'_1 \cosh r'_2 - \sinh r'_1 \sinh r'_2 \cos \Delta\theta') \qquad (9.8)$$

Knowing the lengths $\boldsymbol{d'}$, $\boldsymbol{r'_1}$ and $\boldsymbol{r'_2}$, the angle $\boldsymbol{\alpha'}$ can be found, which will be used to find a series of intermediate points, $\boldsymbol{V'_i}$, along the geodesic $\overline{\boldsymbol{V'_1 V'_2}}$. This is illustrated on Figure 9.4.

If $\mathbf{K} > 0$, then:

$$\boldsymbol{\alpha'} = \mathbf{cos}^{-1}\left(\frac{\cos \boldsymbol{r'_2} - \cos \boldsymbol{r'_1} \cos \boldsymbol{d'}}{\sin \boldsymbol{r'_1} \sin \boldsymbol{d'}}\right) \tag{9.9}$$

If $\mathbf{K} < 0$, then:

$$\boldsymbol{\alpha'} = \mathbf{cos}^{-1}\left(\frac{\cosh \boldsymbol{r'_1} \cosh \boldsymbol{d'} - \cosh \boldsymbol{r'_2}}{\sinh \boldsymbol{r'_1} \sinh \boldsymbol{d'}}\right) \tag{9.10}$$

Note: Equations *(9.7)* and *(9.9)* could become unstable when the angles are small. To make calculations more precise the haversine formula (Korn & Korn, 2000) could be used instead for spherical space. For small angles instead of equations *(9.7)* and *(9.9)*, use the following two equations respectively:

$$\boldsymbol{d'} = \mathbf{hav}^{-1}(\mathbf{hav}(\boldsymbol{r'_1} - \boldsymbol{r'_2}) + \sin \boldsymbol{r'_1} \sin \boldsymbol{r'_2}\, \mathbf{hav}\, \boldsymbol{\Delta\theta'}) \tag{9.11}$$

$$\boldsymbol{\alpha'} = \mathbf{hav}^{-1}\left(\frac{\mathbf{hav}\, \boldsymbol{r'_2} - \mathbf{hav}(\boldsymbol{r'_1} - \boldsymbol{d'})}{\sin \boldsymbol{r'_1} \sin \boldsymbol{d'}}\right) \tag{9.12}$$

In the engine, these two formulae have been used within the tessellation control shader, described in section 9.4.2, due to the calculations becoming less stable on the GPU than on the CPU when $\boldsymbol{\Delta\theta'} < \frac{\boldsymbol{\pi}}{\mathbf{10}}$.

Like in section 2.1.3 distance $\boldsymbol{d'}$ is divided into a number of equal parts, depending on the inner tessellation value, such that:

$$\boldsymbol{d'_i} = \frac{\boldsymbol{i\, d'}}{\boldsymbol{tess\_param}} \tag{9.13}$$

Note: $\boldsymbol{tess\_param}$ varies depending on the distance to $\boldsymbol{C}$. The farther the intermediate coordinate is from $\boldsymbol{C}$, the higher the $\boldsymbol{tess\_param}$.

This gives the distance $\boldsymbol{V'_1 V'_i}$ ($\boldsymbol{d'_i}$) for each of the intermediate points along the geodesic $\boldsymbol{V'_1 V'_2}$. Then $\boldsymbol{r'_i}$ and subsequently $\boldsymbol{\Delta\theta'_i}$ is found using the following formulas:

If $\mathbf{K} > 0$, then:

$$\boldsymbol{r'_i} = \mathbf{cos^{-1}}(\mathbf{cos}\,\boldsymbol{r'_1}\,\mathbf{cos}\,\boldsymbol{d'_i} + \mathbf{sin}\,\boldsymbol{r'_1}\,\mathbf{sin}\,\boldsymbol{d'_i}\,\mathbf{cos}\,\boldsymbol{\alpha'}) \tag{9.14}$$

$$\boldsymbol{\Delta\theta'_i} = \mathbf{cos^{-1}}\left(\frac{\mathbf{cos}\,\boldsymbol{d'_i} - \mathbf{cos}\,\boldsymbol{r'_1}\,\mathbf{cos}\,\boldsymbol{r'_i}}{\mathbf{sin}\,\boldsymbol{r'_1}\,\mathbf{sin}\,\boldsymbol{r'_i}}\right) \tag{9.15}$$

If $\mathbf{K} < 0$, then:

$$\boldsymbol{r'_i} = \mathbf{cosh^{-1}}(\mathbf{cosh}\,\boldsymbol{r'_1}\,\mathbf{cosh}\,\boldsymbol{d'_i} - \mathbf{sinh}\,\boldsymbol{r'_1}\,\mathbf{sinh}\,\boldsymbol{d'_i}\,\mathbf{cos}\,\boldsymbol{\alpha'}) \tag{9.16}$$

$$\boldsymbol{\Delta\theta'_i} = \mathbf{cos^{-1}}\left(\frac{\mathbf{cosh}\,\boldsymbol{r'_1}\,\mathbf{cosh}\,\boldsymbol{r'_i} - \mathbf{cosh}\,\boldsymbol{d'_i}}{\mathbf{sinh}\,\boldsymbol{r'_1}\,\mathbf{sinh}\,\boldsymbol{r'_i}}\right) \tag{9.17}$$

Note: Equations *(9.14)* and *(9.15)* could become unstable when the angles are small. To make calculations more precise the haversine formula (Korn & Korn, 2000) could be used instead for spherical space. For small angles instead of equations *(9.14)* and *(9.15)*, use the following two equations respectively:

$$\boldsymbol{r'_i} = \mathbf{hav^{-1}}(\mathbf{hav}(\boldsymbol{r'_1} - \boldsymbol{d'_i}) + \mathbf{sin}\,\boldsymbol{r'_1}\,\mathbf{sin}\,\boldsymbol{d'_i}\,\mathbf{hav}\,\boldsymbol{\alpha'}) \tag{9.18}$$

$$\boldsymbol{\Delta\theta'_i} = \mathbf{hav^{-1}}\left(\frac{\mathbf{hav}\,\boldsymbol{d'_i} - \mathbf{hav}(\boldsymbol{r'_1} - \boldsymbol{r'_i})}{\mathbf{sin}\,\boldsymbol{r'_1}\,\mathbf{sin}\,\boldsymbol{r'_i}}\right) \tag{9.19}$$

In the engine, these two formulae have been used within the tessellation control shader, described in section 9.4.3, due to the calculations becoming less stable on the GPU than on the CPU when $\boldsymbol{\alpha} < \frac{\boldsymbol{\pi}}{\mathbf{10}}$.

Finally, to find the coordinates of the point $\boldsymbol{V'_i}$, $\boldsymbol{r'_i}$ should be multiplied by $\boldsymbol{r}$ or $\boldsymbol{k}$ depending on the value of $\boldsymbol{K}$; $\boldsymbol{\Delta\theta'_i}$ should be added to or subtracted from angle $\boldsymbol{\theta'_1}$, depending on the direction of the edge $\overline{\boldsymbol{V'_1V'_2}}$, determined previously.

9.4 Texture Shader Pseudocode

Below, the texture shaders are summarised using pseudocode. The vertex shader has only one difference compared to section 7.1.1. In addition to $\boldsymbol{k}$, $\boldsymbol{r_c}$, $\boldsymbol{\theta_c}$, $\boldsymbol{\alpha}$, the tessellation amount and vertex array of $\boldsymbol{r_{local}}$ and $\boldsymbol{\theta_{local}}$ coordinates, a vertex array of texture coordinates, `tex_coord` is passed in as an input.

In GLSL texture coordinates are treated as one of the attributes of a vertex, similarly to normals. Thus, they are added to the vertex array object when a shape is set up and are subsequently passed to the shader when that model is being rendered. The code below shows the layout of a vertex array object for the meshes within the engine:

```
      // sending vertex data to gpu
      ref<vertex_buffer> vb(vertex_buffer::create((float*) vertices.data(),
(uint32_t)vertices.size() * sizeof(vertex)));

      // sending index data to gpu
      ref<index_buffer> ib(index_buffer::create((uint32_t*)indices.data(),
(uint32_t)indices.size()));

      const buffer_layout layout
      {
            {e_shader_data_type::float3, "a_position"},
            {e_shader_data_type::float3, "a_normal"},
            {e_shader_data_type::float2, "a_tex_coord"},
      };
      vb->layout(layout);

      m_va = engine::vertex_array::create();
      m_va->add_buffer(vb);
      m_va->add_buffer(ib);
```

9.4.1 Vertex Shader

Like in section 7.1.1, spherical or hyperbolic trigonometry is used to find global positions of each of the shape's vertices. In addition, the `tex_coord` is not being modified and is just passed through to the tessellation shaders.

In the vertex shader, the global polar coordinates for every vertex are found in parallel. This is possible, because each vertex only depends on: the global coordinates of the centre point of the shape, rotation angle of the shape and local polar coordinates of that vertex. Thus the vertex shader follows the same algorithm as described in section 7.1.1, but is executed on the GPU and subsequently passes the global vertex coordinates to the next shader in the rendering pipeline (tessellation control shader).

```
inputs: k, r_c, theta_c, alpha, vertex_array[r_local, theta_local],
     vertex_array[tex_coord_x, tex_coord_y]


initialise r_v, theta_v, delta_theta, beta
initialise direction_v = 1
r_c = r_c / k
r_local = r_local / k
beta = r_local + alpha
if beta > pi
     direction_v = -1
     beta = 2 * pi - beta
if K > 0
     if beta > pi / 60
          use equation (4.3) to find r_v
          use equation (4.4) to find delta_theta
     else
          use equation (4.7) to find r_v
          use equation (4.8) to find delta_theta
else if K < 0
     use equation (4.5) to find r_v
     use equation (4.6) to find delta_theta
delta_theta = delta_theta * direction_v
theta_v = theta_c + delta_theta


output: cos_r_v, sin_r_v, cosh_r_v, sinh_r_v, r_v, theta_v,
vertex_array[tex_coord_x, tex_coord_y], vertex_array[r_local,
theta_local]
```

9.4.2 Tesselation Control Shader

The shape is separated into patches of 2 vertices each, but to render textures, a third point is used: object position, with coordinates $\boldsymbol{r_c}$, $\boldsymbol{\theta_c}$. For each edge, the tessellation shaders will produce a specified number of intermediate points which lie on a series of geodesics lying within a sector of the shape, as described in section 9.1.

Preliminaries for the calculation of intermediate points are found and tessellation parameters are set in the tessellation control shader. These are subsequently passed on to the tessellation evaluation shader. Using the trigonometry approach, the preliminaries (i.e., the length of the edge, $\boldsymbol{d}$, and the angle between the edge and vertex 1, $\boldsymbol{\alpha}$) are found following the method described in section 0.

Additionally, inner and outer tessellation levels are set up in the tessellation control shader. These affect how many intermediate points are generated by the Primitive Generator step of the pipeline. Three values of the outer tessellation level control the amount of subdivision of each of the edge of the patch triangle. The inner tessellation level controls the number of subdivisions inside this triangle. Figure 9.5 shows three different examples of different tessellation values. Figure 9.5 (a) and (b) illustrate the tessellation when all of the tessellation level values are the same, while (c) shows an example when all of the tessellation levels are set to different values.

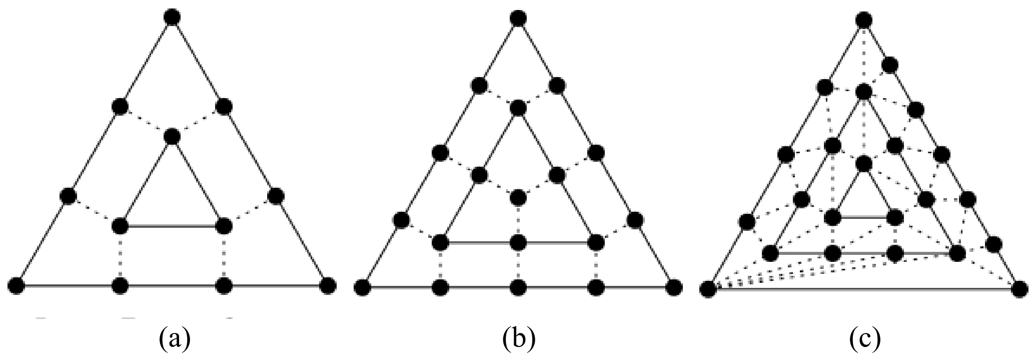


Figure 9.5: Tessellation of a triangle patch with different parameters of the inner and outer tessellation levels. (a) Inner tessellation levels 3, 3, 3 Outer tessellation level 3; (b) Inner tessellation levels 4, 4, 4 Outer tessellation level 4; (c) Inner tessellation levels 4, 1, 6 Outer tessellation level 5 (The Khronos Group, 2020).

In the tessellation evaluation shader each tessellated point is given a tessellation coordinate. The tessellation coordinate contains three values and is defined by barycentric coordinates of a triangle. Each of the three vertices of the patch triangle are associated with one of the three values of the tessellation coordinate and represents the proximity of the tessellated point to that vertex of the triangle (illustrated in Figure 9.6).

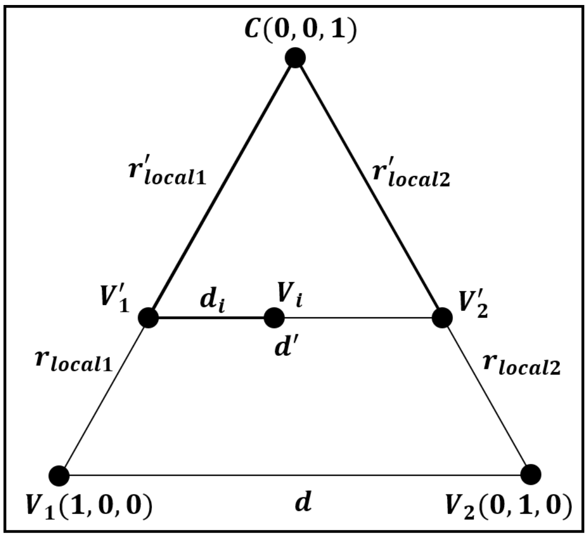


Figure 9.6: Tessellation of a triangle patch and barycentric coordinates of a triangle. Intermediate point, $\boldsymbol{V_i}$, position. $\boldsymbol{C}$, position of the shape and reference point of local coordinate system, third vertex of this triangle patch; $\boldsymbol{V_1}$, $\boldsymbol{V_2}$ vertices of the triangle patch; $\boldsymbol{V'_1}$, $\boldsymbol{V'_2}$, intermediate vertices found during tessellation; $\overline{\boldsymbol{V_1V_2}} = \boldsymbol{d}$, $\overline{\boldsymbol{CV_1}} = \boldsymbol{r_{local1}}$, $\overline{\boldsymbol{CV_2}} = \boldsymbol{r_{local2}}$, $\overline{\boldsymbol{CV'_1}} = \boldsymbol{r'_{local1}}$, $\overline{\boldsymbol{CV'_2}} = \boldsymbol{r'_{local2}}$, $\overline{\boldsymbol{V'_1V'_2}} = \boldsymbol{d'}$, $\overline{\boldsymbol{V'_1V_i}} = \boldsymbol{d_i}$.

Using the tessellation coordinate, $\boldsymbol{tess\_coord}(\boldsymbol{x_{tess\_coord}}, \boldsymbol{y_{tess\_coord}}, \boldsymbol{z_{tess\_coord}})$, of a point, the distances $\mathbf{r'_{local1}}$ and $\mathbf{r'_{local2}}$ can be found as proportions of $\boldsymbol{r_{local1}}$ and $\boldsymbol{r_{local2}}$ respectively:

$$\mathbf{r'_{local1}} = \boldsymbol{r_{local1}}(\mathbf{1} - \boldsymbol{z_{tess\_coord}}) \quad (9.20)$$

$$\mathbf{r'_{local2}} = \boldsymbol{r_{local2}}(\mathbf{1} - \boldsymbol{z_{tess\_coord}}) \quad (9.21)$$

Subsequently, when the length of the intermediate edge $\overline{\mathbf{V'_1}\boldsymbol{V'_2}}$ is found ($\mathbf{d'}$), $\boldsymbol{tess\_coord}$ is used to calculate the distance $\mathbf{d_i}$. It is a proportion of the $\boldsymbol{y_{tess\_coord}}$ in the sum of $\boldsymbol{x_{tess\_coord}}$ and $\boldsymbol{y_{tess\_coord}}$ in the tessellation coordinate of point $\boldsymbol{V_i}$:

$$\mathbf{d_i} = \mathbf{d'}\frac{\boldsymbol{y_{tess\_coord}}}{\boldsymbol{x_{tess\_coord}} + \boldsymbol{y_{tess\_coord}}} \quad (9.22)$$

Note: these equations *(4.20)*, *(4.21)* and *(4.22)* are the same for spherical and hyperbolic space.

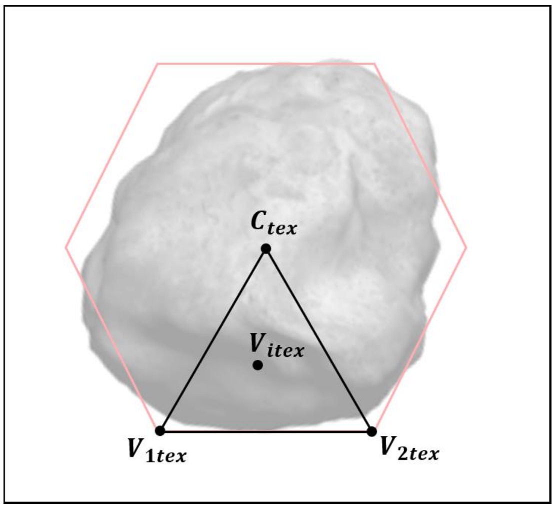


Figure 9.7: Triangle patch overlayed onto the texture file. $\boldsymbol{V_{itex}}$, $\boldsymbol{V_{1tex}}$, $\boldsymbol{V_{2tex}}$, $\boldsymbol{C_{tex}}$ *texture coordinates of points* $\boldsymbol{V_i}$, $\boldsymbol{V_1}$, $\boldsymbol{V_2}$, $\boldsymbol{C}$ *respectively.*

Finally, the tessellation coordinate of point $\boldsymbol{V_i}$ is also used to calculate the corresponding texture coordinates for this point:

$$\boldsymbol{V}_{\mathbf{itex}} = \boldsymbol{V}_{\mathbf{1tex}}\boldsymbol{x}_{\boldsymbol{tess\_coord}} + \boldsymbol{V}_{\mathbf{2tex}}\boldsymbol{y}_{\boldsymbol{tess\_coord}} + \mathbf{C}_{\mathbf{tex}}\boldsymbol{z}_{\boldsymbol{tess\_coord}} \qquad (9.23)$$

Note: $\boldsymbol{V}_{\mathbf{itex}}$, $\boldsymbol{V}_{\mathbf{1tex}}$, $\boldsymbol{V}_{\mathbf{2tex}}$, $\mathbf{C}_{\mathbf{tex}}$ are 2-dimensional vectors representing the cartesian coordinates of a point on a texture file.

```
inputs: r_c, theta_c, array[cos_r_v], array[sin_r_v],
array[cosh_r_v], array[sinh_r_v], vertex_array[r_v, theta_v],
vertex_array[tex_coord_x, tex_coord_y], vertex_array[r_local,
theta_local], vertex_array[r_local, theta_local],
tessellation_parameter_inner, tessellation_parameter_outer


initialise alpha, d
initialise direction = 1
initialise delta_theta = |theta_1 - theta_2|
if K > 0
     if delta_theta > pi / 10
          use equation (4.13) to find d
          use equation (4.15) to find alpha
     else
          use equation (4.17) to find d
          use equation (4.18) to find alpha
else if K < 0
     use equation (4.14) to find d
     use equation (4.16) to find alpha
if the triangles are converging
     direction = -1
initialise tess_level_outer = tessellation_parameter_inner
initialise tess_level_inner = tessellation_parameter_outer


outputs: d, cos_alpha, delta_theta, direction, array[sin_r],
array[cos_r], array[sinh_r], array[cosh_r],
vertex_array[r_v, theta_v], vertex_array[tex_coord_x, tex_coord_y],
vertex_array[r_local, theta_local]
```

9.4.3 Tessellation Evaluation Shader

A number of intermediate vertices is generated depending on the values of the inner and outer tessellation levels, `tess_level_outer` and `tess_level_inner`. Then a series of point coordinates, joining each set of 2 intermediate vertices is generated. Outer tessellation level determines the number of intermediate points.

```
inputs: k, r_c, theta_c, d, cos_alpha, delta_theta, direction,
array[sin_r], array[cos_r], array[sinh_r], array[cosh_r],
vertex_array[r_v, theta_v], vertex_array[tex_coord_x, tex_coord_y],
vertex_array[r_local, theta_local], array[beta]


initialise r_1_local', r_2_local', r_1', r_2', theta_1', theta_2',
delta_theta_1', delta_theta_2', r_i, theta_i, delta_theta_i, d',
alpha'
r_1_local' = (1 – tess_coord.z) * r_1_local
r_2_local' = (1 – tess_coord.z) * r_2_local
if K > 0
    if beta > pi / 60
        use equation (9.1) to find r_1' and r_2'
        use equation (9.2) to find delta_theta_1' and delta_theta_2'
    else
        use equation (9.5) to find r_1' and r_2'
        use equation (9.6) to find delta_theta_1' and delta_theta_2'
else if K < 0
    use equation (9.3) to find r_1' and r_2'
    use equation (9.4) to find delta_theta_1' and delta_theta_2'
delta_theta_1' = delta_theta_1' * direction_v1
delta_theta_2' = delta_theta_2' * direction_v2
theta_1' = theta_c + delta_theta_1'
theta_2' = theta_c + delta_theta_2'
```

```
if K > 0
    if delta_theta' > pi / 10
        use equation (9.7) to find d'
        use equation (9.9) to find alpha'
    else
        use equation (9.11) to find d'
        use equation (9.12) to find alpha'
else if K < 0
    use equation (9.8) to find d'
    use equation (9.10) to find alpha'

initialise d_i = d' * tess_coord.y / (tess_coord.x + tess_coord.y)
if K > 0
    if alpha' > pi / 10
        use equation (9.14) to find r_i
        use equation (9.15) to find delta_theta_i
    else
        use equation (9.18) to find r_i
        use equation (9.19) to find delta_theta_i
else if K < 0
    use equation (9.16) to find r_i
    use equation (9.17) to find delta_theta_i
delta_theta_i = delta_theta_i * direction
theta_i = theta_1' + delta_theta_i
r_i = r_i * k
x_i = r_i * cos(theta_i)
y_i = r_i * sin(theta_i)
```

```
initialise tex_coord_c, tex_coord_i

tex_coord_c = [0.5, 0.5]

tex_coord_i = tex_coord_r1 * tess_coord.x + tex_coord_r2 *
     tess_coord.y + tex_coord_c * tess_coord.z

outputs: x_i, y_i, tex_coord_i
```

9.5 Texture Rendering Results

The method described in this chapter has been implemented in the non-Euclidean engine and an example of non-Euclidean rendering can be seen in Figure 9.2.

The Asteroids game, implementation of which is explained in section 11.1, has been adapted to use texture-based rendering. Figure 9.8, Figure 9.9 and Figure 9.10 show a playthrough of the Asteroids game in Euclidean, spherical and hyperbolic space respectively.

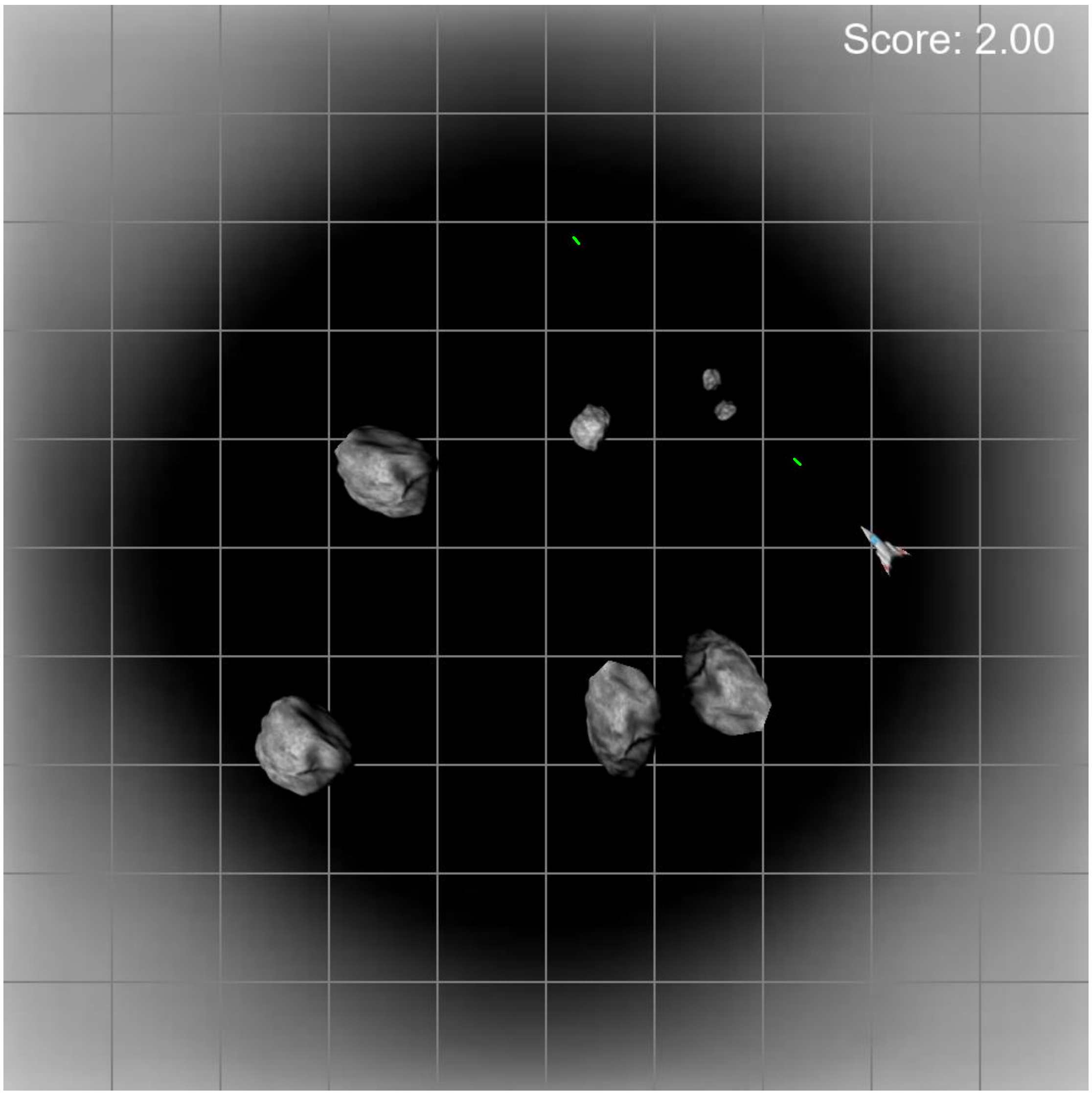


Figure 9.8: A screenshot of Asteroids game playthrough in planar space. One of the large asteroids has been hit and split into two shards. One of the shards has subsequently been hit and split. ($\boldsymbol{K = 0}$)

The game world includes 5 asteroid shapes and a player's spaceship, which has an option to shoot lasers, which are rendered as short green dashes using the line-rendering described in chapter 7.

These playthroughs have the player shooting the laser multiple times, one of the asteroids is hit and splits into two shards. Subsequently one of these shards is also hit and split into two smaller shards. So, the screenshots taken show the player's spaceship, laser particles, four original asteroids, one medium asteroid shard and one small asteroid shard.

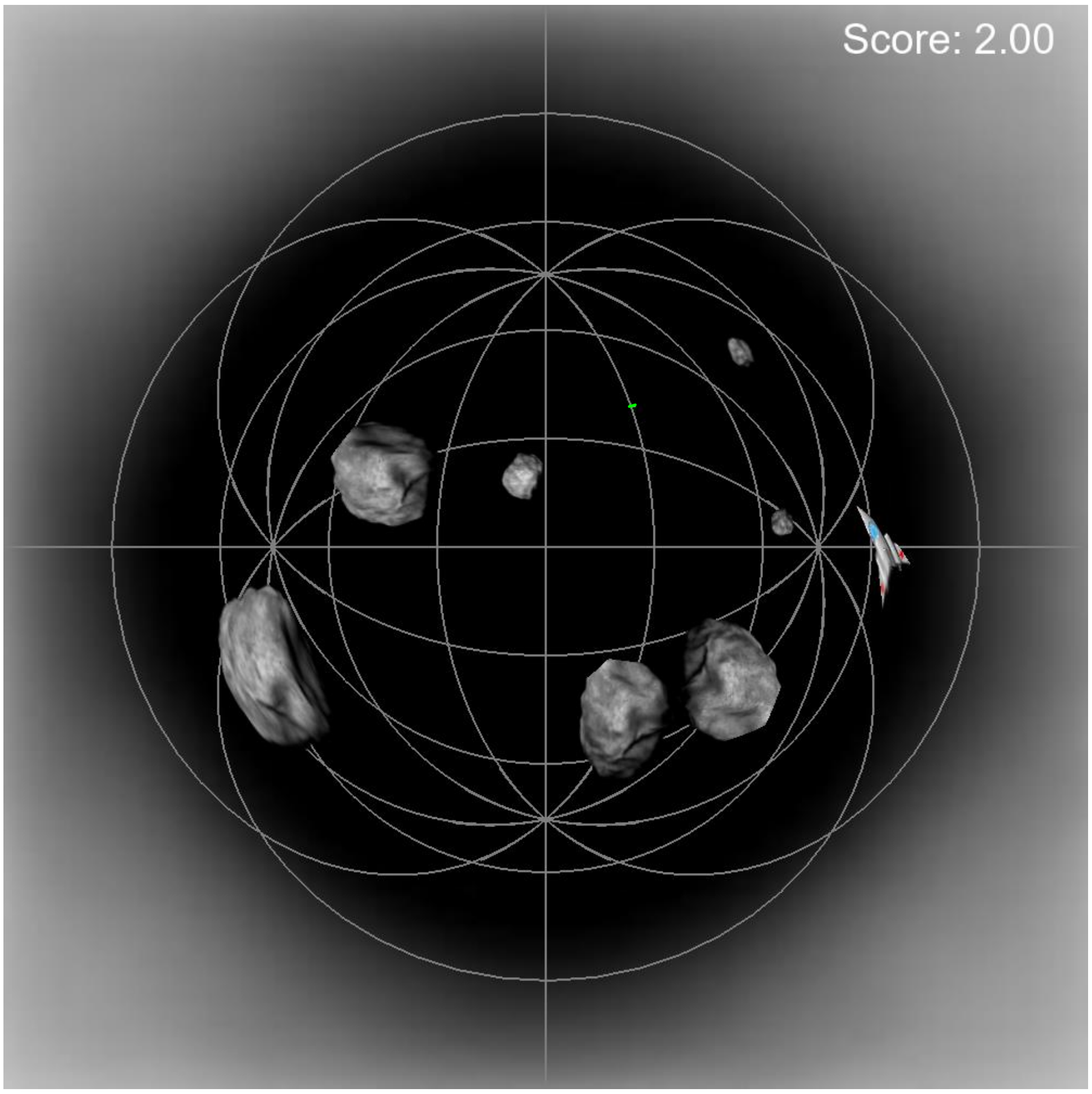


Figure 9.9: A screenshot of Asteroids game playthrough in spherical space. One of the large asteroids has been hit and split into two shards. One of the shards has subsequently been hit and split. $(\boldsymbol{K} = \mathbf{1})$

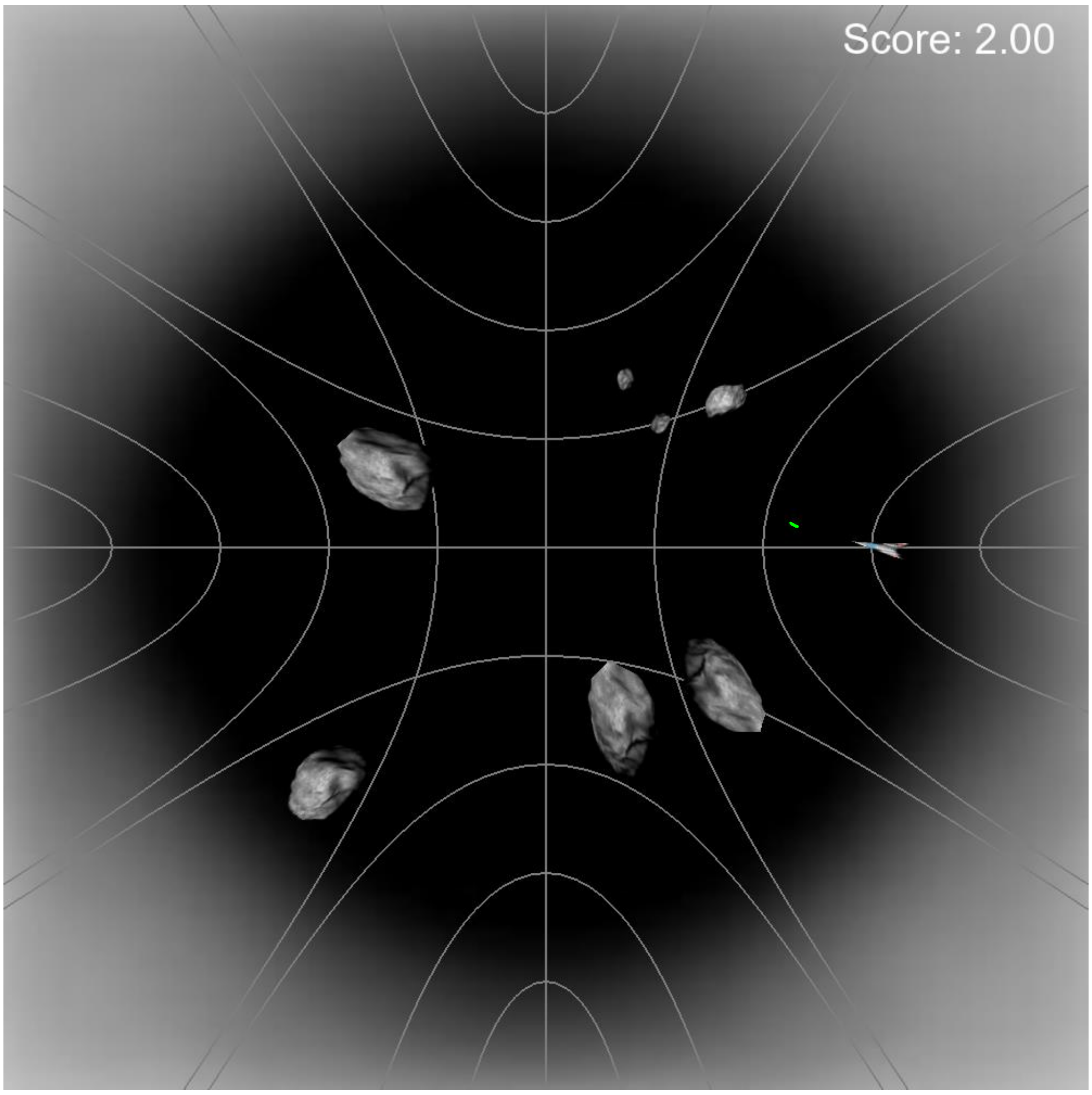


Figure 9.10: A screenshot of Asteroids game playthrough in hyperbolic space. One of the large asteroids has been hit and split into two shards. One of the shards has subsequently been hit and split. $(\boldsymbol{K = -1})$

The spaceship shape is a concave quadrilateral with the following local vertex coordinates:

$$\left[(\mathbf{30\ px}, \mathbf{0\ rad}), \left(\mathbf{30\ px}, \frac{\mathbf{5}\boldsymbol{\pi}}{\mathbf{6}}\ \mathbf{rad}\right), (\mathbf{15\ px}, \boldsymbol{\pi}\ \mathbf{rad}), \left(\mathbf{30\ px}, \frac{\mathbf{7}\boldsymbol{\pi}}{\mathbf{6}}\ \mathbf{rad}\right)\right]$$

A “simple spaceship” (Xevin, 2014) texture has been applied to the spaceship object. The textures with shape outlines are shown in Figure 9.11.

Asteroids are created as regular hexagons with the following local vertex coordinates:

$$\left[\left(\mathbf{60}, \frac{\boldsymbol{\pi}}{\mathbf{6}}\right), \left(\mathbf{60}, \frac{\boldsymbol{\pi}}{\mathbf{2}}\right), \left(\mathbf{60}, \frac{\mathbf{5}\boldsymbol{\pi}}{\mathbf{6}}\right), \left(\mathbf{60}, \frac{\mathbf{7}\boldsymbol{\pi}}{\mathbf{6}}\right), \left(\mathbf{60}, \frac{\mathbf{3}\boldsymbol{\pi}}{\mathbf{2}}\right), \left(\mathbf{60}, \frac{\mathbf{11}\boldsymbol{\pi}}{\mathbf{6}}\right)\right]$$

Each asteroid uses one of the randomly determined grey asteroids textures from the “asteroids” textures pack (Phaelax, 2014). Figure 11.1, Figure 11.2 and Figure 11.3 show the non-textured versions of asteroid and spaceship shapes in Euclidean, spherical and hyperbolic space respectively.

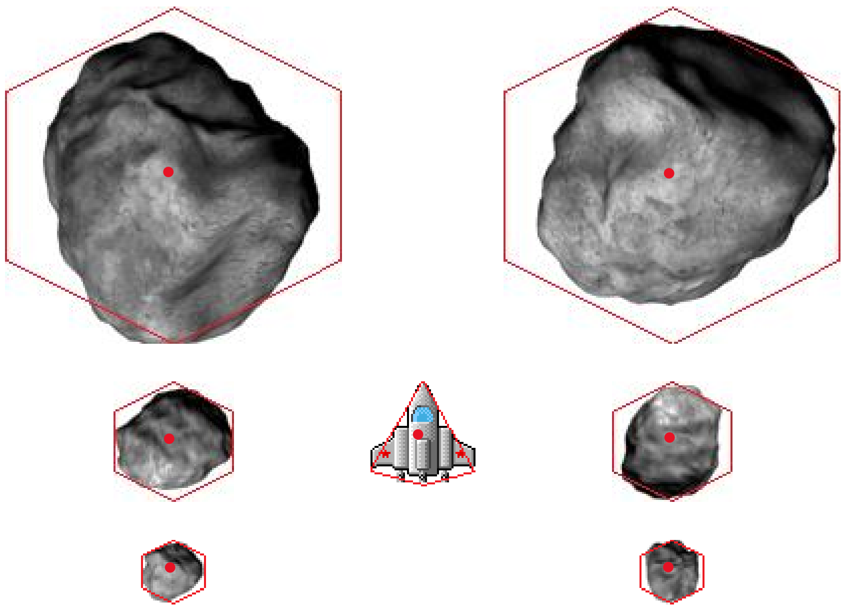


Figure 9.11: A spaceship as well as large, medium and small asteroids textures with the shape outlines set by the texture coordinates. Centre of each shape is marked with a red dot.

Some of the textures are cut off as shown on Figure 9.11. For the spaceship this is done to keep the shape similar to the spaceship from the original Asteroids game, which is an arrowlike concave quadrilateral (for example, the concave spaceship shape can be seen in the Figure Figure 5.11). The coordinates set on the texture, shown in Figure 9.11, are that of a convex quadrilateral to include the engines of the spaceship. The texture is compressed down the middle of the spaceship, but stretched on the sides to maintain the original Asteroids spaceship shape.

Some of the textures of the asteroids are cut off as well, which can be seen on the large asteroids in Figure 9.11 and within the game in Figure 11.1, where the leading corner of the asteroid closest to the origin has the texture cut off and an angle of the hexagon can be seen.

The asteroid textures are not perfectly centred and have a random orientation within the texture pack, as they are used to create an in-game sprite animation of the asteroid spinning. This meant that if all orientations were accounted for when setting texture coordinates, there was too much empty space recorded around the asteroid within its texture, which would mean the collision detection would be less precise. For this reason it was decided to have a trade off between the textures being slightly cut off and the collision detection being more accurate.**Note:** the position of the centre of the shape is important because the shape in hyperbolic and spherical space changes depending on the position of the centre. The bigger the shape, the more pronounced this change is. This happens because vertices of the shape are just a representation of an object in polar coordinates and the flat texture file is then stretched accordingly. If a curvature changes during the execution of the game, the internal distance between any two points, which do not lie on the same line passing through the origin, gets stretched or contracted.

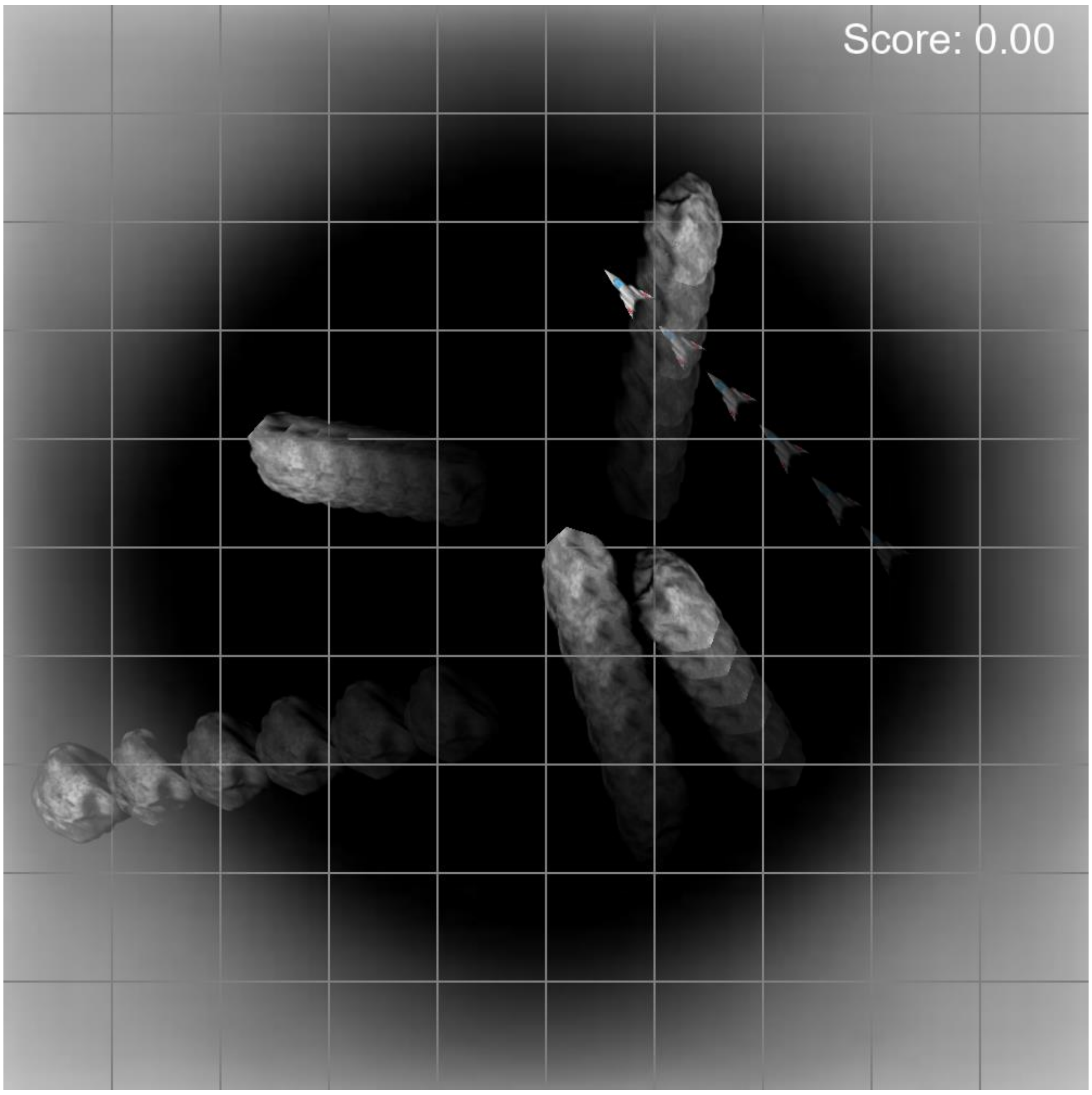


Figure 9.12: Time-lapse of multiple textured objects (5 asteroids and a spaceship) moving through planar space within the Asteroids game simulation ($\boldsymbol{K} = \mathbf{0}$). This simulation is not the same playthrough as the one in Figure 9.8.

To illustrate how textured objects move within the simulation dynamically time-lapses have been created. Figure 9.12, Figure 9.13 and Figure 9.14 show the time-lapses in Euclidean, spherical and hyperbolic space respectively. The same initial parameters have been set for the time-lapses as for the playthroughs shown in Figure 9.8, Figure 9.9 and Figure 9.10.

Collision detection between the spaceship and asteroids has been switched off in order to have a continuous timelapse.

The time-lapses consist of 6 separate screenshots of the game simulation. Each screenshot is taken 1 second after the previous one.

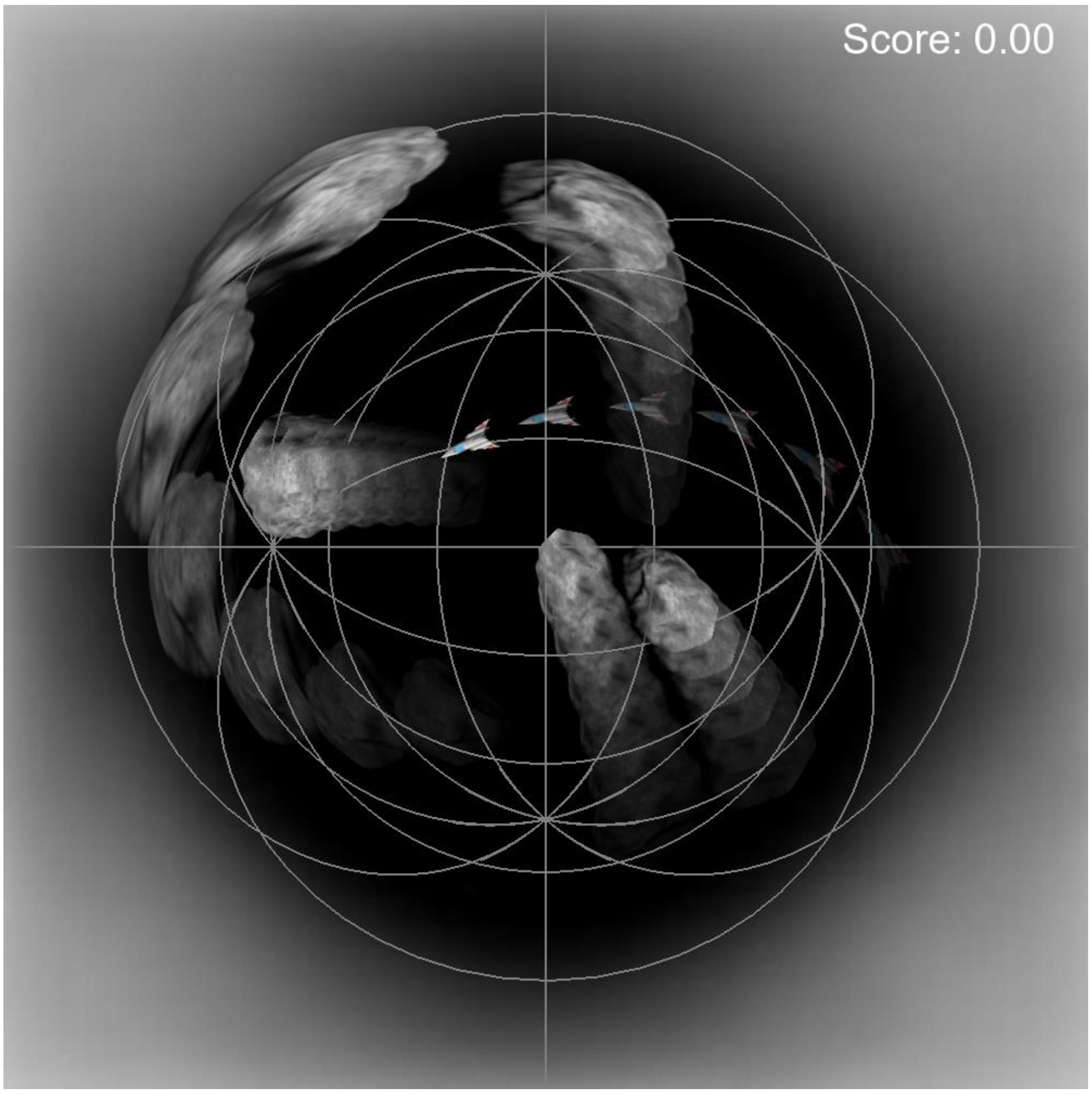


Figure 9.13: Time-lapse of multiple textured objects' (5 asteroids and a spaceship) movement through spherical space within the Asteroids game simulation ($\boldsymbol{K} = \mathbf{1}$) This simulation is not the same playthrough as the one in Figure 9.9.

The initial parameters (i.e., position and velocity) for the objects are identical irrespective of curvature. However, the trajectories the objects follow in the time-lapse are different. Because the object's parameters are not modified during the execution of the simulation, the geodesic they follow is uniquely determined by the initial parameters and the projection used (i.e., Azimuthal Equidistant projection).

In Figure 9.13 the objects follow the great circle paths through spherical space, but in hyperbolic space (shown in Figure 9.14) they follow the hyperbolic lines. The gridlines show some of the geodesics, which helps the player visualise the object's trajectories through the curved space.

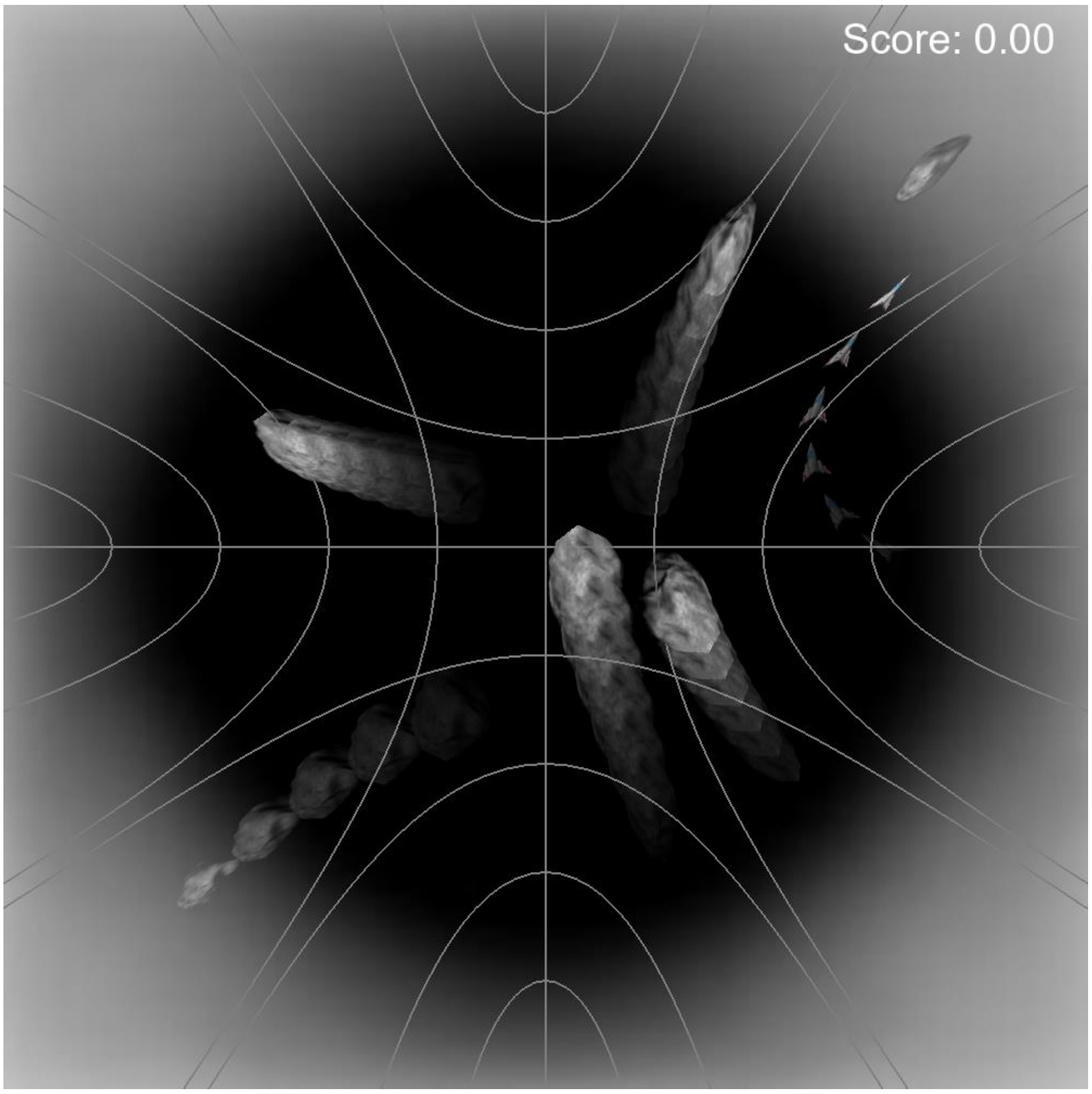


Figure 9.14: Time-lapse of multiple textured objects' (5 asteroids and a spaceship) movement through hyperbolic space within the Asteroids game simulation ($\boldsymbol{K = -1}$). This simulation is not the same playthrough as the one in Figure 9.10.

Some of the objects in Euclidean and hyperbolic space would move off screen, so they are wrapped around to the antipodal point of the circular projection defined as dark circle within the figures.

It is possible to see how the textures get stretched laterally the further they are from the centre of projection in spherical space (shown in Figure 9.9 and Figure 9.13), while in hyperbolic space they get compressed instead (shown in Figure 9.10 and Figure 9.14).

9.6 Summary

This chapter has described a method for texturing shapes in non-Euclidean geometry by finding the correct texture coordinates throughout the shape. This is done by first subdividing the shape into segments for each of the edges. Segment are subsequently subdivided into multiple smaller similar triangles sharing the vertex C to find the intermediate texture coordinates. Segments are processed by the shader pipeline as patches. First, the vertex shader finds global vertex coordinates for each of the vertices within the patch using the method described in section 0. Subsequently tessellation shaders find intermediate vertices along the inner edges of the segment. These are then used to find texture coordinates along each of the intermediate edges throughout the segment. This method has been implemented within the engine and an Asteroids game has been adapted to use textured shapes. Time-lapses and playthrough screenshots of this game have been presented in this chapter.

10. Analysis II: Texture Rendering Performance

This chapter is original research, which describes the empirical testing method used to compare the performance of line rendering and texture rendering. The analysis of the data gathered from this testing is then presented. Testing has been performed in order to collect the data required to compare texture rendering approaches and line rendering approaches.

The test was conducted on a Personal Computer with the following technical specifications. CPU: Intel(R) Quad-Core(TM) i7-6700K Quad Core @ 4.00GHz; RAM: 16GB; GPU: NVIDIA GeForce GTX 980; Operating System: 64-bit, Windows 10 Pro. An unchanging resolution of 1000x1000 pixels has been used for all tests. In order to eliminate any sudden spikes and drops in FPS, all background processes were disabled. The engine can operate with any Gaussian curvature ($\boldsymbol{K}$), however the testing has been performed at three different curvature values: $\mathbf{0}$, $\mathbf{1}$ and $\mathbf{-1}$.

The test was performed multiple times for a different number of objects rendered: 100, 200, 400 and 800 objects. These values were chosen such that the number of objects is doubled for each subsequent test. This would help determine how the number objects affects the frame rate at different tessellation parameter values. A regular quadrilateral has been used for all shapes similarly to the testing performed in chapter 8. Size, rotation and position of all shapes on the screen were pre-determined before the testing started and the same values were used for all of the tests. These were chosen to provide a spread of possible values. All 800 objects are illustrated in Figure 10.1 rendered in hyperbolic (left), Euclidean (centre) and spherical (right) space. Note that the background image and the gridlines have not been rendered during the testing process.

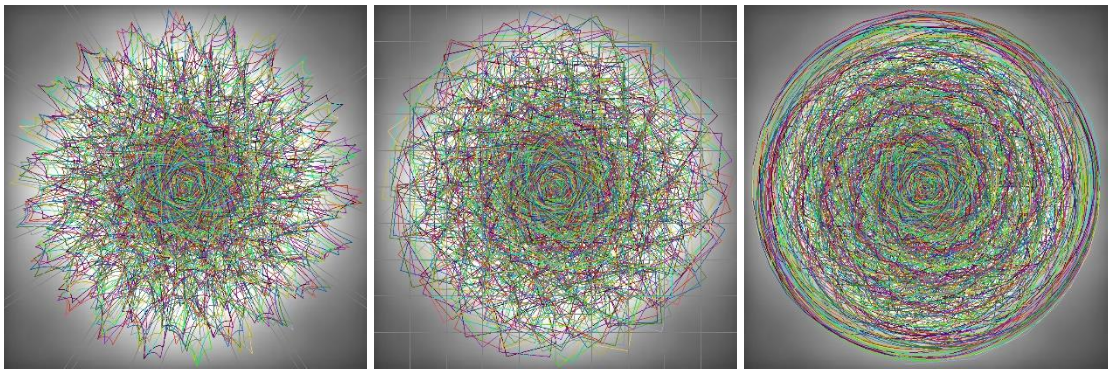


Figure 10.1: 800 regular quadrilateral objects used for testing rendered in hyperbolic (left), Euclidean (centre) and spherical (right) space. They are positioned and rotated randomly in the bounded area of the projection.

The test has been run multiple times in a loop. The test was first run for the line rendering approach with a tessellation parameter of 25. Each individual test was conducted for 5 seconds before the tessellation parameter was increased from 25 to 90 in increments of 5, then it progressed through the different numbers of objects (i.e., 100, 200, 400 and 800), again starting with tessellation parameter of 25. Once the test for the line rendering approach had completed, the test for texture rendering approach was run with the similar structure. After one such cycle was complete, the loop started again and was repeated 15 times. A constant increment of the tessellation parameter was chosen to gauge the effect it has on the frame rate.

When all 15 cycles were completed, the averages were taken for each unique combination of approach, number of objects and tessellation parameter from across the 15 cycles to reduce the variance. Two baseline tests have been performed for No Tessellation and Euclidean texturing.

Figure 10.2, Figure 10.3, Figure 10.4 and Figure 10.5 show the results of the test for 100, 200, 400 and 800 shapes respectively. The graphs show the average FPS for increasing tessellation parameter values in 4 different approaches and two baselines.

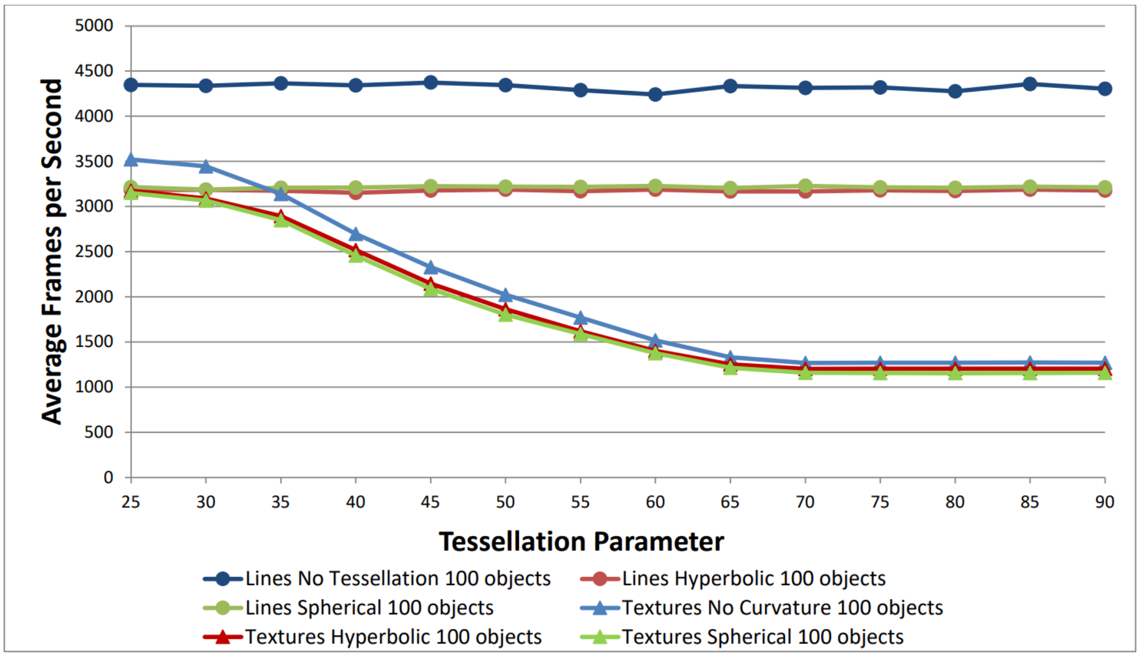


Figure 10.2: Performance test with 100 quadrilateral objects rendered with increasing tessellation parameter.

Figure 10.2 shows the data from a test with 100 objects. The engine was rendering objects without tessellation at approximately 4,400 FPS. Hyperbolic and spherical line tessellation approaches were rendered at approximately 3,100 FPS regardless of the tessellation parameter value. The texture rendering approaches are performing at approximately 3,100 FPS when tessellation parameter is 25, but the performance drops gradually as the tessellation parameter increases to a minimum value of approximately 1,200 FPS.

Regardless of the number of shapes, the graphs show a similar pattern, non-Euclidean line tessellation and texture tessellation have roughly equal average FPS values when the tessellation parameter is low. As it increases, performance for the texture tessellation gradually drops until a tessellation parameter value of $\boldsymbol{n} \approx \mathbf{70}$. At that value, the performance plateaus and does not decrease with higher values of $\boldsymbol{n}$. This is because a hardware dependent maximum tessellation level value (`GL_MAX_TESS_GEN_LEVEL`) is reached. For the personal computer used for this test, the value is between $\boldsymbol{n} = \mathbf{66}$ and $\boldsymbol{n} = \mathbf{70}$, which would create between 3,367 and 3,781 intermediate points per patch respectively (calculated using equation *(10.1)*).

This is where the maximum number of output components per patch is reached; this value is hardware dependent.

Figure 10.3 shows the data from a test with 200 objects. No tessellation baseline was rendered at approximately 3,400 FPS. Curved space line tessellation approaches were performing at approximately 2,100 FPS at all tessellation parameter values. The textured shapes were rendered at approximately 2,000 FPS for tessellation parameter of 25, but it decreased as the tessellation parameter increased to approximately a minimum value of 600 FPS.

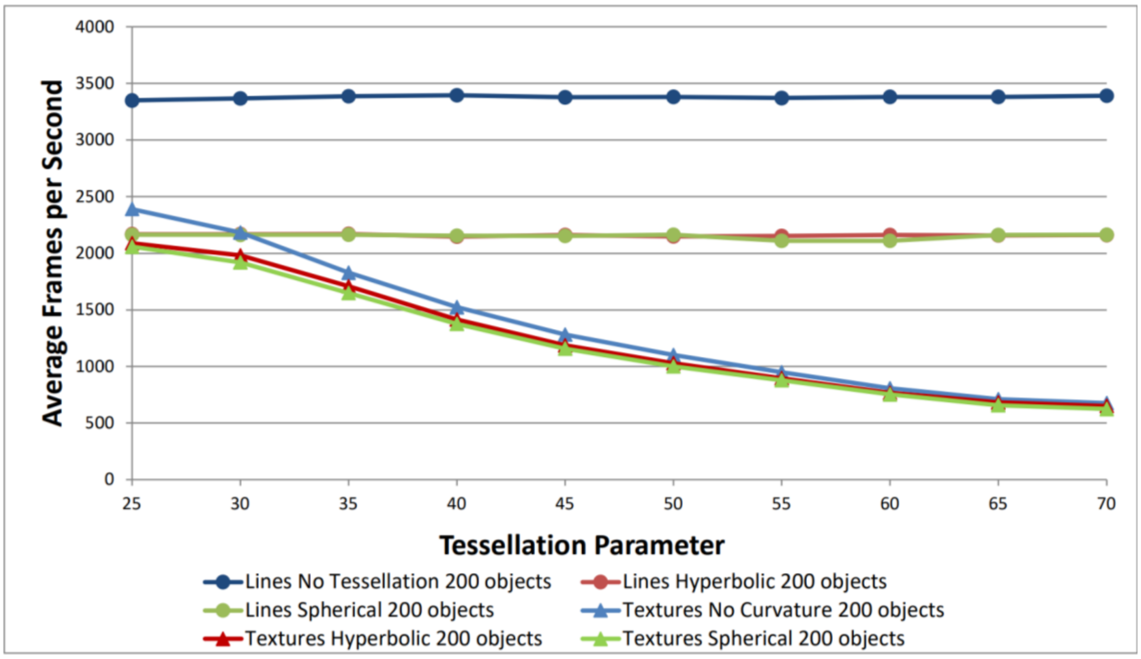


Figure 10.3: Performance test with 200 quadrilateral objects rendered with increasing tessellation parameter.

Figure 10.4 shows the data from a test with 400 objects. The engine was rendering objects without tessellation at approximately 2,500 FPS. Hyperbolic and spherical line tessellation was rendered at approximately 1,400 FPS regardless of the tessellation parameter value. textures were rendered at approximately 1,400 FPS in Euclidean and hyperbolic space for tessellation parameter of 25, but in spherical space they were rendered at approximately 1,300 FPS. As tessellation parameter increased, the performance decreased to a minimum value of approximately 400 FPS.

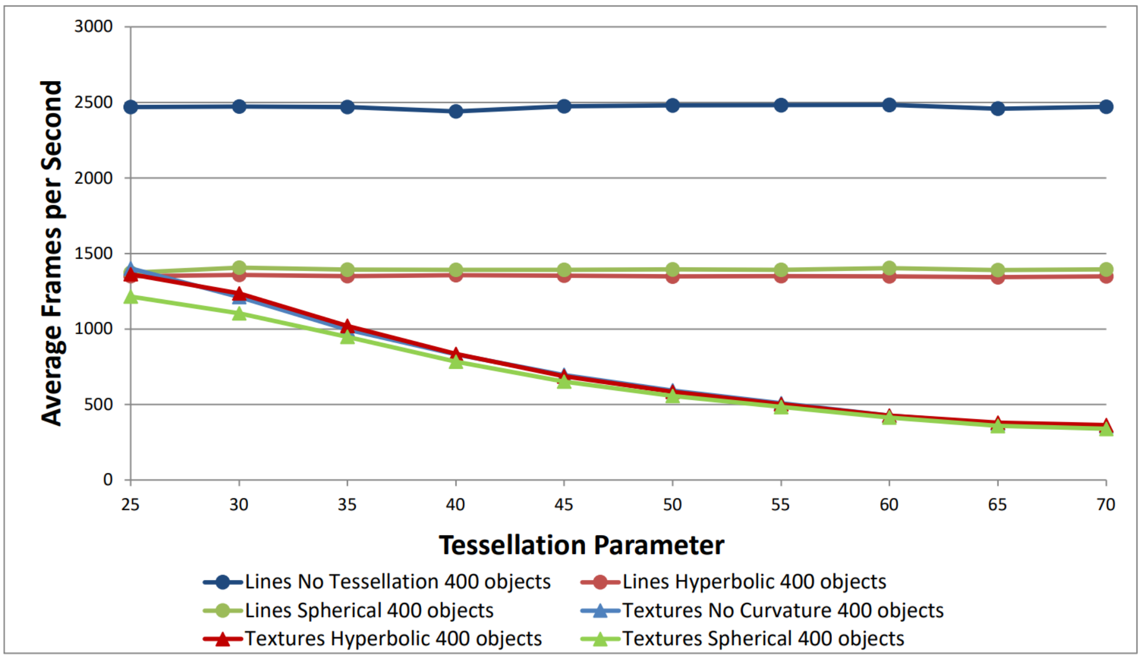


Figure 10.4: Performance test with 400 quadrilateral objects rendered with increasing tessellation parameter.

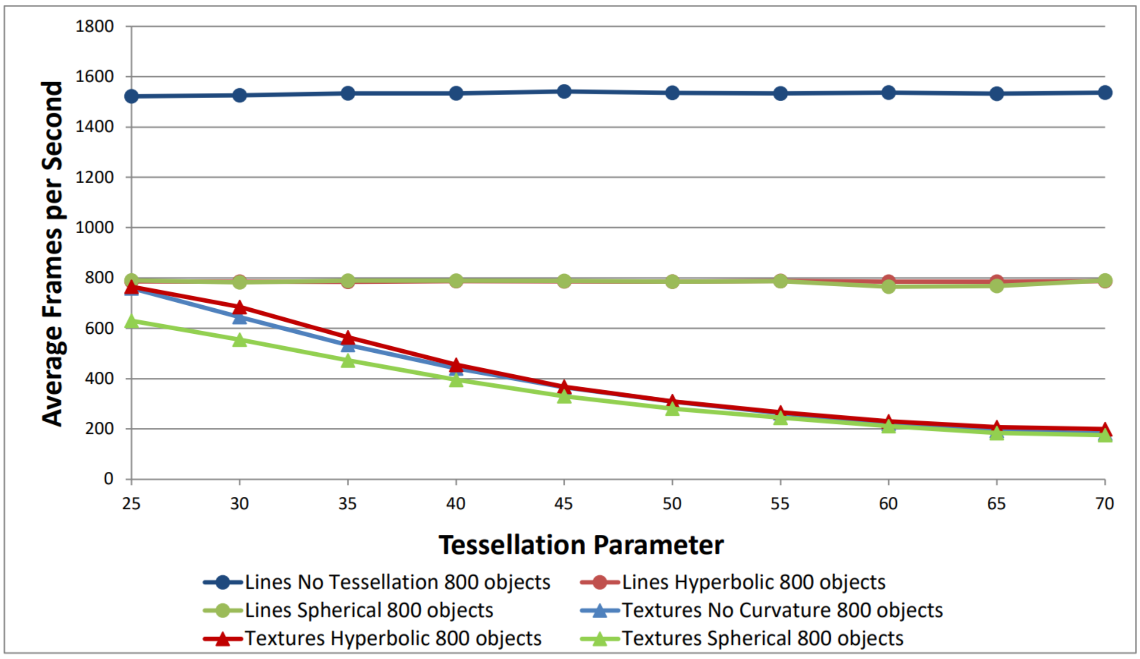


Figure 10.5: Performance test with 800 quadrilateral objects rendered with increasing tessellation parameter.

Figure 10.5 shows the data from a test with 800 objects. The engine was rendering objects without tessellation at approximately 1,500 FPS. Hyperbolic and spherical line tessellation was rendered at approximately 800 FPS regardless of the tessellation parameter value. Euclidean and hyperbolic texture rendering approaches are performing at approximately 800 FPS and spherical texture rendering approach at approximately 600 FPS when tessellation parameter is 25. The performance drops gradually as the tessellation parameter increases to a minimum value of approximately 200 FPS for all texture rendering approaches.

For the line rendering approaches, the tessellation parameter determines the number of segments on each line. For the texture rendering approaches, the tessellation parameter sets both the inner and outer tessellation values, so the number of texture coordinates does not increase linearly with the increased tessellation. If all inner and outer tessellation parameters are $\boldsymbol{n}$, the number of texture coordinates throughout a patch is:

If $\boldsymbol{n}$ is even, then:

$$\frac{\mathbf{3}\boldsymbol{n}^{\mathbf{2}}}{\mathbf{4}} + \frac{\mathbf{3}\boldsymbol{n}}{\mathbf{2}} + \mathbf{1} \tag{10.1}$$

If $\boldsymbol{n}$ is odd, then:

$$\frac{\mathbf{3}\boldsymbol{n}^{\mathbf{2}}}{\mathbf{4}} + \frac{\mathbf{3}\boldsymbol{n}}{\mathbf{2}} + \frac{\mathbf{3}}{\mathbf{4}} \tag{10.2}$$

The most significant term is of $O(n^2)$ complexity, so the number of texture coordinates increases with a square of the tessellation parameter.

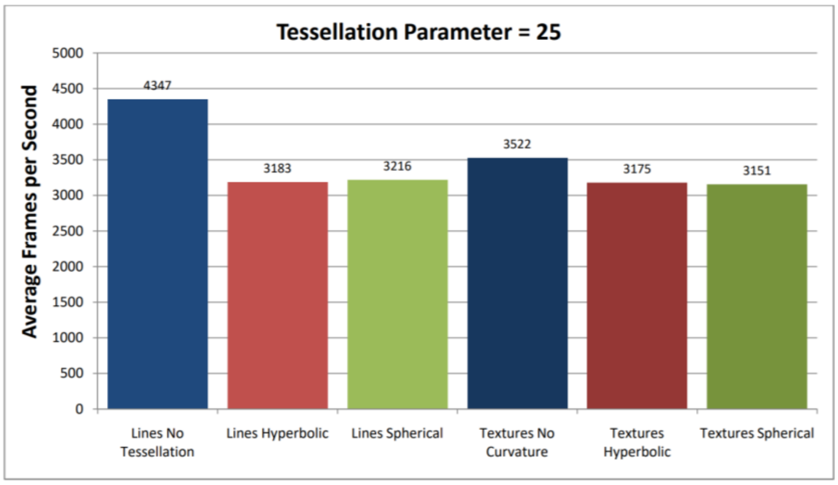


Figure 10.6: Average FPS values for the test with 100 shapes and tessellation parameter 25.

Figure 10.6 shows the average FPS values per approach for tessellation parameter of 25 in the test with 100 shapes.

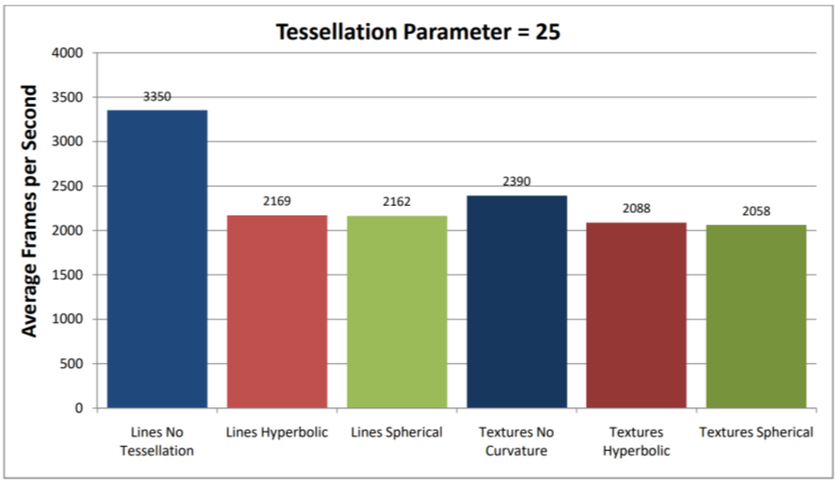


Figure 10.7: Average FPS values for the test with 200 shapes and tessellation parameter 25.

Figure 10.7 shows the average FPS values per approach for tessellation parameter of 25 in the test with 200 shapes. No tessellation baseline decreased approximately 23% compared to the test with 100 shapes, while all other approaches decreased approximately 32-35%.

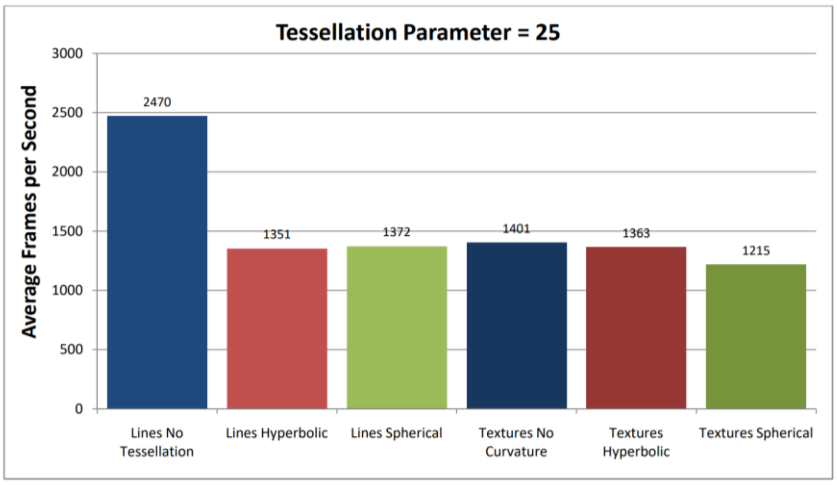


Figure 10.8: Average FPS values for the test with 400 shapes and tessellation parameter 25.

Figure 10.8 shows the average FPS values per approach for tessellation parameter of 25 in the test with 400 shapes. No tessellation baseline decreased approximately 26% compared to the test with 200 shapes, while all other approaches decreased approximately 36-40%.

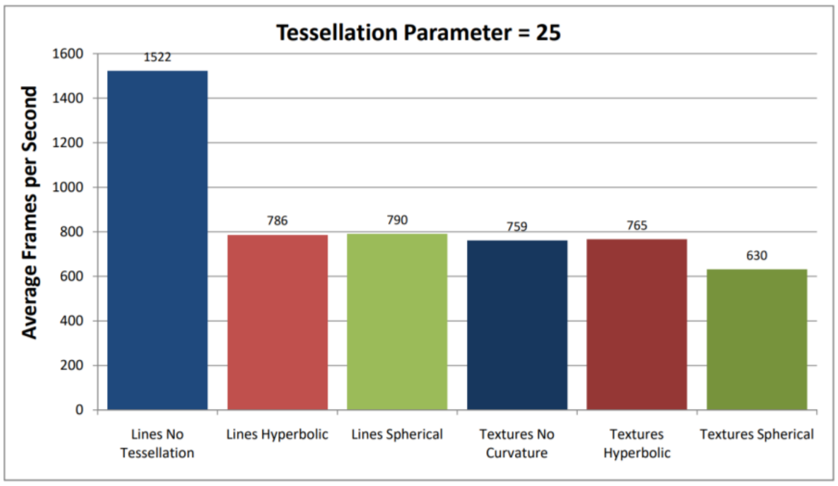


Figure 10.9: Average FPS values for the test with 800 shapes and tessellation parameter 25.

Figure 10.9 shows the average FPS values per approach for tessellation parameter of 25 in the test with 800 shapes. No tessellation baseline decreased approximately 38% compared to the test with 200 shapes, while all other approaches decreased approximately 42-48%.

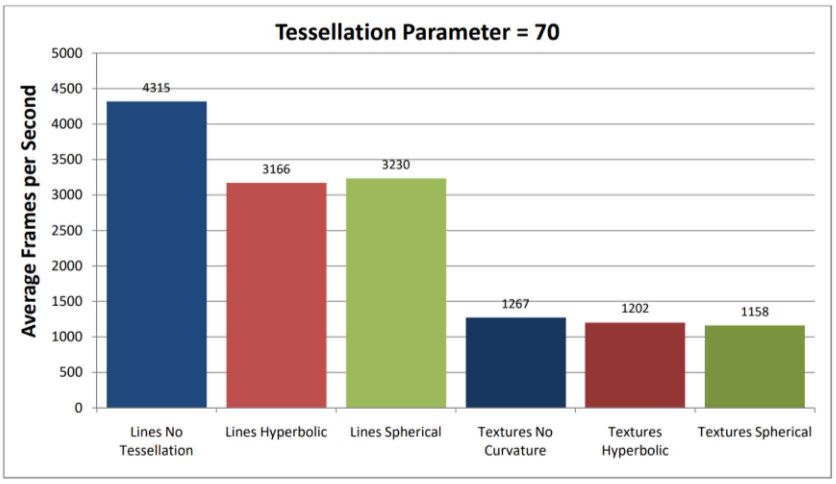


Figure 10.10: Average FPS values for the test with 100 shapes and tessellation parameter 70.

Figure 10.10 shows average FPS values per approach for tessellation parameter value of 70 in the test with 100 shapes.

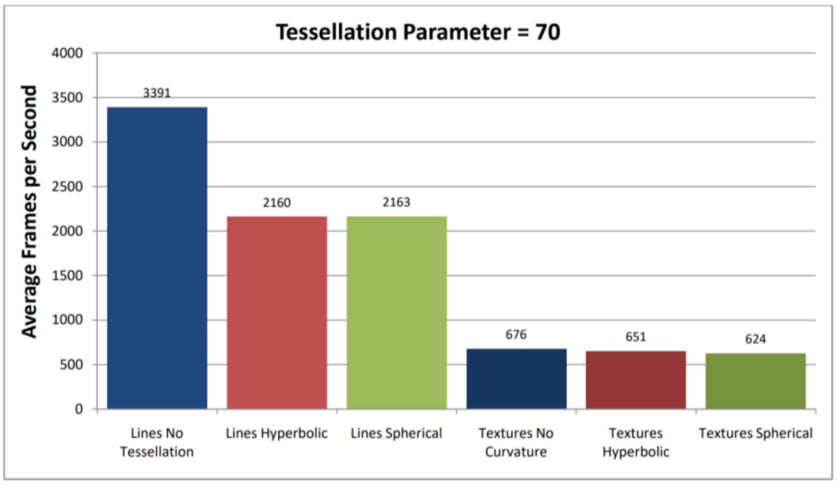


Figure 10.11: Average FPS values for the test with 200 shapes and tessellation parameter 70.

Figure 10.11 shows average FPS values per approach for tessellation parameter value of 70 in the test with 200 shapes. No tessellation baseline decreased approximately 22% compared to the test with 200 shapes, spherical and hyperbolic line rendering approaches decreased in FPS approximately 31-33%, which is similar to the decrease measured in the test for tessellation parameter value of 25. Texture rendering approaches decreased approximately 46%.

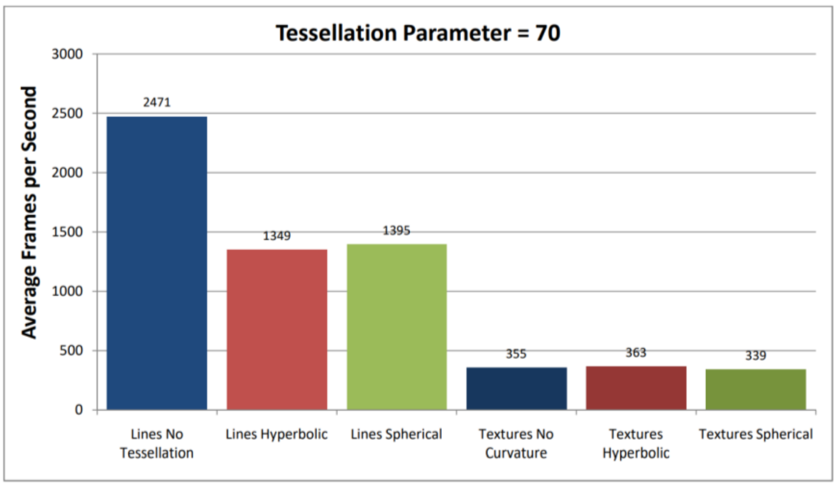


Figure 10.12: Average FPS values for the test with 400 shapes and tessellation parameter 70.

Figure 10.12 shows average FPS values per approach for tessellation parameter value of 70 in the test with 400 shapes. No tessellation baseline decreased approximately 27% compared to the test with 200 shapes, spherical and hyperbolic line rendering approaches decreased in FPS by approximately 36-38%. Texture rendering approaches decreased approximately 44-47%.

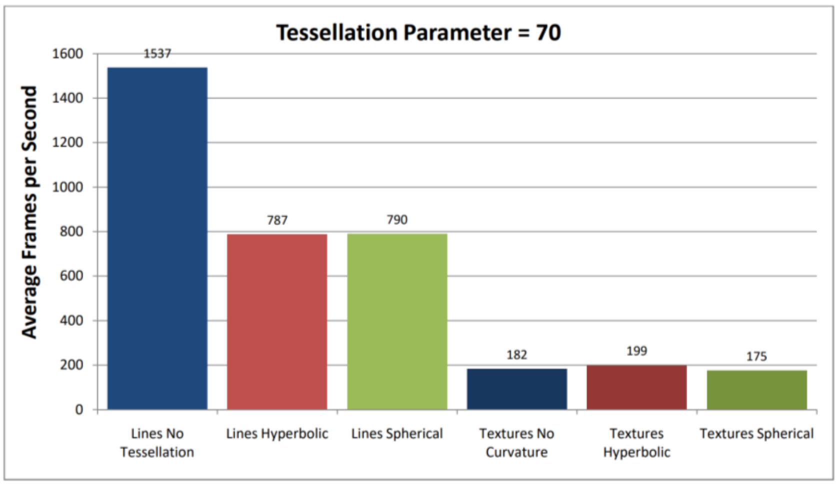


Figure 10.13: Average FPS values for the test with 800 shapes and tessellation parameter 70.

Figure 10.13 shows average FPS values per approach for tessellation parameter value of 70 in the test with 800 shapes. No tessellation baseline decreased approximately 38% compared to the test with 200 shapes, spherical and hyperbolic line rendering approaches decreased in FPS by approximately 42-43%. Texture rendering approaches decreased approximately 45-48%.

Euclidean tessellation for the baseline test was designed to calculate the same number of intermediate points as the non-Euclidean approaches described in chapter 9. As such, the shape was split into patches (one per edge), global vertex coordinates were found and subsequently the coordinates of intermediate points throughout the shape were calculated. The difference between Euclidean and non-Euclidean texturing was only in the types the calculations performed. For non-Euclidean texturing spherical and hyperbolic trigonometry has been used as described in chapter 9. For Euclidean texturing baseline cartesian vector-based calculations have been used instead. The tests show that there is little difference in average FPS values between Euclidean and non-Euclidean texturing approaches, meaning that the overhead from trigonometric operations on the GPU is negligible.

From the bar charts, it is clear that the values for all approaches stay proportionally the same with an increasing number of shapes. The engine still runs at 200 FPS with 800 textured shapes, which is more than enough for any 2D game or simulation. Furthermore, this could even be extended to 3D. While the 3D meshes consist of multiple triangles, they would not all need to be individually tessellated, because of being very small already. Only larger flat surfaces would need to be tessellated, while smaller triangles in the mesh could be transformed similarly to how texture coordinates in the method for 2D shapes are transformed (described in chapter 9).

The FPS achieved when testing texture rendering approaches is well suited for real-time game applications, which should run at a stable 60 FPS minimum. In recent years, a lot of games have moved to higher frame rates due to most displays supporting 144Hz format. The GPU approaches exceed both of these limits even when a large number of objects are rendered on the screen and a high tessellation parameter is chosen for each line. The number of shapes in a 2D game is likely to be much lower than 800. As shown on Figure 10.1, the screen gets cluttered and it becomes hard to distinguish individual shapes when there are that many rendered at once. Additionally, the tessellation parameter can be adjusted depending on the size of the shape and the position of the shape in the projection. Smaller shapes and shapes closer to the reference point of the projection require smaller tessellation parameter, because curvature does not affect them as strongly.

10.1 Summary

This chapter has described the testing performed to compare line and texture rendering approaches for non-Euclidean geometry. The data gathered from the test has been analysed and it shows that the overhead from the non-Euclidean texturing depends on the tessellation parameter. At low enough values the overhead is negligible when compared to the line rendering. At high values of the tessellation parameter the overhead is more noticeable, but the engine can still perform well in real-time even when a large number of shapes is being rendered at the same time.

11. Survey: Games in non-Euclidean Environment

To research the interest in the features developed in the described software, it was decided to re-create well-known 2D games. These games would then be available for people to play and subsequently fill out the questionnaires about their experience.

At the time of performing the survey, the non-Euclidean texturing has not been implemented within the engine, hence the games used for the survey were rendered using line rendering described in chapters 4, 5 and 7.

This chapter contains original research in re-creating classic 2D games in non-Euclidean engine (described in chapters 4, 5 and 7); setting up a user study to study the player experience playing games in non-Euclidean environment; analysing and discussing the player experience data gathered by the user study.

11.1 Games Developed – non-Euclidean Asteroids

The first game developed using the non-Euclidean rendering engine was an implementation of the classic arcade game Asteroids. It fit well with the capabilities of the engine and major shapes (i.e., spaceship, lines and asteroids) have already been implemented in the software testing, as illustrated in Figure 5.8. Figure 11.1, Figure 11.2 and Figure 11.3 show time-lapses of the Asteroids game in Euclidean, Spherical and Hyperbolic space respectively.

These timelapses have been created by collating multiple screenshots of the game. These screenshots have been taken and stored by the engine. One screenshot was taken every second of the game running. The spaceship character, controlled by the player, represented by a red quadrilateral, arrow-like shape does not move in these time-lapses for consistency in movement between planar, spherical and hyperbolic timelapses. The asteroids' initial positions, movement speed and trajectories are randomised using a `rand()` function. However, for the purposes of time-lapses, the initial seed value has been set to a hard-coded value in order to get identical initial parameters.

Grid lines are rendered as separate objects and trace the geodesics through the curved space. They help visualise the curvature of the space and choose the correct trajectory when playing the game.

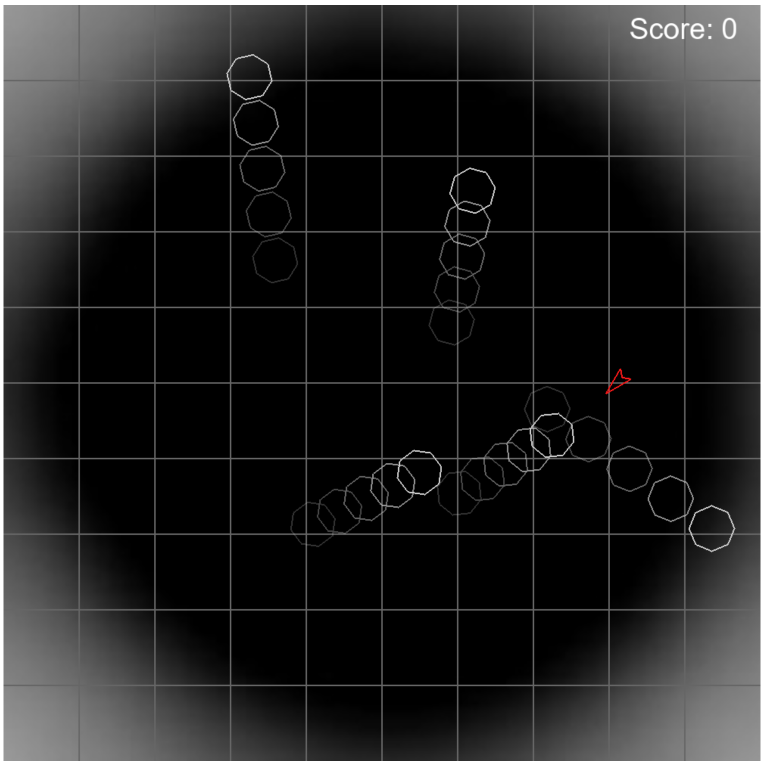


Figure 11.1: Time-lapse of the Asteroids game in Euclidean space rendered with the engine. 5 Asteroids move at different speed. The player's spaceship (red) remains stationary.

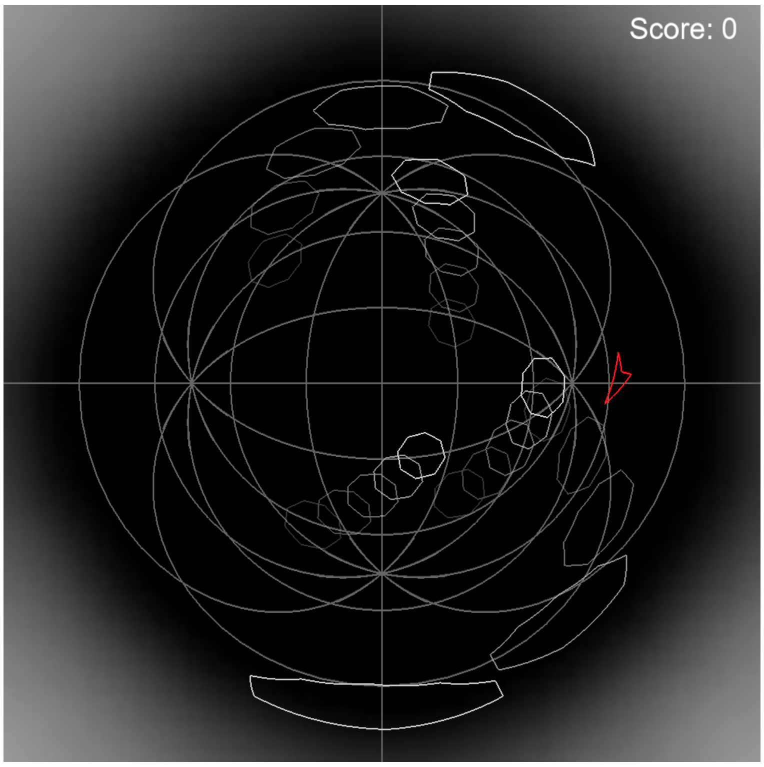


Figure 11.2: Time-lapse of the Asteroids game in Spherical space rendered with the engine. 5 Asteroids move at different speed. The player's spaceship (red) remains stationary.

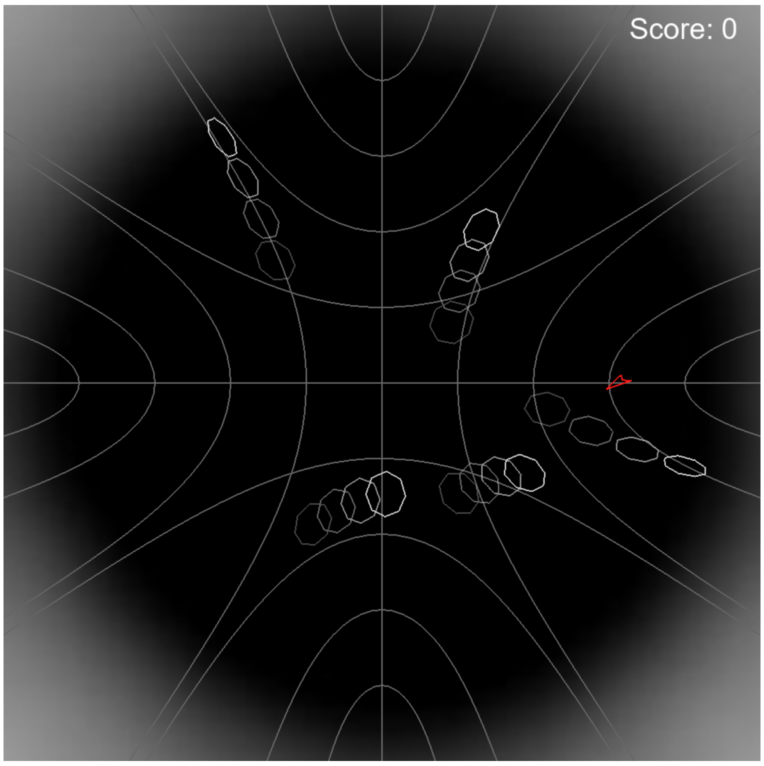


Figure 11.3: Time-lapse of the Asteroids game in Hyperbolic space rendered with the engine. 5 Asteroids move at different speed. The player's spaceship (red) remains stationary.

Some additional game logic had to be implemented to develop the Asteroids game.

Proximity-based collision detection has been implemented using the bounding circle method. It is a ubiquitous, if simple, method of collision detection, explained well in an article by Jeffrey Thompson (Thompson, 2015). A bounding circle radius is assigned to each object, i.e. spaceship, asteroids and bullets.

Circle-circle collision detection works by finding the distance between the centres of two bounding circles and comparing it to the sum these circles' radiuses. Let $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$ be centre points of two bounding circles with radiuses R_1 and R_2 respectively, the collision is found if:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} < R_1 + R_2 \tag{11.1}$$

This is often optimised by squaring the sum of radiuses instead of taking the square root on the left-hand side of the inequality:

$$(x_2 - x_1)^2 + (y_2 - y_1)^2 < (R_1 + R_2)^2 \tag{11.2}$$

However, due to the engine working with the polar coordinate system, the distance calculation had to be adapted to use the cosine rule. Let $P_1(r_1, \theta_1)$ and $P_2(r_2, \theta_2)$ be centre points of two bounding circles with radiuses R_1 and R_2 respectively, in Euclidean space the collision is found if:

$$r_1{}^2 + r_2{}^2 - 2\, r_1 r_2 \cos \Delta\theta < (R_1 + R_2)^2 \qquad \textit{(11.3)}$$

Likewise, collision detection in spherical and hyperbolic spaces uses spherical and hyperbolic cosine rules.

If **K** > 0, then:

$$\cos^{-1}(\cos r_1 \cos r_2 + \sin r_1 \sin r_2 \cos \Delta\theta) < R_1 + R_2 \qquad \textit{(11.4)}$$

If **K** < 0, then:

$$\cosh^{-1}(\cosh r_1 \cosh r_2 - \sinh r_1 \sinh r_2 \cos \Delta\theta) < R_1 + R_2 \qquad \textit{(11.5)}$$

Note: it is possible to speed up the calculations for collision detection in spherical and hyperbolic space, by pre-computing $\cos\,(R_1 + R_2)$ or $\cosh\,(R_1 + R_2)$ for equations *(11.4)* and *(11.5)* respectively. The equations would then change to:

If **K** > 0, then:

$$\cos r_1 \cos r_2 + \sin r_1 \sin r_2 \cos \Delta\theta < \cos\,(R_1 + R_2) \qquad \textit{(11.6)}$$

If **K** < 0, then:

$$\cosh r_1 \cosh r_2 - \sinh r_1 \sinh r_2 \cos \Delta\theta < \cosh\,(R_1 + R_2) \qquad \textit{(11.7)}$$

However, this would require storing additional data, especially if there is a large number of shapes of different sizes, as each combination of radii would need to be precomputed and subsequently looked up from some type of a data structure.

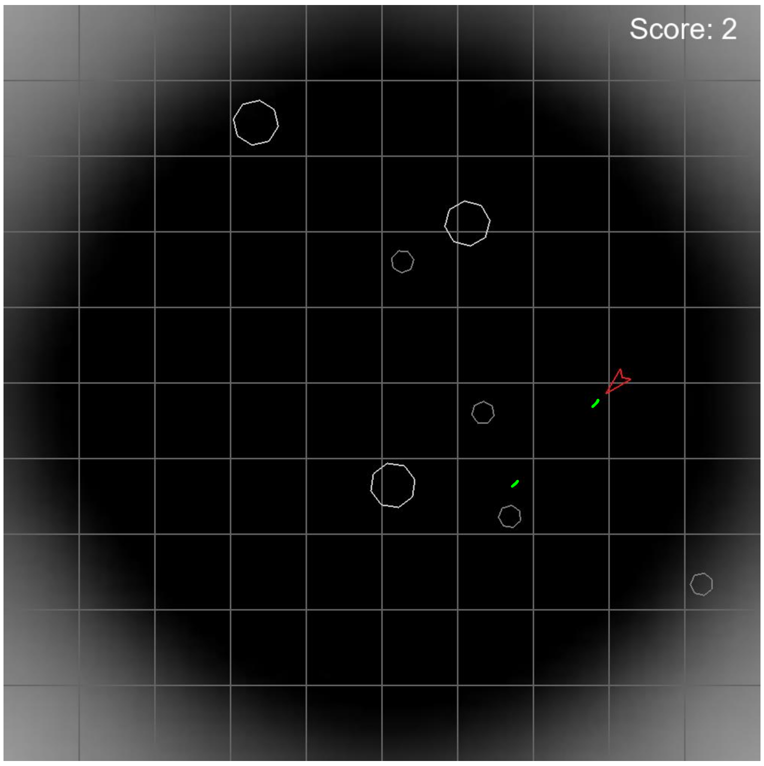


Figure 11.4: Screenshot of the playthrough of the implemented Asteroids game in Euclidean space. Asteroids split into 2 smaller asteroids when hit by the player's bullets (green dashes).

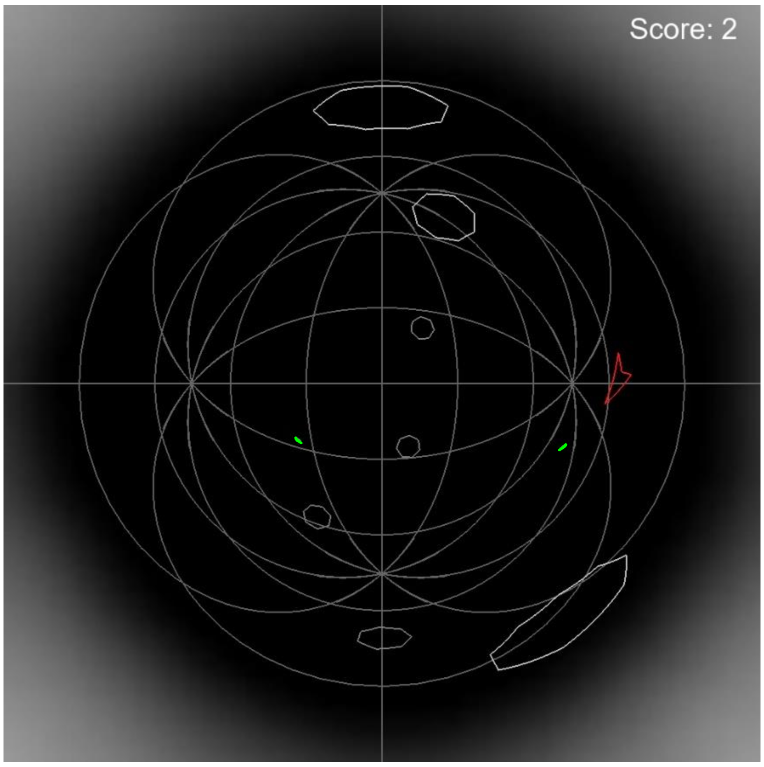


Figure 11.5: Screenshot of the playthrough of the implemented Asteroids game in spherical space. Asteroids split into 2 smaller asteroids when hit by the player's bullets (green dashes).

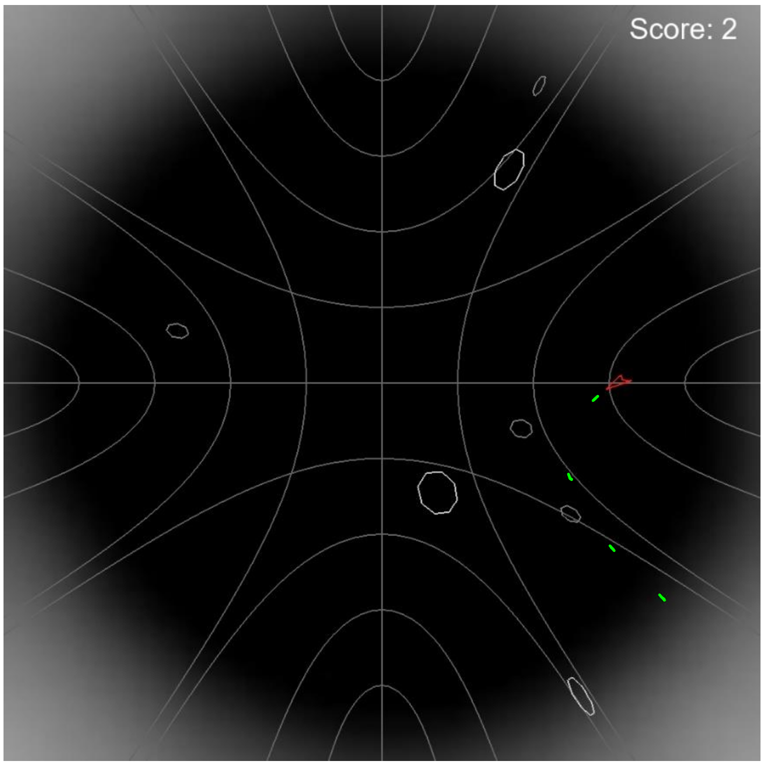


Figure 11.6: Screenshot of the playthrough of the implemented Asteroids game in hyperbolic space. Asteroids split into 2 smaller asteroids when hit by the player's bullets (green dashes).

In addition to collision detection, the logic for shooting with the player's spaceship and splitting of the asteroids into smaller asteroid shards has been implemented. Figure 11.4, Figure 11.5 and Figure 11.6 illustrate the non-Euclidean Asteroids game being in planar, spherical and hyperbolic space respectively.

Each screenshot is showing the player shooting bullets, displayed as green dashes, which follow the geodesic passing through the player's position in the direction the player is facing.

The asteroids split into two smaller asteroid shards when a collision is detected between the said asteroid and player's bullets. There are two levels of smaller asteroids, as in the original Asteroids game. Also, each time a collision between an asteroid (or an asteroid shard) and bullets is detected, the score is incremented.

If a player collides with the asteroid (or an asteroid shard), a game over screen is displayed.

11.2 Games Developed – non-Euclidean Snake

The second game implemented within the non-Euclidean engine, was chosen to be Snake. It is a popular 2D game with a top-down view, which is a good fit for the capabilities of the engine.

Some of the features implemented for the Asteroids game have been used to build Snake, including collision detection, object position randomisation and score functionality.

However, Snake does not move in a continuous simulation, instead it moves in increments of the size equal to the single section of the Snake. This step-based simulation has been implemented to update the Snake's head position following the same method as described in Chapter 5, passing in the distance travelled equal to snake's section size, rather than calculating it from the object's velocity. Instead of running the update every game loop iteration, it is run once the trigger timer runs out. This trigger time determines the difficulty of the game, making the snake move faster or slower depending on its value. The trigger timer is reset once the step update is run.

Another design choice which needed to be made when implementing the snake's movement was game controls. In the classic implementation of the Snake game, up, down, right and left keys are used to turn the Snake with respect to the screen. Thus, regardless whether the snake is moving up or down the screen, left key input turns the snake towards the left edge of the screen. There are only 4 directions the snake can move in and it makes it intuitive for the player to control the snake.

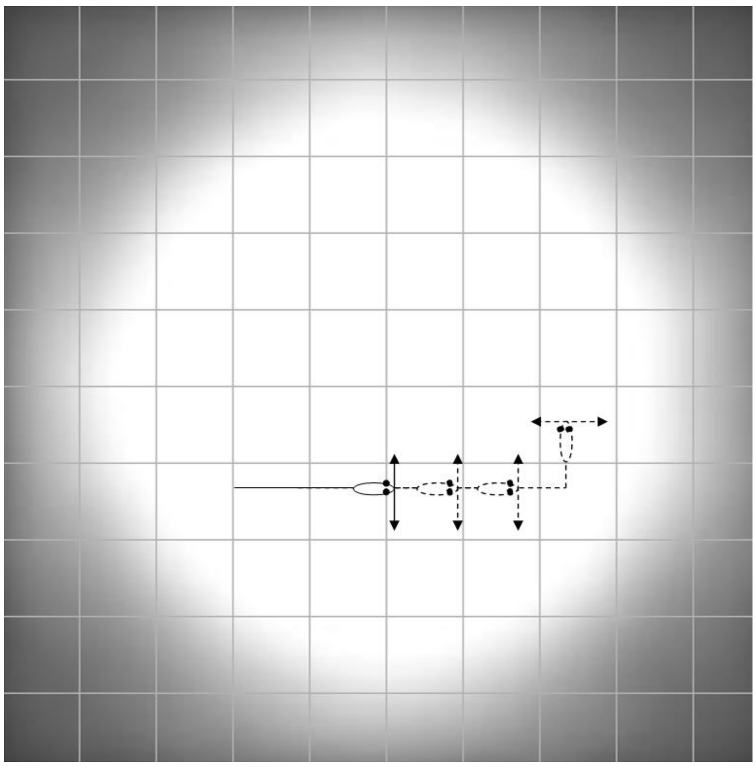


Figure 11.7: Snake's movement on a non-curved 2D plane. Forward movement trajectory followed by a turn to the left.

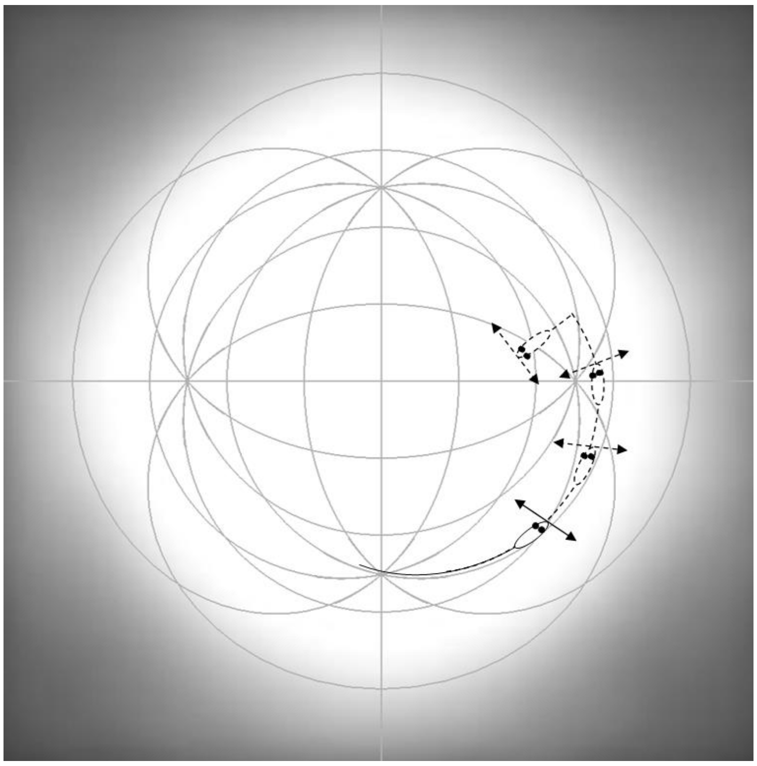


Figure 11.8: Snake's movement on a positively curved 2D plane. Forward movement trajectory followed by a turn to the left.

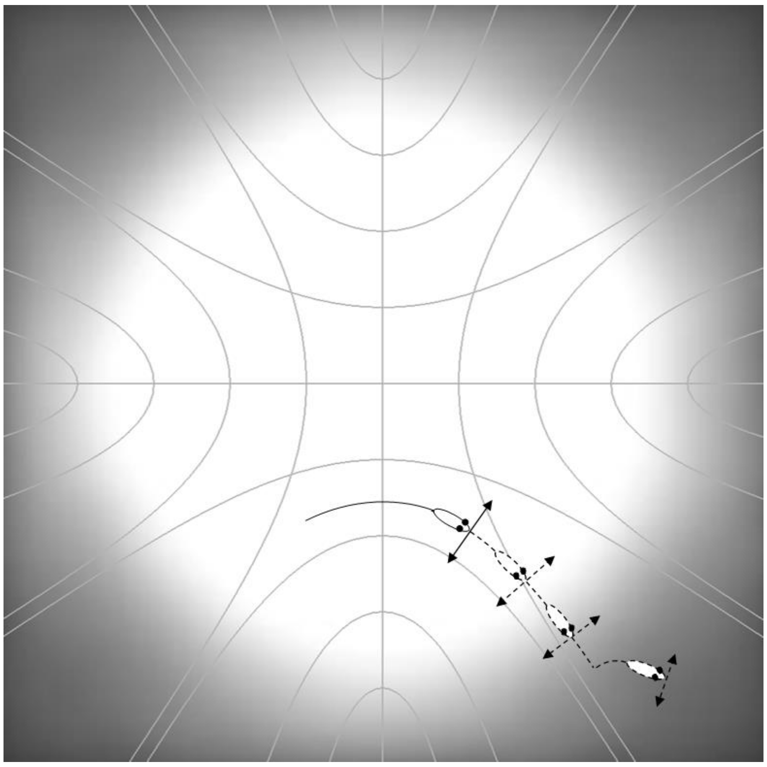


Figure 11.9: Snake's movement on a positively curved 2D plane. Forward movement trajectory followed by a turn to the left.

However, in Spherical and Hyperbolic environments, due to the shape of the space, the snake could move in any direction on the screen. So, to keep the controls consistent between the different spaces in the game, it was decided to turn the snake with respect to itself, rather than the screen. Regardless of where the snake is on the screen, the left key input would turn the snake 90 degrees to the left. This is illustrated in Figure 11.7, Figure 11.8 and Figure 11.9 for Euclidean, spherical and hyperbolic spaces respectively.

A snake is made up of multiple game objects, one for each section of the snake. To reduce the number of movement calculations, the position update function is run for the head section of the snake. After that, the previous positions are propagated through the snake's sections. Meaning that the previous position of the head is assigned to the next snake's section and so on until the tail section is updated. The snake grows in size by 1 section, when a collision between the snake's head and the food is detected. The score is updated when the food is consumed and a new position is randomised for it. Figure 11.10 Figure 11.11 and Figure 11.12 show the time-lapse of the non-Euclidean snake game's execution in Euclidean, Spherical and Hyperbolic space respectively. Each image within a timelapse is generated every 3 position updates of the snake. After the 5th position update (2nd update after the second screenshot taken) the input has been generated to turn the snake to the right.

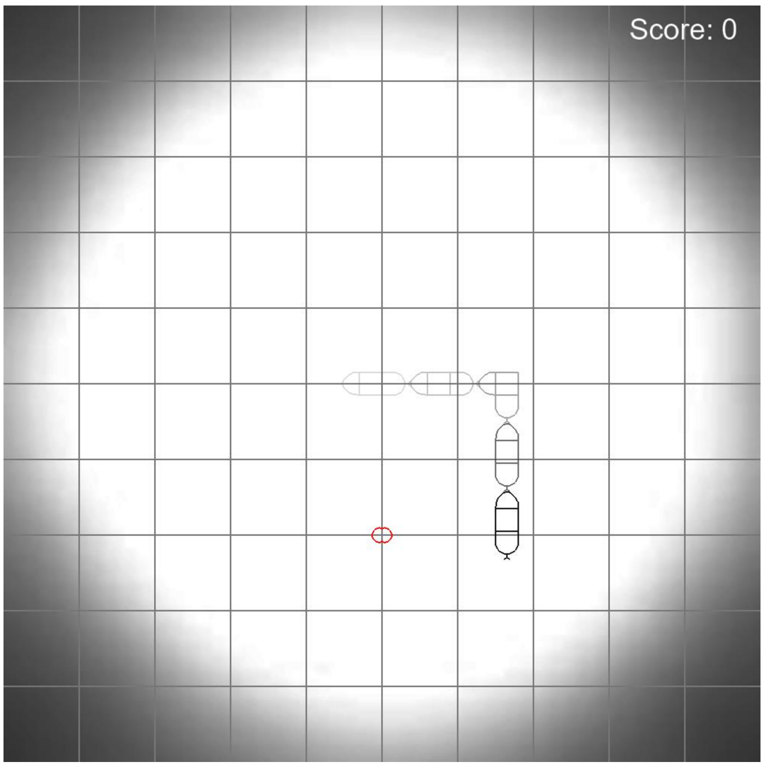


Figure 11.10: Time-lapse of the snake game in Euclidean space rendered with the engine. The snake moves forward and makes a turn to the right, toward the food (red shape).

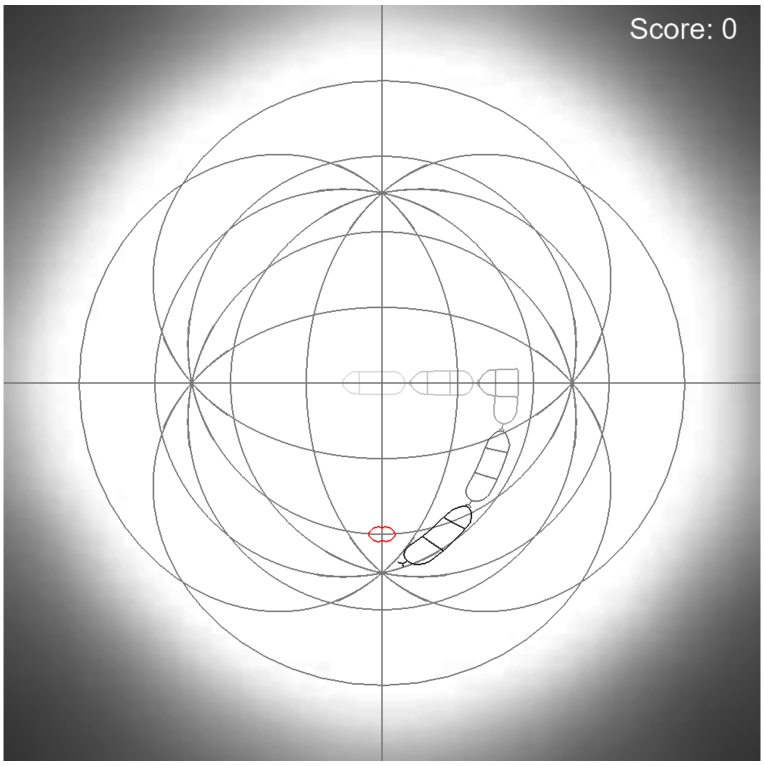


Figure 11.11: Time-lapse of the snake game in Spherical space rendered with the engine. The snake moves forward and makes a turn to the right, toward the food (red shape).

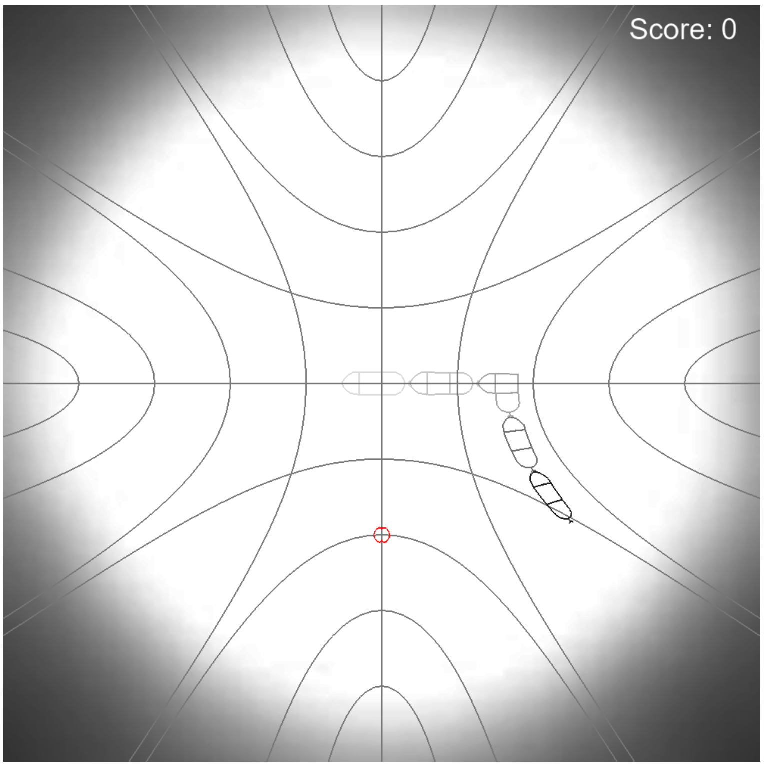


Figure 11.12: Time-lapse of the snake game in Hyperbolic space rendered with the engine. The snake moves forward and makes a turn to the right, toward the food (red shape).

11.3 Qualtrics Survey

Qualtrics has been chosen as a tool for building the survey (Qualtrics, 2021). It has good reviews (TrustRadius, 2014), is intuitive, but also allows for a range of powerful features, like survey flow discussed in section 11.3.2.

11.3.1 Aims

An engine has been developed which can calculate and render objects/shapes in a non-Euclidean environment. The aim of this study was to find out how the non-Euclidean Environment impacts the gameplay.

The main aspects that were explored in terms of gameplay:

- How easy is it to understand and adjust to the non-Euclidean environment in a familiar game?
- Does the extra complexity change the gameplay in a positive (more interesting, challenging, enjoyable) or negative (confusing, frustrating) direction?

11.3.2 Structure of the Survey

The survey was split into a number of sections: Demographics, Asteroids, Snake and end of survey sections.

The demographics section contained questions about the participant's age and gender (which they could skip if they were not comfortable answering those questions) as well as questions that gauged the participants familiarity with video games in general and Asteroids and Snake specifically.

In the Asteroids and Snake sections the participants were tasked to play the respective game for 5 minutes before answering the questionnaire. This step was repeated three times for each game, asking the participants to play the game in Euclidean, Spherical and Hyperbolic space.

To reduce participants' bias, the order in which the participants were presented with the two games and subsequently the three curvatures of space was randomised. Figure 11.13 displays the flowchart showing the structure of the survey with a Randomizer determining the order in which participants received the Snake and Asteroids sections of the survey; and subsequently another Randomizer determining the order of the curvature-based subsections for each game. The screenshot of the survey flow tool taken from Qualtrics is included in Appendix A.

After participants have played a game in all three curvatures of space, they were presented with a summary subsection for the game and asked to fill in a number of free-text questions.

After participants have played both games, they were presented with the end of survey section asking them to compare their experience playing the two games.

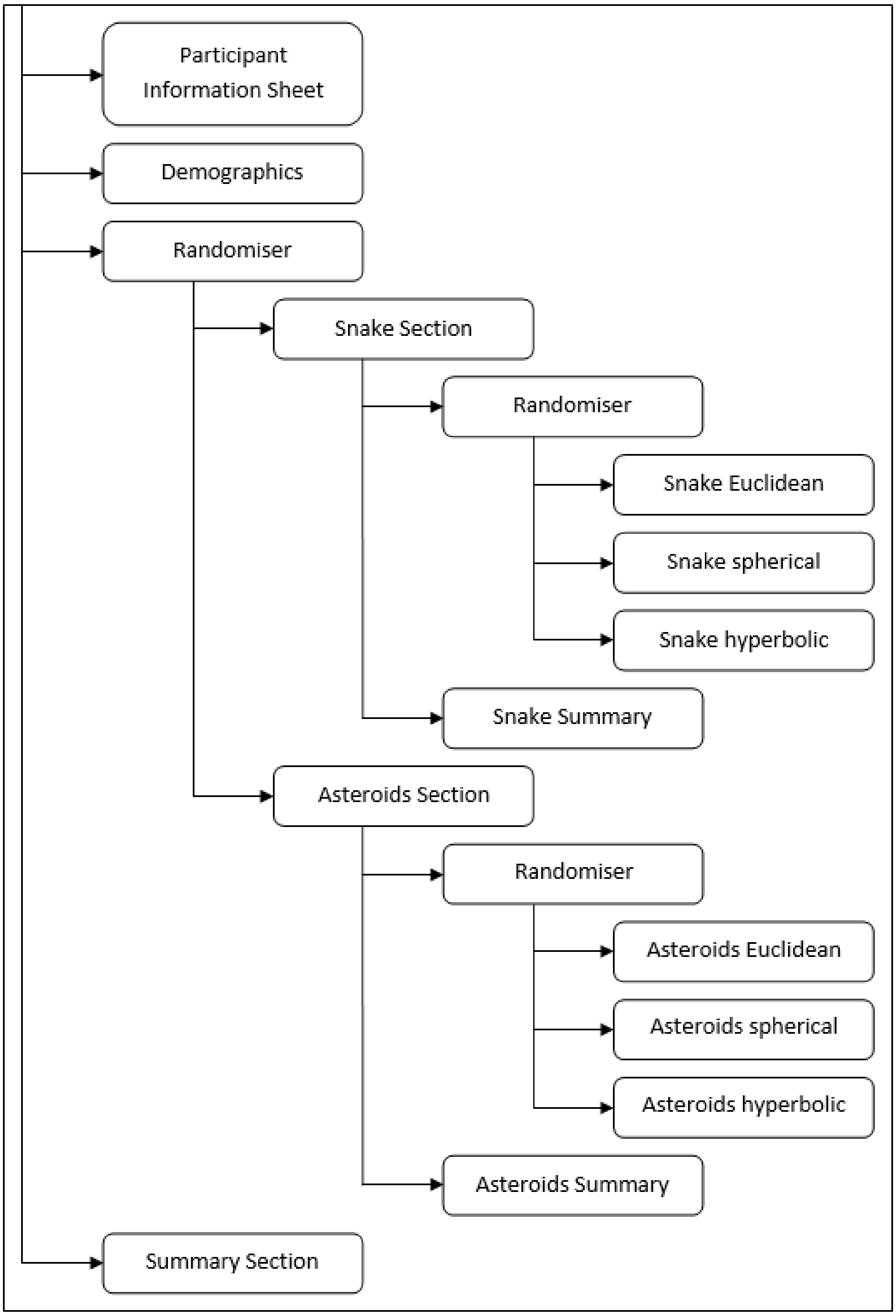


Figure 11.13: Flowchart showing the structure of the survey. Randomizer blocks display the sections in a random order, which helps to reduce participants' bias.

Game controls were explained in the task description before each part of the questionnaire. Figure 11.14 and Figure 11.15 illustrate the game controls provided for participants.

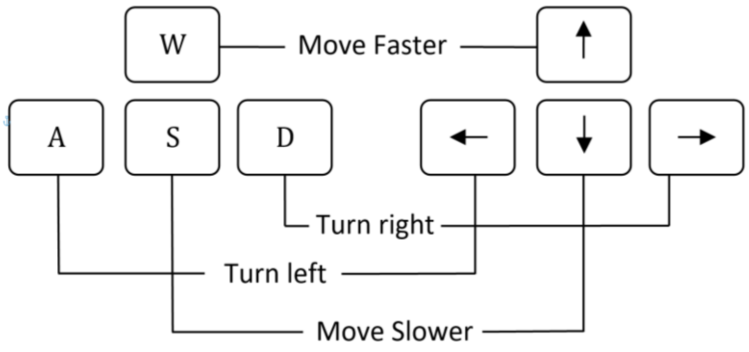


Figure 11.14: Game controls for Non-Euclidean Snake

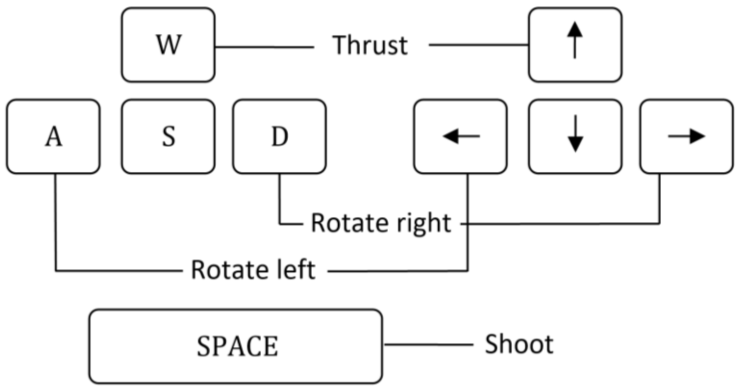


Figure 11.15: Game controls for Non-Euclidean Asteroids. The thrust button adds acceleration in the direction the player controlled object is facing.

11.3.3 Questionnaire

The survey is structured with set a number of constraints on the questionnaire. Because the questionnaire had to be repeated six times (once for each subsection of two games), the number of questions had to be kept small to not make the survey too long. Overly long surveys tend to have a low participation and especially completion rate (Galesic & Bosnjak, 2009).

Three experience-based questionnaires were considered: IEQ, The Immersive Experience Questionnaire (Jennett, et al., 2008); GEQ, The Games Experience Questionnaire (Ijsselsteijn, de Kort, & Poels, 2013); PXI Bench, The Player Experience Inventory Bench (Haider, Gerling, & Vanden Abeele, 2020); and UPEQ, Ubisoft Perceived Experience Questionnaire

(Azadvar & Canossa, 2018). Several factors were considered when evaluating the potential of each questionnaire in this context.

The issue with UPEQ questionnaire used by Ubisoft was that it did not focus on the experiences this survey was interested in, namely: player interest, immersion and how easy is the game to understand and control.

IEQ would have measured the experiences this study was interested in; however, it only provides valid results if all 31 items are used in the questionnaire. This would have made the questionnaire part of the survey 186 questions in total, which is unreasonably long.

GEQ and PXI are both modular, so only the most relevant experiences could be measured, thus reducing the length of the questionnaire. The problem with GEQ is that it is not validated and as such does not have the best rating amongst games researchers (Law, Brühlmann, & Mekler, 2018).

Hence, PXI was the questionnaire used in this study. It is a broad enough questionnaire that it allowed the measurement of different facets of player experience but also it has several sub-scales inside it, so the questionnaire could include only a number of items that were the most relevant, without losing the validity of the measurement.

PXI Bench is designed with 10 constructs measuring different aspects of player experience. These are divided into two groups, functional experience (Ease of control, Progress feedback, Audio-visual appeal, Clarity of goals and rules and Challenge) and psychological experience (Mastery, Curiosity, Immersion, Autonomy and Meaning). Each of the constructs has three items associated with it, which ask similar questions in subtly different ways, ensuring reliability of the measurement.

Not all of the constructs were relevant to the study, which helped reduce the total number of items in the questionnaire from 30 to 12. It was decided to focus on Controls, Challenge, Mastery and Immersion. Each construct is measured via a 7-point Likert (Likert, 1932):

Strongly disagree	Disagree	Somewhat disagree	Neither Agree nor Disagree	Somewhat agree	Agree	Strongly Agree

11.4 User Study

The aim of this research was to explore how players perceive the two non-Euclidean environments. To do this, a within-subject design user study was conducted with 22 players to assess player experience of mastery, controls, immersion and challenge when playing three versions of Snake and Asteroids. Qualitative responses were also collected from players to learn about their experiences of and preferences in terms these novel game environments.

11.4.1 Materials

To assess player experience (PX), four out of the ten dimensions of the Player Experience Inventory (PXI) were used (Abeele, Spiel, Nacke, Johnson, & Gerling, 2020). These were deemed most relevant in the context of our research: Ease of Control, Challenge, Immersion and Mastery. Each question in all sub-scales were assessed on a 7-point Likert scale with 1 being anchored to “Strongly Disagree" and 7 - to “Strongly Agree". The PXI was chosen amongst other existing PX questionnaires as it has been validated and used in several recent studies and it allows for the measurement of different facets of PX in a broad sense.

The two games used in the study were the aforementioned Snake and Asteroids. Each game had three versions: one Euclidean and two non-Euclidean versions (hyperbolic and spherical). The games used in the survey are available on GitHub (Osudin, GitHub.com, 2022).

11.4.2 Hypotheses

The following hypotheses were generated based on the review of related literature and the assumptions made about the curved spaces with regards to their effects on player experience of mastery, challenge, control and immersion:

- **H1:** Mastery is more difficult to achieve when playing games with non-Euclidean curvature than games with Euclidean curvature.
- **H2:** Games with non-Euclidean curvature are more difficult to control than games with Euclidean curvature.
- **H3:** Games with non-Euclidean curvature are more immersive than games with Euclidean curvature.
- **H4:** Games with non-Euclidean curvature are more challenging to play than games with Euclidean curvature.

11.4.3 Procedure

To be able to compare the experiences between the different conditions (Euclidean, Spherical and Hyperbolic) in the two games, it was decided to do a within-participant study which would allow players to compare their experiences across games and conditions. It was also important to know whether players had any preferences towards specific conditions, so being able to compare across these conditions was essential.

Hence, a Qualtrics survey was created and distributed that contained:

1) Information sheet and consent form and a demographics section;
2) Instructions for the study and the games that could be played on the participants' personal computers;
3) Demographics questions;
4) Six copies of the four dimensions of the PXI (controls, immersion, challenge and mastery) for each condition in each game;
5) Questions about players' preferences for any of the versions in both games;
6) Open-ended questions to collect comments from players about the games and their experiences with them.

A repeated-measures ANOVA was performed to test for the effects of the manipulations on the four experiences as measured by the PXI. Bonferroni post hoc test was used for multiple comparisons at a significance level of $\alpha = 0.05$.

11.4.4 Participants

48 people took part in the study. The data has been reviewed and cleaned to remove:

1) Incomplete responses;
2) Insincere responses;
3) Responses which were completed too quickly to be genuine.

Overall 22 legitimate responses were included for the analysis. This sample included 16 men, 5 women and a non-binary participant. The average age of the participants in the sample was 30.35 ($SD = 6.80$).

Most participants (12) said that they played video games daily, 6 participants played games several times a week, 3 - several times a month and one several times a year. Five players said that they had played Snake regularly, 12 had played is several times and 4 said that they had never seen this game before. As for Asteroids, players were somewhat less familiar with the game with 6 people saying that had never seen the game before, 2 played it regularly and 14 had played the game several times.

11.4.5 Results

The SPSS Statistics tool (IBM Corp, 2020) has been used to perform inferential statistical tests to measure whether there was any statistically significant difference between the versions of the games within the data collected.

		Snake					**Asteroids**				
	K	M	SD	$F(2,19)$	p	$\eta^2_{partial}$	M	SD	$F(2,19)$	p	$\eta^2_{partial}$
Mastery	0	5.47	1.51				5.31	1.27			
	1	4.71	1.43	4.797	$\mathbf{0.013}^{*}$	0.186	4.47	1.56	7.021	$\mathbf{0.002}^{**}$	0.251
	-1	4.64	1.67				4.38	1.41			
Controls	0	5.68	1.47				5.67	1.25			
	1	5.03	1.41	4.243	$\mathbf{0.021}^{*}$	0.168	4.76	1.38	8.808	$< \mathbf{0.001}^{***}$	0.295
	-1	4.76	1.79				4.85	1.22			
Immersion	0	5.08	1.42				4.85	1.40			
	1	5.38	1.09	1.066	0.353	0.048	5.05	1.45	0.463	0.633	0.022
	-1	5.06	1.51				5.06	1.31			
Challenge	0	5.20	1.26				5.35	1.47			
	1	4.80	1.26	0.818	0.448	0.038	4.64	1.56	2.506	0.094	0.107
	-1	4.91	1.54				4.77	1.71			

Table 11.1: Player experience (Mastery, Controls, Immersion, and Challenge) in each of the three versions (Euclidean, Spherical and Hyperbolic) of Snake and Asteroids. Significant results are shown in bold (for < 0.05, ** for < 0.01 and *** for < 0.001).*

For both Snake and Asteroids, mastery and controls differed significantly between the three conditions: Euclidean, Spherical and Hyperbolic geometry (shown in Table 11.1).

H1: Accepted – Participants felt more masterful when playing Snake with Euclidean physics than with Spherical ($p = 0.041$) but not with Hyperbolic ($p = 0.072$) physics. For Asteroids, players felt more masterful when playing the Euclidean version than the Spherical version ($p = 0.043$) as well as the Hyperbolic ($p = 0.015$).

H2: Accepted – Overall, ease of control was significantly different between the three conditions in both Snake and Asteroids. There were no significant differences between the experiences of control in the three versions of Snake but in Asteroids, players felt more in control in the Euclidean version than in the Spherical ($p = 0.009$) and the Hyperbolic ($p = 0.012$) version.

H3 & H4: Rejected – The two non-Euclidean version of both Snake and Asteroids did not differ significantly in the experiences of mastery or controls.

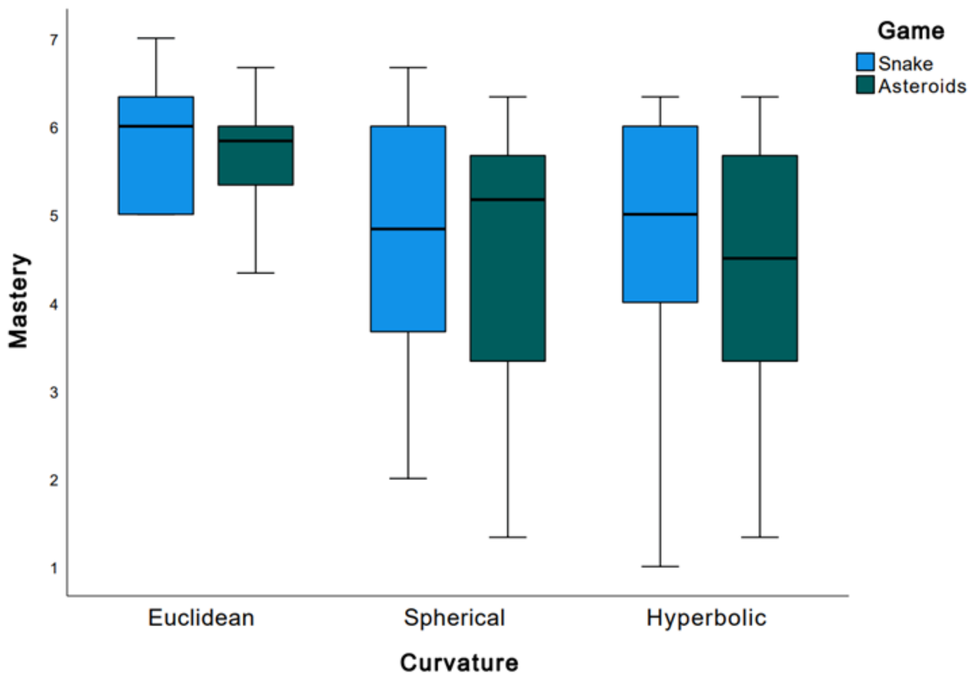


Figure 11.16: Box plot comparing data for the Mastery dimension between Snake and Asteroids in Euclidean (left), spherical (centre) and hyperbolic (right) space.

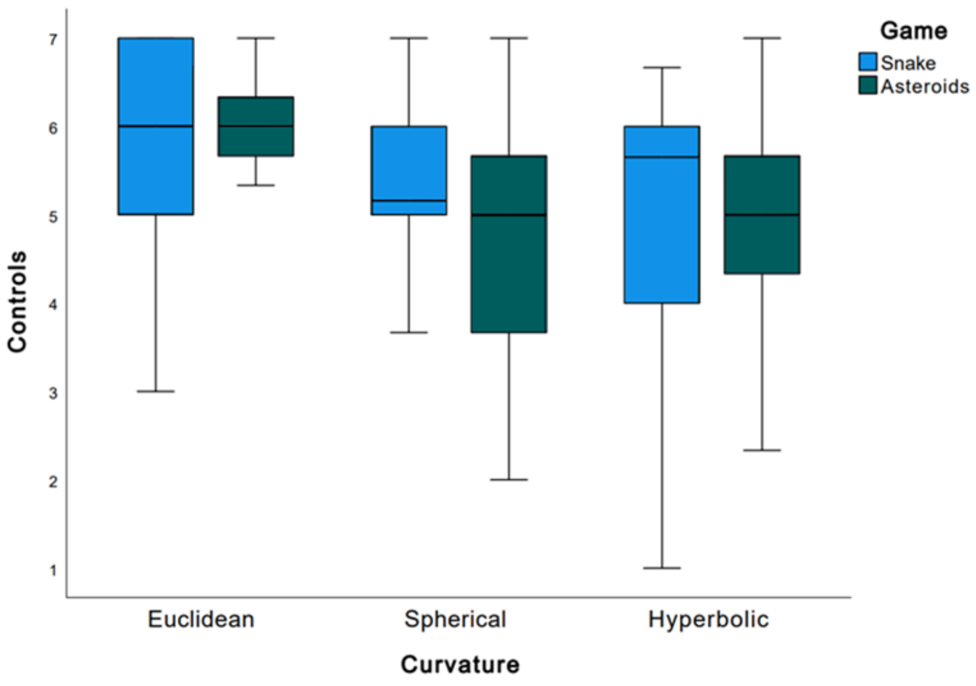


Figure 11.17: Box plot comparing data for the Controls dimension between Snake and Asteroids in Euclidean (left), spherical (centre) and hyperbolic (right) space.

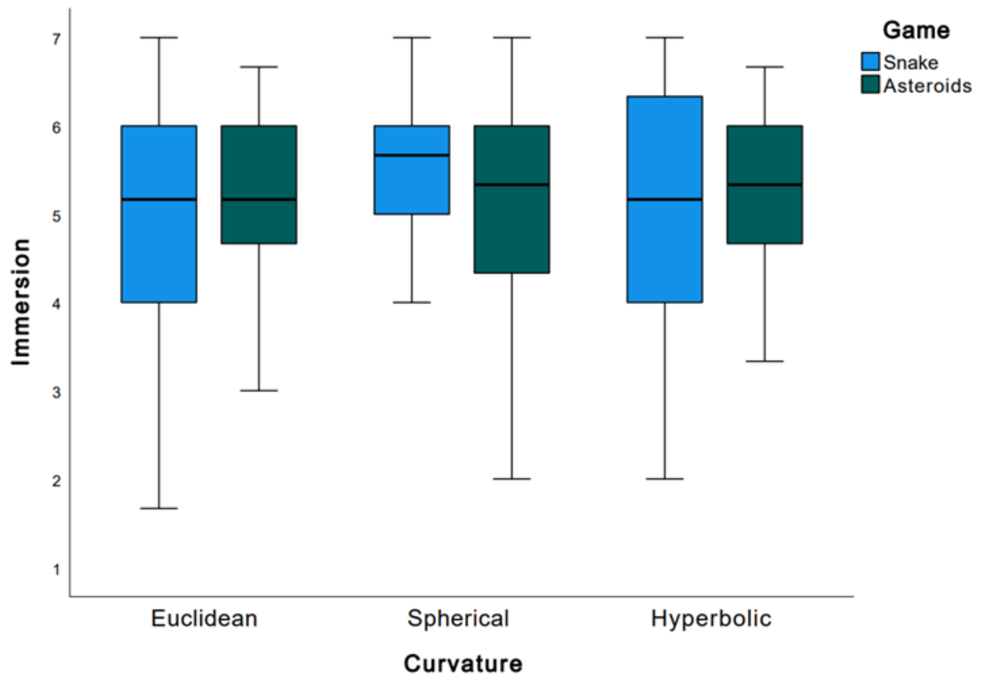


Figure 11.18: Box plot comparing data for the Immersion dimension between Snake and Asteroids in Euclidean (left), spherical (centre) and hyperbolic (right) space.

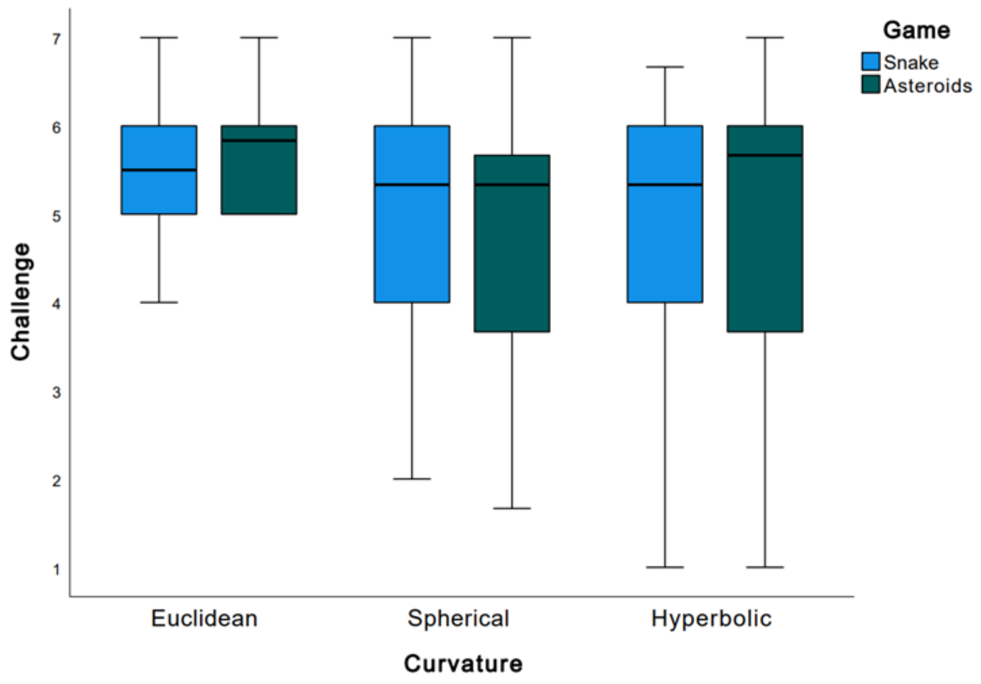


Figure 11.19: Box plot comparing data for the Challenge dimension between Snake and Asteroids in Euclidean (left), spherical (centre) and hyperbolic (right) space.

With regards to the preferences for specific curvatures, 12 participants said they preferred spherical space in Snake, 7 said they preferred hyperbolic space, and 3 did not have a preference. As for Asteroids, 9 players preferred spherical space, 5 preferred hyperbolic space and 8 had no preference. This difference was not statistically significant neither for Snake ($\chi^2(2) = 5.545; p = 0.06$) nor for Asteroids ($\chi^2(2) = 1.182; p = 0.56$).

11.5 Discussion

From the data gathered, participants felt like they could play Snake better than Asteroids, which might be due to the fact that on average participants were more familiar with Snake. Asteroids game is more complex and dynamic than Snake, which likely influenced why players felt less masterful playing non-Euclidean Asteroids. When playing Snake, the player just needs to keep track of one object moving and it is a step-based simulation. When playing Asteroids, the player has to keep track of many objects flying along the geodesics as well as to plan where the to move the spaceship and which of the asteroids to shoot at. It is surprising that participants thought they were less masterful on average in hyperbolic space rather than spherical space. As can be seen in Figure 11.2 and Figure 11.5, it is more difficult to avoid colliding with the asteroids in spherical space, because radially there is less space when one moves away from the origin compared to flat space. This makes the difficulty level for Asteroids higher in spherical space.

Additionally, the unfamiliarity of object movement through curved space creates the perception of a game being harder to master. This is supported by the responses to the free-text questions regarding user experience when first seeing the curved space. Multiple participants mentioned that they did not immediately understand how to control the player character in curved space. One participant wrote:

“[It was] difficult to predict how different curvatures would affect the snake’s movement”

The results show that participants felt spherical Snake was harder to master than Euclidean and hyperbolic versions of the game. This could be due to the novelty of the snake’s movement. In spherical space the snake follows the great circle geodesics, so ends up going in a circle if not controlled, hence it is easier to collide with the tail of the snake. On the other hand, in hyperbolic space, the snake will go off screen (and be reflected to the antipodal point) following the geodesics. This point of view is summarised well in an answer to the same free-text question:

“I noticed I was still trying to apply a Cartesian style of movement to reach the object. I began using the geodesics on the Spherical surface to reach the object instead of turning so much, but I think the Hyperbolic surface really shines here – it forces you to think hyperbolically since L/R turns onto different geodesics can have totally different effects, at least compared to what you expect in a Cartesian system."

This comment refers to the fact that when turning 90 degrees with a right or left turn, you switch the geodesic the snake is following, but this new geodesic will not continue orthogonally on the screen. The snake will also curve away from the origin due to exponentially expanding amount of space, so it is easier to play a much longer game of snake in hyperbolic space than in spherical space. This might have influenced the player's perception of mastery playing spherical Snake.

As for the game controls, the participants felt that the controls were significantly harder in curved space compared to Euclidean space. This difference is more pronounced in the Asteroids game than the Snake game. It could be due to controls and movement in the Snake game being more constrained than in Asteroids. Snake has step-based movement and turns in sharp 90-degree angles, thus is might be easier for participants to see get used to. On the other hand, the Spaceship in the Asteroids game has free range of movement, so the controls are more difficult. However, not all participants thought this, for example one response said:

"The spherical and hyperbolic were very similar and did not feel as different as Snake did, it felt much easier to control."

This participant has encountered Asteroids after having played Snake, so could compare the experience.

Overall, even though the ratings for functional dimensions (mastery and controls) have been lower for the non-Euclidean versions of both games, the non-functional dimensions did not show a significant difference between the curved and non-curved space games. This would suggest that the players did not feel the games to be negatively impacted by being set in curved space environment. This is supported by responses to summary part of the survey. One of the participant said the following:

"I loved watching the spacecraft fire follow geodesics, and using that as a tool to hit the asteroids. Further, loved watching the asteroids warp their shapes. I would like to be able to fly through the geodesics unfettered to understand those dynamics a bit more, but otherwise, the controls (acceleration especially) were really well rendered and made it fun to move, and that ease of motion helped me focus on hitting targets."

This response would suggest that the more natural movement, using free range of motion as well as gradual acceleration is better suited for making the curved space more intuitive than the step-based movement. For the question of whether curved space improved their experience, only two participants answered “no”, while other participants answered positively. One of the answers has said:

“Improved the experience of playing the game and made it more fun.”

While recreating existing games in 2D has been well received, another comment has suggested that the two chosen examples have been illustrative to help participants understand curved space, but they have not truly taken advantage of curved space in developing the game itself:

“Improved! It's nice to have multiple options here to balance challenge vs. comfort. I'd certainly enjoy playing the games in all curvatures, but I see these more as pedagogical tools for the moment. A truly immersive game that forced its concept on curved spaces would make adapting to the curved spaces more enjoyable – not that it's not enjoyable here, but the point of playing these two on those surfaces seems to force someone to understand how those surfaces work more so than shoot the asteroids or avoid a tail. I'm totally fine with that :) but I wonder if players would want more of a ‘goal’.”

Both Snake and Asteroids in curved space can help visualise the spherical and hyperbolic projections well and help understand the object movement in such environments. However, the properties of these environments are not taken into account when designing the challenges of the game. This comment ties into the next free-text question, which has asked participants if they would want to see other games adapted to curved space and which game genres would work best. There were suggestions of recreating other classic 2D games, like Breakout, Mario, Sonic, Pacman, Space Invaders, as well as more general racing games and ball games. Participants have also said:

“Puzzle games and platformers could work. The mechanic would have to be introduced slowly.”

“Absolutely – this is a vastly under-tapped dimension for gaming. I would imagine re-creating many 2D classics (space invaders, Pacman, etc) would be a lot of fun, but coming up something that truly exploits the surfaces for their own right would force new creative challenges.”

These comments agree that it would be best to use the properties of the curved space when designing the game. Perhaps what would be a better use of this mechanic is a puzzle game that can only function in spherical or hyperbolic space because of the unique properties of the respective environment. Hyperrogue, an exploration game in hyperbolic space (described in section 2.4.4), has some mechanics, which are only possible due to the game being set in hyperbolic geometry. For example, it is possible to escape from being cornered by enemies much easier than in Euclidean space. Additionally it is easier to run away from threats, as the distance increases exponentially if not exactly the same path is taken by the pursuer. These and other properties of hyperbolic and spherical space could be taken advantage of when designing innovative game mechanics.

One participant pointed out that for Asteroids it would be a good idea to add adjustable difficulty, which would suggest that the set difficulty has not been appropriate:

"The barebones rendering seemed fine – in a full game it would be nice to have difficulty options, but that's perhaps another discussion?"

11.6 Summary

This chapter has described the survey designed to research the interest in the features of the engine developed in this project. It covered the implementation of two classic games (i.e., Snake and Asteroids) in non-Euclidean environment and the challenges in adapting the game controls and collision detection to the curved space.

This chapter has also detailed the structure of the survey including the aims, the structure and the questionnaire chosen to measure different aspects of participants' interaction with the game.

It has summarised and analysed the data gathered by the survey. The findings show that snake generally had better ratings, which could have been due to the participants greater prior familiarity with this game. The controls and collision detection worked seamlessly, but the difficulty of the game could have been better standardised to not change as much between different curvatures of space. Overall, the participants responded positively to adapting games to curved space and would be interested in playing other games implemented in non-Euclidean environment or even novel games designed to be played in curved space.

12. Conclusions

This thesis presented a range of approaches to rendering non-Euclidean geometry in real-time 2D simulations. These approaches are varied and this chapter covers the extent to which the implemented engine achieves the original aims and objectives set out when the project started.

The motivation for this project was to create an intuitive and approachable method of simulating and rendering curved space; and to create software that can calculate and render arbitrary shapes in curved space in real-time, while remaining intuitive for users to be able to create physical worlds with non-Euclidean space. This was to be accomplished by a set of aims and objectives, and the achievements with regards to each of these will be discussed.

12.1 Evaluation of Aims & Objectives

The first objective was to *"Create a method for rendering shapes in non-Euclidean geometry: it should be intuitive for the users and developers."*. The method for calculating object's vertices and subsequently tessellating the object's vertices using spherical and hyperbolic trigonometry has been described in chapter 4. The approach uses azimuthal equidistant projection, which is intuitive for the users to understand the curvature of space, as the objects get stretched (in spherical space) and shrunk (in hyperbolic space) laterally the further they are from the centre of the projection; but the distance from the centre of projection does not change with different curvatures of space. Additionally, the ability to play the same game in different curvatures makes it more intuitive for players to understand how curvature affects the look of shapes and their movements within the game world. This is supported by the answers of survey participants in section 11.5. For example, one of the participants said:

"I noticed I was still trying to apply a Cartesian style of movement to reach the object. I began using the geodesics on the Spherical surface to reach the object instead of turning so much, but I think the Hyperbolic surface really shines here – it forces you to think hyperbolically since L/R turns onto different geodesics can have totally different effects, at least compared to what you expect in a Cartesian system."

The gridlines also help users visualise the geodesics through the curved space. On the other hand, the use of polar coordinate system makes it harder for developers to create the game world, because the object's parameters have to be set up in $(\boldsymbol{r}, \boldsymbol{\theta})$ form. This makes it difficult to accurately place objects in the game world.

For example, the following is the code that creates a simple quadrilateral asteroid shape:

```
	square_properties.shape = engine::shape::create(true,
		engine::polygon(30.f, 4).points(), 30.f);
	square_properties.position = engine::polar_vector2(200.f, -0.8f);
	square_properties.rotation = m_pi / 2.0f;
```

This has been mitigated by developing conversion methods so that the world is created in cartesian coordinates and then converted to polar coordinates and recorded within the engine. The following two methods convert a vector from polar to Cartesian form and vice versa:

```
	glm::vec3 to_cartesian(float r, float theta) {
		return glm::vec3(r * cos(theta), r * sin(theta), 0.0f);
	}

	static polar_vector2 to_polar(float x, float y) {
		return polar_vector2(sqrt(x*x + y*y), atan2f(y, x));
	}
```

Conversion methods are slow due to using expensive square root and trigonometry operations, so the application should use them rarely during runtime. However, when initialising the world their use is preferrable, as they allow for a more intuitive set up of the game world.

The second objective was to *"Implement the method to create a rendering engine: the software should be capable of recording objects' parameters, calculate object transformations and render them on screen."*. The structure of the implemented engine has been described in section 3.1. It has a high degree of decoupling of the code: game logic and engine code are separated. This allows for reusability of the engine and makes it easier to introduce changes to the code. The polar coordinate system is curvature agnostic, which makes it possible to record the object's parameters and use them regardless of the curvature value within the game world.

Non-Euclidean line rendering was the first feature implemented in the engine. The method for line rendering is described in chapter 4 and it uses spherical and hyperbolic trigonometry to render objects in curved space. Functions were written which take object's properties as parameters (position, rotation and shape recorded in local coordinate system) and return a transformed shape. The engine is capable of drawing objects in curved space as well as changing curvature dynamically when the simulation is running. For testing purposes this has been set to change to pre-set values when user presses the following keyboard inputs: key "I" set the curvature to $\mathbf{K} = -\mathbf{1}$, key "O" set the curvature to $K = 0$, key "P" set curvature to $K = 1$. Additionally, the curvature can change continuously, so when the user pressed the key "K" or "L", the curvature would gradually decrease to a minimum of $\mathbf{K} = -\mathbf{1}$ or increase to a maximum of $\mathrm{K} = 1$ respectively. The rate of change of curvature has been set to be $\pm\mathbf{0.25K}/\boldsymbol{sec}$ and has been adjusted to be frame rate independent.

Alternative approaches for line rendering described in chapter 6 have been implemented in the engine. Great Circle Path (described in section 6.1) and Orthogonal Vectors (described in section 6.2) approaches have been implemented as an alternative to spherical trigonometry approach; while Poincare Disc Line Equation approach (described in section 6.3) has been implemented as an alternative to the hyperbolic trigonometry approach. Line rendering capability has been modified to work on the GPU using OpenGL Shading Language (GLSL) as described in chapter 7. This is implemented directly in the rendering pipeline when an object is submitted for rendering within the scene.

Additionally, texture rendering capability has been implemented within the engine following the method described in chapter 9. It uses spherical and hyperbolic trigonometry and is implemented in the shader, so works within the rendering pipeline. It is possible to specify whether each individual object is using line or texture rendering, which makes the engine more flexible. An example of this can be seen in Figure 9.8, Figure 9.9 and Figure 9.10, where the gridlines and spaceship's lasers are drawn using line rendering, while spaceship and asteroids are drawn using texture rendering.

The third objective was to *"Expand the engine for object movement: record additional parameters for the objects and simulate the movement of the objects along the geodesics of planar, spherical or hyperbolic space."*. The method for calculating the movement of the object movement through curved space has been described in chapter 5. The engine calculates the coordinates of the position of the object in the next frame based on its position, velocity and trajectory in the previous frame. The method uses hyperbolic and spherical trigonometry to translate an object along its trajectory geodesic keeping the orientation with respect to the geodesic constant. This is illustrated in Figure 5.2, Figure 5.3, Figure 5.4 and Figure 5.5. Additionally, the engine is capable of performing proximity-based collision detection in curved space, which allows for object interaction. This is used in the implementation of Asteroid and Snake games described in sections 11.1 and 11.2 respectively.

The engine has the capability to calculate object movement through curved space using the method described in chapter 5. This is implemented as part of the `update()` method in the `game_object` class. When called, it finds the position and rotation of the object in the next frame based on the properties of the object in the previous frame.

The fourth objective was to *"Optimize performance of the engine: the software should be able to render complex scenes in real-time to be a viable game engine."*. Multiple approaches to improve performance of the described software have been covered in chapters 6 and 7. First, there were three approaches implemented designed to improve the performance of the shape tessellation in curved space. These were alternatives to spherical and hyperbolic trigonometry approaches and included two methods for spherical space (Great Circle Path approach covered in section 6.1 and Orthogonal Vectors approach covered in section 6.2) and one approach for hyperbolic space (Poincare Disc Line Equation approach covered in section 6.3).

These as well as trigonometry approaches were adapted to work on a GPU, which has parallelised the computations. Chapter 8 described the theoretical and empirical analysis of the GPU and non-GPU approaches. While the parallelisation has greatly improved the performance of the engine and increased the frames per second the engine can render by an order of magnitude, the alternative tessellation approaches provided negligible benefit in comparison. Spherical and hyperbolic trigonometry subsequently were chosen to develop the texture rendering approach due to the similarity between the two approaches. The line equation approaches could have provided an approximately 2-4% improvement in performance, but two different approaches would need to be developed, which would make the engine less adaptable for potential future modifications.

The fifth objective was to *"Expand the engine to render textured shapes: create a method for texturing shapes in curved space and implement it within the engine."*. The method for rendering textures in curved space has been described in chapter 9. The engine is capable of drawing shapes in curved space using both line and texture rendering. The shape is subdivided into segments (one for each edge), which are then traced with lines parallel (number of which depends on inner tessellation coordinate) to the edge of that segment, forming multiple smaller similar triangles with a common vertex at the local coordinate system origin. The texture coordinates are found throughout the segment along these lines (the number of these intermediate points depends on the tessellation coordinates). As a proof of concept, an adaptation of the Asteroids game has been implemented in the engine using the texture rendering capabilities. Section 11.1 covers the implementation of the Asteroids game in non-Euclidean geometry, while section 0 covers the changes made to use textures for rendering the Asteroids game. Overall, the texture rendering works well and provides better visualisation of the curvature for the users, this is shown in Figure 9.12, Figure 9.13 and Figure 9.14.

The sixth objective was to *"Evaluate people's opinions towards non-Euclidean elements in games: use the engine to adapt well-known games to work in non-Euclidean space. Run a survey to learn whether participants like or dislike the idea."*. The survey designed to measure player's enthusiasm towards games with the curved space mechanic as well as the intuitiveness of the games using it has been presented in chapter 11. Two classic games (Asteroids and Snake) have been adapted to work in curved space. Subsequently a Qualtrics survey has been developed to measure different aspects of participants' interaction with the game. This survey has measured how easy to understand the participants found the curved space feature as well as the comparative enjoyment of the different versions of these games (Euclidean, spherical and hyperbolic). The results show that on average participants preferred non-Euclidean Asteroids to non-Euclidean Snake, which could have been due to Snake being better without texture rendering. The controls and collision detection worked well, however the difficulty was too varied between different curvatures of space. In general people responded well to classic games being adapted to work in non-Euclidean space. Further statistical analysis should be performed on the data gathered to determine whether the results were statistically significant.

12.2 Related Work and Alternative Approaches

There are different approaches to making games and simulations in non-Euclidean geometry. The closest approach to the methods described in this thesis is the work by Guimarães, et al. (Guimarães, Mello, & Velho, 2015). Both studies are researching a way to use non-Euclidean geometry in 2D game development.

Guimarães, et al. have adapted the classic Asteroids game to work in a non-Euclidean environment, which is similar to the non-Euclidean Asteroids game built using the engine described in section 11.1. However, the approaches and results are different.

Figure 12.1 shows a comparison between the renderings created by the engine described in this thesis (top row) and the software created by Guimarães, et al. (bottom row). Instead of using the trigonometric calculations and polar coordinate system for rendering and transforming shapes, the authors use spherical and hyperbolic metrics and isometry equations in a generalised $(\mathbf{x}, \mathbf{y})$ coordinate system. They have also described a Lagrangian equation for calculating motion through curved space based on current and consecutive object positions.

Guimarães, et al. specified a screen wrap around function works in a similar way to the implementation described in section 11.1, but because they use the $(\mathbf{x}, \mathbf{y})$ coordinate system, the wrapping was represented differently for Euclidean, spherical and hyperbolic space. For

Euclidean it was kept the same as in the original Asteroids game; in spherical space the wrapping is a consequence of the great circle geodesics wrapping around the projection naturally; and for hyperbolic space, the boundary of the playable area was chosen to be represented as a connected sum of two tori, which created the boundary shape seen in Figure 12.1 (bottom right). The engine described in this thesis can manage the wrapping independent of the curvature by moving the object to the antipodal point of its θ-coordinate as well as adjusting the velocity and rotation accordingly.

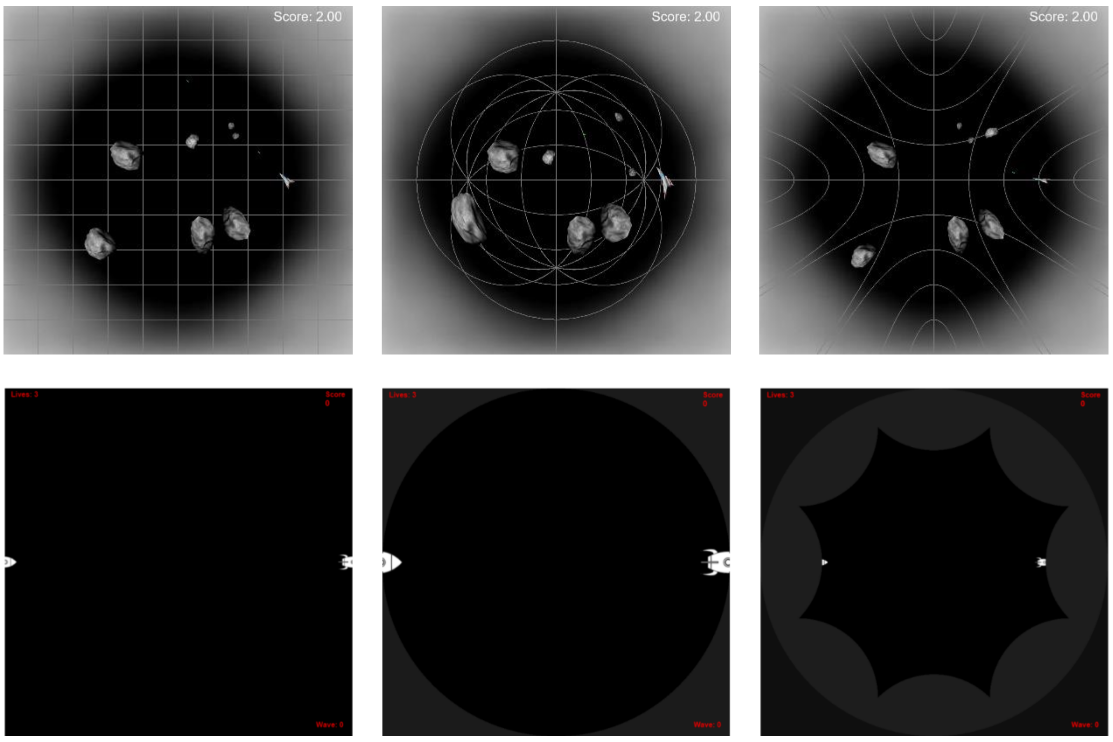


Figure 12.1: Comparison of adaptations of the Asteroids game rendered by this engine (top row) and Guimarães, et al. (bottom row) in Euclidean (left column), spherical (centre column) and hyperbolic (right column) space.

David Madore has created two variants of the Hyperbolic Maze game (Madore, Hyperbolic maze, 2013) (Madore, Hyperbolic Maze 2, 2013). Like the work described in this thesis, the game by D. Madore uses the conformal model. But, in his work the computations use matrix multiplication using Möbius transformations to preserving the unit circle (i.e., hyperbolic transformations under the Poincaré disk model). Another difference between the approaches is the coordinate system representation. D. Madore has said:

“Main difficulty in representing hyperbolic space in a computer is that any real/continuous coordinate system (whether polar coordinates, Poincaré projection, Beltrami-Klein projection…) soon becomes unusable when you move away from its centre.”

This is true, due to the nature of negatively curved space, the further you move away from the centre of the projection, the faster the space is expanding in area. This can be solved in different ways: restricting the accessible area within the game world or split the coordinates into discrete sections. The former approach has been used when implementing non-Euclidean Asteroids and Snake games described in sections 11.1 and 11.2; this approach was natural due to original Asteroids and Snake games using wrapping world feature (i.e., when player moves off the screen, he reappears on the directly opposite side of the screen). D. Madore chose to split the world into discrete parts (quadrilateral tiles in hyperbolic tiling), which were then numerically encoded and handled exactly. The finite set of tiles is then wrapped around and repeated indefinitely if the player keeps moving in one direction through the maze.

Another difference is that Hyperbolic Maze game, as the name suggests, only works in hyperbolic space, while the engine described by this thesis is capable of working in any arbitrary curvature of space in the range of $\mathbf{-1 \leq K \leq 1}$, where $\mathbf{K}$ is the Gaussian curvature.

Jeff Weeks has created software in non-Euclidean space. The most similar to the work done in this project is KaleidoTile, application which allows users to explore different polyhedral tilings of Euclidean, spherical and hyperbolic space (Weeks, KaleidoTile, 2020). All of the calculations use the projective model of non-Euclidean geometry which allows for the adaptation of traditional transformation matrices (i.e., translation, rotation and reflection) for curved space (Weeks, Real-time rendering in curved spaces, 2002). J. Weeks has also explored the adaptation of 3D games to non-Euclidean space using the projective model (Weeks, Non-euclidean billiards in vr, 2020).

There are further features that could be added to the engine. Firstly, a more precise collision detection system could be developed. An adaptation of the axis aligned or even object aligned bounding box collision detection would be possible by calculating these as geodesics which bound the object rather than cartesian straight lines. This would allow for more precise interaction between the objects, due to additional techniques like subdivision of space being analogous in non-Euclidean geometry.

Another feature that would improve the capabilities of the engine is non-Euclidean pathfinding. Pathfinding algorithms are widely used in games, especially for non-playable characters, but also for user-controlled characters in genres such as Real-Time Strategy games or Multiplayer Online Battle Arena games. The pathfinding could work by subdividing the space in a game world into a variable number of cells (dependent on curvature)

Participants of the survey described in chapter 11 have suggested that a novel game which takes advantage of the feature of non-Euclidean space would be enjoyable and immersive for players, because the challenge would come from understanding the curved space and object interactions within it. Thus, the one of the potential future developments is making a novel game with a focus of the gameplay being around the curved space mechanic. The best genre for such a game would be a 2D puzzle game. Once the game is complete, another survey to measure people's enthusiasm towards such a game could be set up. This survey could compare the enjoyment to the non-Euclidean adaptations of Asteroids/Snake or be self-contained and thus shorter survey, which could make the completion rate higher.

Another potential feature that could be added to the engine is non-uniform curvature. Local areas of different curvature could be added to a uniformly curved space. These would be strongest at the centre of the local curvature area and would gradually be interpolated towards the global curvature away from the area's centre. Variable curvature is illustrated in Figure 12.2.

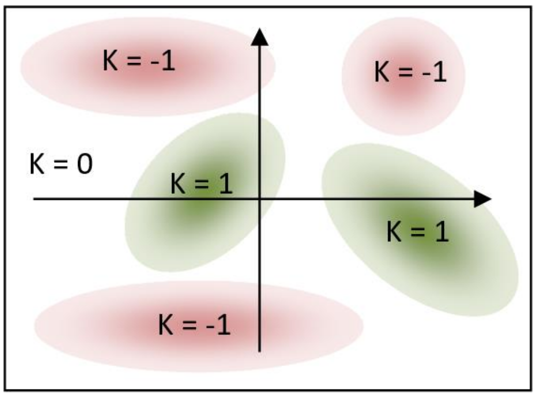


Figure 12.2: Game world with global Euclidean curvature $(\boldsymbol{K} = \mathbf{0})$ *and local areas with different curvature. Two areas with positive curvature* $(\boldsymbol{K} = \mathbf{1})$ *marked green and three areas with negative curvature* $(\boldsymbol{K} = -\mathbf{1})$ *marked red.*

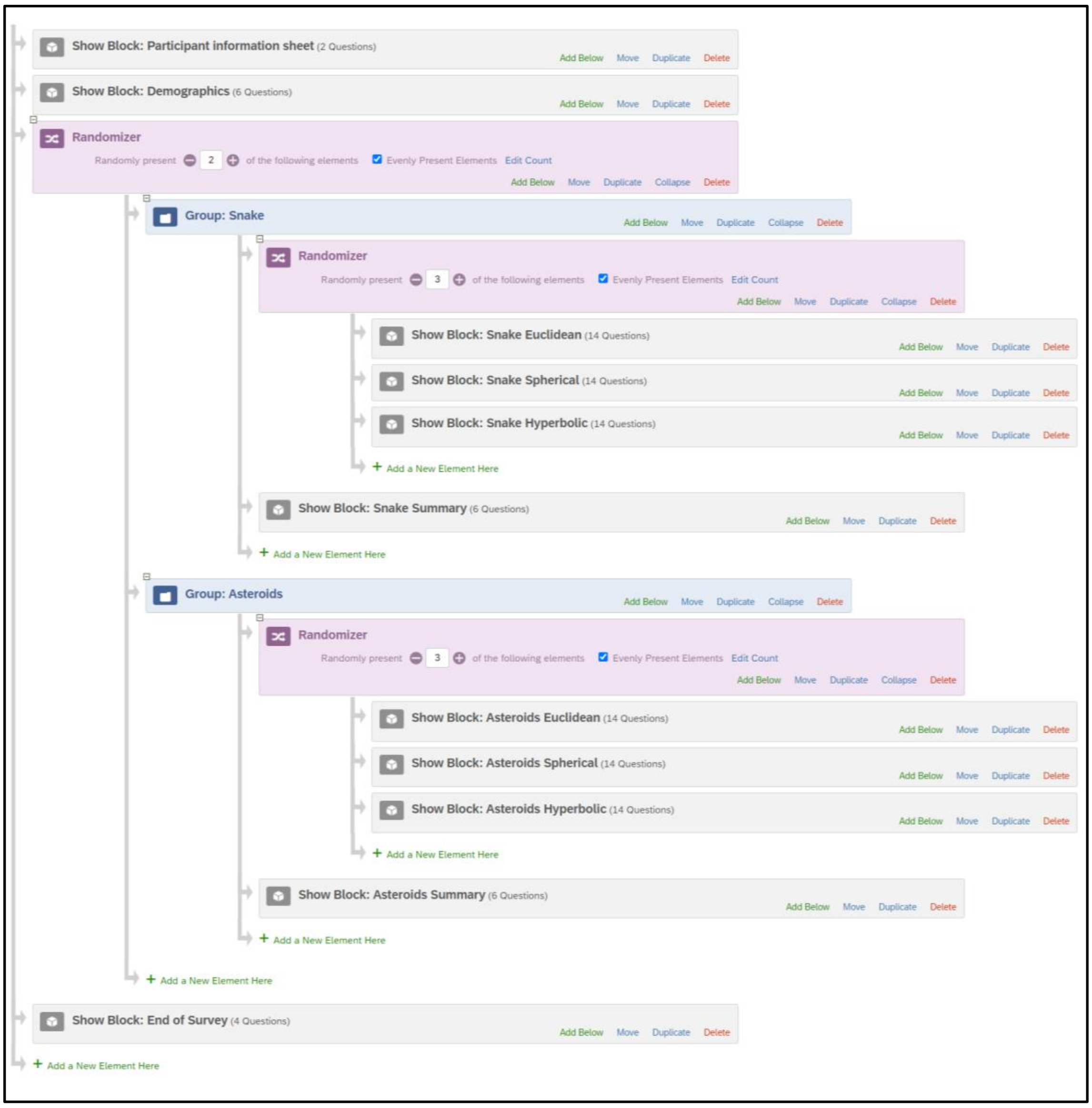
Finally, engine could be adapted to work in 3D non-Euclidean space. This would require a rework of the methods in chapter 4 to render lines in 3D space, followed by a rework to methods in chapter 5 for movement in 3D space and chapter 9 to render textures in 3D curved space. When implemented, it would be an alternative to using the projective model for rendering 3D objects in curved space. Empirical testing could be performed to measure the comparative performance of these approaches.

12.3 Summary

This chapter discussed the results of this project in the context of its initial aims and objectives, related work and future work. Overall, the objectives have been largely fulfilled and the engine created can be used to make games and simulations in non-Euclidean environments. Some issues were described and subsequently addressed in the further work section, which described additional features that could be implemented in the engine as well as potential research into player interaction with games in non-Euclidean space. The research has been compared to other related work in this area in terms of rendering of games in 2D non-Euclidean space.

Appendix A

Survey Flow diagram screenshot taken from the created Qualtrics survey.



- **Show Block: Participant information sheet** (2 Questions) — Add Below · Move · Duplicate · Delete
- **Show Block: Demographics** (6 Questions) — Add Below · Move · Duplicate · Delete
- **Randomizer** — Randomly present 2 of the following elements ☑ Evenly Present Elements · Edit Count — Add Below · Move · Duplicate · Collapse · Delete
  - **Group: Snake** — Add Below · Move · Duplicate · Collapse · Delete
    - **Randomizer** — Randomly present 3 of the following elements ☑ Evenly Present Elements · Edit Count — Add Below · Move · Duplicate · Collapse · Delete
      - **Show Block: Snake Euclidean** (14 Questions) — Add Below · Move · Duplicate · Delete
      - **Show Block: Snake Spherical** (14 Questions) — Add Below · Move · Duplicate · Delete
      - **Show Block: Snake Hyperbolic** (14 Questions) — Add Below · Move · Duplicate · Delete
      - + Add a New Element Here
    - **Show Block: Snake Summary** (6 Questions) — Add Below · Move · Duplicate · Delete
    - + Add a New Element Here
  - **Group: Asteroids** — Add Below · Move · Duplicate · Collapse · Delete
    - **Randomizer** — Randomly present 3 of the following elements ☑ Evenly Present Elements · Edit Count — Add Below · Move · Duplicate · Collapse · Delete
      - **Show Block: Asteroids Euclidean** (14 Questions) — Add Below · Move · Duplicate · Delete
      - **Show Block: Asteroids Spherical** (14 Questions) — Add Below · Move · Duplicate · Delete
      - **Show Block: Asteroids Hyperbolic** (14 Questions) — Add Below · Move · Duplicate · Delete
      - + Add a New Element Here
    - **Show Block: Asteroids Summary** (6 Questions) — Add Below · Move · Duplicate · Delete
    - + Add a New Element Here
  - + Add a New Element Here
- **Show Block: End of Survey** (4 Questions) — Add Below · Move · Duplicate · Delete
- + Add a New Element Here

13. References

Abeele, V. V., Spiel, K., Nacke, L., Johnson, D., & Gerling, K. (2020). Development and Validation of the Player Experience Inventory: A Scale to Measure Player Experiences at the Level of Functional and Psychosocial Consequences. *International Journal of Human-Computer Studies, 135*, 102370.

Amenta, N., Levy, S., Munzer, T., & Phillips, M. (1995). Geomview: A system for geometric visualization. *In Proceedings of the eleventh annual symposium on Computational geometry*, 412-413.

Anderson, J. W. (2006). *Hyperbolic Geometry.* Springer Science & Business Media.

Artmann, B. (2018, May 16). *Projective geometry.* Retrieved September 27, 2022, from Encyclopedia Britannica: https://www.britannica.com/science/projective-geometry

Azadvar, A., & Canossa, A. (2018). Upeq: ubisoft perceived experience questionnaire: a self-determination evaluation tool for video games. *Proceedings of the 13th international conference on the foundations of digital games*, 1-7.

Bachmann, P. (1894). *Die analytische zahlentheorie, vol. 2.* Leipzig: Teubner.

Beltrami, E. (1868a). Saggio di interpretazione della geometria non-euclidea. *Giornale di Mathematiche, VI*, 285–315.

Beltrami, E. (1868b). Teoria fondamentale degli spazii di curvatura costante. *Annali di Matematica Pura ed Applicata, series II*, 232–255.

Bonola, R. (1912). Non-Euclidean geometry; a critical and historical study of its development. *Bulletin of the American Mathematical Society, 19(1)*, 22-23.

Brockwell, H. (2016, April 3). *Forgotten genius: the man who made a working VR machine in 1957*. Retrieved November 23, 2021, from Tech Radar: https://www.techradar.com/news/wearables/forgotten-genius-the-man-who-made-a-working-vr-machine-in-1957-1318253/2

Carslaw, H. S. (1916). The Elements of Non-Euclidean Plane Geometry and Trigonometry. *The Mathematical Gazette, 9(141)*, 374-374.

Cayley, A. (1859). A sixth memoir upon quantics. *Philosophical Transactions of the Royal Society of London, 149*, 61–90.

Chernikov, Y. (2012). *The Cherno.* Retrieved from Youtube: https://www.youtube.com/c/TheChernoProject/videos

Cotter, H. (1976). The Early History of Great Circle Sailing. *Journal of Navigation, 29(3)*, 254-262.

Coxeter, H. S. (1979). The non-Euclidean symmetry of Escher's picture 'Circle Limit III'. *Leonardo, 12(1)*, 19–25.

Coxeter, H. S. (2003). *Projective geometry.* Springer Science & Business Media.

Eder, M. (2000). Views of Euclid's Parallel Postulate in Ancient Greece and in Medieval Islam. *History of mathematics*.

Einstein, A. (1921). *Relativity: The special and general theory.* New York: Holt.

Escher, M. (1959). *Circle Limit III.*

Francis, G. K., Goudeseune, C. M., Kaczmarski, H. J., Schaeffer, B. J., & Sullivan, J. M. (2003). ALICE on the eightfold way: exploring curved spaces in an enclosed virtual reality theater. *Visualization and mathematics III*, 305-315.

Furuti, C. A. (2012). *Visualizations of the Azimuthal Orthographic Geometry.* Retrieved August 2, 2018, from Progonos: http://www.progonos.com/furuti/MapProj/Dither/CartHow/HowOrtho/Img/im_orthoRays.png

Galesic, M., & Bosnjak, M. (2009). Effects of questionnaire length on participation and indicators of response quality in a web survey. *Public opinion quarterly, 73(2)*, 349-360.

Gauss, C. F. (1824, November 8). *letter to Franz Adolph Taurinus.* Retrieved from Friedrich Gauss Letters: https://gauss.adw-goe.de/handle/gauss/1688

Gellert, W. (1989). Great circles, small circles and lunes. In S. Gottwald, M. Hellwich, H. Kästner, & H. Küstner, *The VNR Concise Encyclopedia of Mathematics, 2nd ed* (pp. 261–282). New York: Van Nostrand Reinhold.

Gessler, A., Schulze, T., & Kulling, K. (2006). *Assimp*. Retrieved from GitHub: https://github.com/assimp/assimp

Gray, J. (1979). Non-euclidean geometry – A re-interpretation. *Historia Mathematica, 6(3)*, 236–258.

Gray, J. (2004). *János Bolyai, non-Euclidean geometry, and the nature of space.* Cambridge, Mass.: Burndy Library MIT Press.

Gray, J. (2006). Gauss and non-Euclidean geometry. In *Non-Euclidean Geometries* (pp. 61-80). Boston, MA: Springer.

Greenberg, M. J. (2008). *Euclidean and Non-Euclidean Geometries, 4th ed.* New York: W.H. Freeman.

Guimarães, F. D., Mello, V. M., & Velho, L. (2015). Geometry independent game encapsulation for non-euclidean geometries. *In Proceedings of SIBGRAPI Workshop of Works in Progress, vol. 1.*

Gunn, C. (1993). Discrete groups and visualization of three-dimensional manifolds. *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, 255-262.

Gunn, C. (2010). Advances in metric-neutral visualization. *Computer Graphics, Computer Vision and Mathematics 2010 proceedings*, 17-26.

Gunn, C., Epstein, D. B., & Maxwell, D. (1991). *"Not Knot" [videotape].* Jones and Bartlett Pub.

Haider, A., Gerling, K., & Vanden Abeele, V. (2020). The Player Experience Inventory Bench: Providing Games User Researchers Actionable Insight into Player Experiences. *Extended Abstracts of the 2020 Annual Symposium on Computer-Human Interaction in Play*, 248-252.

Hairetdinova, N. G. (1986). On spherical trigonometry in the medieval Near East and in Europe. *Historia mathematica, 13(2)*, 136-146.

Halsted, G. B. (1900). Gauss and the non-Euclidean geometry. *The American Mathematical Monthly, 7(11)*, 247-252.

Heath, T. L. (1956). *Euclid's Elements (translated).* Dover.

Hedegaard, R. (2004, April 14). *Orientable Manifold.* Retrieved September 26, 2022, from MathWorld--A Wolfram Web Resource: https://mathworld.wolfram.com/OrientableManifold.html

Hehl, F. W., & Kerlick, G. D. (1978). Metric-affine variational principles in general relativity. I. Riemannian space-time. *General Relativity and Gravitation, 9(8)*, 691-710.

Hoboken, G. (1994, August 29). *Hyperbolic Geometry*. Retrieved February 11, 2018, from Non-Euclidean Geometry: https://noneuclidean.tripod.com/hyperbolic.html

Hudson, R., Gunn, C., Francis, G. K., Sandin, D. J., & DeFanti, T. A. (1995). Mathenautics: using vr to visit 3-d manifolds. *In Proceedings of the 1995 symposium on Interactive 3D graphics*, 167-170.

IBM Corp. (2020). IBM SPSS Statistics for Windows, version 27.0. Armonk, NY.

Ijsselsteijn, W. A., de Kort, Y. A., & Poels, K. (2013). *The Game Experience Questionnaire.* Eindhoven: Technische Universiteit Eindhoven.

Jennett, C., Cox, A. L., Cairns, P., Dhoparee, S., Epps, A., Tijs, T., & Walton, A. (2008). Measuring and defining the experience of immersion in games. *International journal of human-computer studies, 66(9)*, 641-661.

Kessenich, J., Baldwin, D., & Rost, R. (2017). *GLSL 4.5 specification.* Retrieved July 20, 2020, from OpenGL: https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.50.pdf

Khronos Group. (1997). *Glad*. Retrieved from GitHub: https://github.com/Dav1dde/glad

Khronos group. (2010, March 11). *OpenGL 4.0.* Retrieved September 27, 2022, from Khronos.org: https://registry.khronos.org/OpenGL/specs/gl/glspec40.core.pdf

Klein, F. (1871). Ueber die sogenannte Nicht-Euklidische Geometrie. *Mathematische Annalen, 4(4)*, 573–625.

Korn, G. A., & Korn, T. M. (2000). Appendix B: B9. Plane and Spherical Trigonometry: Formulas Expressed in Terms of the Haversine Function. In *Mathematical handbook for scientists and engineers: Definitions, theorems, and formulas for reference and review, 3rd ed* (pp. 892-893). Mineola, New York: Dover Publications.

Kragh, H. (2012). Is space Flat? Nineteenth century astronomy and non-Euclidean geometry. *Journal of Astronomical History and Heritage, 15(3)*, 149-158.

Kreyszig, E. (1991). In *Differential Geometry* (p. 131). New York: Dover.

Kriz, J. (2010). *Great Circle on Spherical Earth.* Retrieved May 13, 2020, from Nosco: https://www.nosco.ch/mathematics/en/great-circle.php

Lamping, J., & Rao, R. (1996). The hyperbolic browser: A focus+ context technique for visualizing large hierarchies. *Journal of Visual Languages & Computing, 7(1)*, 33-55.

László, S.-K., & Magdics, M. (2021). Adapting Game Engines to Curved Spaces. *The Visual Computer*, 1-13.

Law, E. L., Brühlmann, F., & Mekler, E. D. (2018). Systematic review and validation of the game experience questionnaire (geq)-implications for citation and reporting practice. *In Proceedings of the 2018 Annual Symposium on Computer-Human Interaction in Play*, 257-270.

Likert, R. (1932). A technique for the measurement of attitudes. *Archives of Psychology, 140*, 1-55.

Madore, D. (2013). *Hyperbolic maze*. Retrieved November 25, 2017, from Madore.org: http://www.madore.org/~david/math/hyperbolic-maze.html

Madore, D. (2013). *Hyperbolic Maze 2*. Retrieved from madore.org: http://www.madore.org/~david/math/hyperbolic-maze-2.html

Madore, D. (2013, November 19). *Visualizing the sphere and the hyperbolic plane: five projections of each*. Retrieved Novemeber 25, 2017, from Youtube: https://www.youtube.com/watch?v=xHvAqDuWG2M

Marzec, C. (2010, October). *When Art Meets Science: The Hyberbolic Crochet Coral Reef.* Retrieved October 26, 2021, from Smithsonian Ocean: https://ocean.si.edu/ocean-life/invertebrates/when-art-meets-science-hyberbolic-crochet-coral-reef

Merriam-. (n.d.).

Merriam-Webster.com Dictionary. (2009, April 24). *Azimuthal equidistant projection.* Retrieved September 26, 2022, from Merriam-Webster: https://www.merriam-webster.com/dictionary/azimuthal%20equidistant%20projection

Merriam-Webster.com Dictionary. (2011, November 14). *gnomonic projection.* Retrieved September 26, 2022, from Merriam-Webster.com: https://www.merriam-webster.com/dictionary/gnomonic%20projection

Merriam-Webster.com Dictionary. (2015, September 21). *stereographic projection.* Retrieved September 26, 2022, from Merriam-Webster.com: https://www.merriam-webster.com/dictionary/stereographic%20projection

Mervis, J. (2002). The Geometry Center, 1991-1998. RIP. *Science, 297*, 508.

Munzner, T. (1998). Exploring large graphs in 3D hyperbolic space. *IEEE computer graphics and applications, 18(4)*, 18-23.

Novikov, P. (2001). *Axiomatic method.* Retrieved from Encyclopedia of Mathematics: URL: http://encyclopediaofmath.org/index.php?title=Axiomatic_method&oldid=45531

Osudin, D. (2022, November 8). *GitHub.com.* Retrieved from nonEuclidean_engine: https://github.com/danosudin1/nonEuclidean_engine

Osudin, D. (2022, November 8). *GitHub.com.* Retrieved from survey_curved_games_exe: https://github.com/danosudin1/survey_curved_games_exe

Osudin, D., Child, C., & He, Y.-H. (2019). Rendering non-euclidean space in real-time using spherical and hyperbolic trigonometry. *International Conference on Computational Science, Springer*, 543-550.

Oxford University. (2021). *OED Online.* Oxford: Oxford University Press.

Papadopoulos, A. (2014). On the works of Euler and his followers on spherical geometry, Gan. ita Bharatı (Indian Mathematics). *the Bulletin of the Indian Society for History of Mathematics, 36*, 1-2.

Petrunin, A. (2019). *Euclidean Plane and Its Relatives: A Minimalist Introduction.* Independently Published.

Phaelax. (2014, February 1). *Asteroids.* Retrieved October 1, 2021, from OpenGameArt: https://opengameart.org/content/asteroids

Phillips, M., & Gunn, C. (1992). Visualizing hyperbolic space: Unusual uses of 4x4 matrices. *In Proceedings of the 1992 symposium on Interactive 3D graphics*, 209-214.

Poincaré, H. (1881). Sur les applications de la géométrie non-euclidienne à la théorie des formes quadratiques. *Association Française Pour l'Avancement des Sciences, 10*, 132–138.

Qualtrics. (2021). Qualtrics. Provo, Utah, USA. Retrieved from Qualtrics.

Riccio, C. (2005). *GLM*. Retrieved from GitHub: https://github.com/g-truc/glm

Richards, M. (2015). *Software architecture patterns, vol. 4.* Sebastopol: O'Reilly Media, Incorporated.

Rodin, A. (2015). Did Lobachevsky Have A Model Of His "imaginary Geometry"? *Philosophy of Science, 3*, 34-63.

Rosenfeld, B. A. (2012). *A history of non-Euclidean geometry: Evolution of the concept of a geometric space.* Springer Science & Business Media.

Rowland, T. (2000, June 9). *Manifold.* Retrieved from MathWorld--A Wolfram Web Resource: https://mathworld.wolfram.com/Manifold.html

Scratchapixel. (2015, May 14). *Ray Tracing: Rendering a Triangle.* Retrieved from Scratchapixel: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/barycentric-coordinates

Taimina, D. (2018). *Crocheting Adventures with Hyperbolic Planes, 2nd ed.* Boca Raton, FL: CRC Press.

Tamfang. (2011). *Equilateral triangles.* Retrieved August 3, 2018, from https://pointatinfinityblog.files.wordpress.com/2018/02/triangle5.png?w=480\&h=480

The American Heritage Science Dictionary. (2016, June 1). *dictionary.com.* Retrieved September 25, 2022, from azimuthal-projection: https://www.dictionary.com/browse/azimuthal-projection

The GLFW Development Team. (2002). Retrieved from GLFW: https://www.glfw.org/

The Khronos Group. (2020, October 11). *Tessellation.* Retrieved from OpenGL: https://www.khronos.org/opengl/wiki/Tessellation

Thompson, J. (2015). *jeffreythompson.org.* Retrieved from Circle/Circle Collision Detection: http://www.jeffreythompson.org/collision-detection/circle-circle.php

Todhunter, I. (1863). *Spherical trigonometry, for the use of colleges and schools: with numerous examples.* Macmillan.

Traver, T. (2014). Trigonometry in the Hyperbolic Plane. *Manuscript, May*.

TrustRadius. (2014). *Qualtrics.* Retrieved from TrustRadius: https://www.trustradius.com/products/qualtrics/reviews

Turner, D., Wilhelm, R., & Lemberg, W. (1996). *FreeType* . Retrieved from GitLab: https://gitlab.freedesktop.org/freetype

United Nations. (2021, April 4). *United Nations Emblem and Flag.* Retrieved September 26, 2022, from un.org: https://www.un.org/en/about-us/un-emblem-and-flag

Wald, R. M. (2010). *General relativity.* University of Chicago Press.

Weeks, J. (2002). Real-time rendering in curved spaces. *IEEE Computer Graphics and Applications, 22*, 90-99.

Weeks, J. (2020, September 7). *KaleidoTile*. Retrieved December 15, 2021, from Geometry Games: https://www.geometrygames.org/KaleidoTile/index.html

Weeks, J. (2020). Non-euclidean billiards in vr. *Bridges 2020 Conference Proceedings*, 1-8.

Weintrit, A., & Neumann, T. (2011). *Methods and Algorithms in Navigation: Marine Navigation and Safety of Sea Transportation.* Boca Raton, FL: CRC Press.

Weißmann, S., Gunn, C., Brinkmann, P., Hoffmann, T., & Pinkall, U. (2009). jReality: a java library for real-time interactive 3D graphics and audio. *In Proceedings of the 17th ACM international conference on Multimedia*, 927-928.

Weisstein, E. W. (2000, May 17). *Proclus' Axiom – MathWorld.* Retrieved from Wolfram MathWorld: https://mathworld.wolfram.com/ProclusAxiom.html

Weisstein, E. W. (2002, January 8). *Conformal Projection.* Retrieved September 26, 2022, from MathWorld--A Wolfram Web Resource: https://mathworld.wolfram.com/ConformalProjection.html

Wheatstone, C. (1838). XVIII. Contributions to the physiology of vision.—Part the first. On some remarkable, and hitherto unobserved, phenomena of binocular vision. *Philosophical transactions of the Royal Society of London, 128*, 371-394.

Wood, L. (2012). *SaVi: satellite constellation visualization.* Retrieved from arXiv: https://arxiv.org/abs/1204.3265

Xevin. (2014, June 18). *simple spaceship.* Retrieved October 1, 2021, from OpenGameArt: https://opengameart.org/content/simple-spaceship

Zeno Rogue. (2022, 09 04). *Hyperrogue Gallery.* Retrieved from roguetemple.com: https://www.roguetemple.com/z/hyper/ms-escher.png