UNIVERSIDADE DA BEIRA INTERIOR
Engenharia Informática

Ray Tracing in Non-Euclidean Spaces

João Rodrigo de André e Alves Silva

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática
(2^o ciclo de estudos)

Supervisor: Prof. Dr. Abel João Padrão Gomes

Covilhã, Outubro 2018

Acknowledgements

To my advisor, Professor Abel Gomes, who was enthusiastic with the ideas from the start and guided, supported and shared substantial insights during the development of this dissertation.

To Jouko Orava who gave me many suggestions which were pursued in this dissertation and aided me with some of the mathematics.

And to my father who fostered my interest in science.

Resumo

Esta dissertação descreve um método para modelar, simular e renderizar aproximaçoes lineares de espaços não Euclideanos de forma genérica e em tempo real.

A técnica de renderizaçao 3D mais comum, que multiplica a matriz de projeção 4×4 por cada vértice para determinar as coordenadas do respetivo pixel no ecrã, nem sempre funciona quando o espaço não obedece a um postulado de Euclides (espaço não-Euclideano).

Além disso, enquanto outras ferramentas para renderizar espaços não-Euclideanos só funcionam para certos tipos de espaços, a nossa técnica permite modelar, simular e renderizar qualquer espaço não-Euclideano embebível isometricamente, bem como eventuais objetos nele existentes.

Antevemos pelo menos dois usos para a nossa técnica. A primeira para ajudar matemáticos a compreender melhor o aspeto de espaços arbitrários (e.g., espaço hiper-cónico, espaço hiper-esférico, etc.). A segunda para ajudar físicos a visualizar e simular os efeitos de espaço curvo (e.g., buracos negros, buracos de minhoca, deformações Alcubierra drive, etc.) em luz e objetos físicos circundantes.

Palavras-chave

Espaço não-Euclideano; buraco de minhoca; buraco negro; *ray casting*; *ray tracing*; renderização.

Abstract

This dissertation describes a method for modeling, simulating and real-time rendering piecewise linear approximations of generic non-Euclidean 3D spaces.

The 3D rendering pipeline most commonly used, where one multiplies each vertex coordinate by a 4×4 matrix to project it on the screen does not work for all cases where the space does not obey Euclid's postulates (non-Euclidean space).

Furthermore, while other non-Euclidean rendering tools only work for a limited type of spaces, our approach allows us to model, simulate, and render any isometrically embeddable non-Euclidean space and eventual objects lying therein.

We envision at least two main applications for our approach. The first for helping mathematicians get a better understanding of what arbitrary spaces look like (e.g., hyper-conical space, hyper-spherical space, and so forth). The second for assisting physicists to visualize and simulate the effects of bent space (e.g., black holes, wormholes, Alcubierre drive, and so forth) on light, and on physical objects.

Keywords

Non-Euclidean space; wormhole; blackhole; ray casting; ray tracing; rendering.

Contents

List of Figures

List of Tables

List of Abbreviations

2D	Two dimensional
3D	Three dimensional
4D	Four dimensional
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GPU	Graphics Processing Unit
OpenCL	Open Computing Language
OpenGL	Open Graphics Language
$\mathbb{E}^n$	n-dimensional Euclidean space
$\mathbb{H}^n$	n-dimensional hyperbolic space
$\mathbb{D}^n$	n-dimensional disk space

Chapter 1

Introduction

This chapter briefly describes the rationale behind the research carried out in the context of the present dissertation. This dissertation fits in the field of computer graphics and visualization.

1.1 Motivation

Computer graphics techniques have evolved alongside advancements in hardware, allowing for increasing realism of real-time rendering. However, these techniques assume the space and objects therein are Euclidean.

Real-time rendering in non-Euclidean spaces is not so common in the computer graphics literature, although it is an important part of geometry, as well as a fundamental topic used in general relativity. Furthermore, the current non-Euclidean rendering techniques are not general because they have been designed for specific non-Euclidean spaces, not for non-Euclidean spaces in general. The motivation of the present dissertation is to propose a general renderer for visualizing non-Euclidean spaces.

1.2 Research Hypothesis

Taking into consideration what was said above, we can place the following research hypothesis:

> *Is that feasible to come up with a* general *renderer for non-Euclidean spaces?*

As seen further ahead, we can say right now that this hypothesis leads to a positive statement. To achieve such a goal, we need to first discretize the space itself and find a way to propagate light rays across such space.

1.3 Research Plan

This work started with the problem of visualizing the strange effects at a wormhole's event horizon. To grasp how non-Euclidean spaces look like, as well as a clear understanding of mathematics that describes such spaces, we opted using an induction-based methodology on the dimension of spaces. We first studied 2D non-Euclidean spaces in 3D and then we carried out the necessary generalizations to 3D non-Euclidean spaces in 4D.

Concerning 2D non-Euclidean spaces, we elaborated the following research plan:

- Computationally describe a 2D wormhole surface. Usually, a 2D surface is represented in computer graphics as triangle mesh, which is equivalent to discretizing the surface (or space).
- By describing straight paths on the surface, which is a common problem in game development, we can create a ray tracer. These paths are called geodesics.
- Create the 2D wormhole surface through parametric tools so its resolution and shape may be defined interactively, allowing at the same time its generalization to spaces other than wormholes;

After that, we proceed to solve the problem for 3D space:

- Generalize such triangle mesh generation tools into tetrahedral mesh generation tools.
- Generalize the straight path calculation on top of triangles into straight path calculation on top of tetrahedra.
- Use the geodesic calculation to simulate photons and generate images in real-time (ray tracing).
- Use the tools to generate spaces other than wormholes;.
- Embed objects within the generated spaces.

After these steps, we obtained a general ray tracer not only for wormholes, but for any isometrically embeddable (and possibly non-embeddable) spaces as long as we can discretize them.

1.4 Contributions

The contributions resulting from this dissertation's work can be listed as follows:

- Modeling of arbitrary non-Euclidean spaces into discrete meshes.
- Algorithm for tracing geodesics in triangle meshes (2D case) and tetrahedral meshes (3D case). At our best knowledge, there is no similar algorithm for calculating geodesics across tetrahedral meshes.
- Ray tracer which allows for the real-time visualization of the inside view from any discretizable non-Euclidean space.

- Editing of the space (e.g., skew or change the depth of a wormhole) and visualizing the insider's view results in real-time.
- Visualization of spaces never seen before (e.g., tesseract wormholes, hyper-conical spaces).

1.5 Thesis Structure

The thesis is structured as follows:

- **Chapter 1**: This is the present chapter, which overviews the research work behind this dissertation.
- **Chapter 2**: This chapter briefly reviews the literature related to rendering in computer graphics, with a focus on non-Euclidean spaces.
- **Chapter 3**: This chapter addresses non-Euclidean spaces and techniques for their discretization.
- **Chapter 4**: This chapter describes the generic ray tracing algorithm for non-Euclidean spaces using the techniques of the previous chapter.
- **Chapter 5**: This chapter concludes the dissertation, and points out some hints for future work.

Chapter 2

Rendering in Non-Euclidean Spaces: A Review

As known, a non-Euclidean space (e.g., hyperbolic space) does not satisfy all five Euclid's postulates. Most rendering techniques were developed relying on space being Euclidean, and do not work when that is not the case. In this chapter, we review the most relevant rendering techniques and their limitations.

2.1 Introduction

Let us now briefly review the current rendering techniques and their most relevant limitations for both Euclidean and non-Euclidean spaces. As shown further ahead, the most important limitation of current rendering techniques is that they are not general in the sense that each only applies to specific spaces.

2.2 Euclidean Spaces

2.2.1 Rasterization

The most common and efficient method for drawing 3D objects into an image comprises the following steps: (i) to define each object as a polygon mesh; (ii) to multiply each mesh vertex by a 4×4 *MVP* matrix (i.e., model-view-projection matrix), which yields the position of each vertex when linearly projected onto a plane; (iii) to convert the plane coordinates to 2D pixel coordinates of the image, so that the raster "paints" the pixels between each vertex that compose a polygon. The rules by which the pixels are "painted" may range from very simple to complex, including those rules to simulate lighting that may look real. For more details about rasterization, the reader is referred to Hughes et al. [HvDF+14].

The MVP matrix is the result of $M \cdot V^{-1} \cdot P$, where M is often called the *model matrix*, a matrix which describes the position, rotation and scale of the object being drawn; V describes the position, rotation and scale of the camera (or viewer), while V^{-1} is often called the *view matrix*; and P is the *projection matrix* which is defined by the camera's near clipping plane, far clipping plane and its field of view, and it is the matrix responsible of projecting every single point onto the camera's view plane.

The projection of each mesh is equivalent to tracing a straight line between each vertex and the focal point of the simulated camera and calculating the intersection of the traced line with a plane that is at a fixed distance from the camera. Note that all rays of light that are visible to a camera (or viewer) correspond to relatively straight paths from the object reflecting or emitting that light to the camera's focal point.

When drawing the objects contained in an Euclidean space, the straight path computation can be performed through a simple matrix multiplication because Euclidean geometry dictates that two straight lines either do not intersect, intersect in all points, or only intersect at one point. Because light rays follow a relatively straight path, and that straight path must intersect the camera's focal point, each vertex must only have one 2D coordinate on the projected image.

By contrast, in non-Euclidean spaces, there may be several straight paths (i.e., geodesic paths) from a vertex to the camera's focal point. Consequently, the same vertex may be visible on several different locations of the drawn image. Thus, taking into consideration that a projection matrix multiplication by a vertex only results in one 2D coordinate in the image, using a single MVP matrix could never be used to draw objects in non-Euclidean spaces.

2.2.2 Ray tracing

Rasterization assumes that photons travel in straight paths when performing linear projections. However, that is not the case in real life. Light may be reflected, refracted by objects in the scene, not to mention its caustics effects on objects. Ray tracing techniques allow to emulate the illumination of objects in real life, and unlike rasterization, it allows for seeing the same object on different regions on the screen; for example, a reflective sphere allows to render an object already displayed in another region of the screen; that is, we see such an object twice on screen.

In order to see a real-life object, there must be photons traveling from that object to viewer's eye, which means that an illuminated object is also contributing to the illumination of all nearby points from where the object may be seen. This is called indirect illumination, and is something that can be approximated through ray tracing. These ray tracing techniques contribute to the realism of 3D scenes. However, here we are only interested in visualizing and understanding the shape and feel of non-Euclidean spaces and their embedded objects. For further details about ray tracing algorithms and techniques, the reader is referred to [Kaj86] [Laf96] [CNS+11] [RdSCP17] [CSK+17].

2.3 Mostly Euclidean, Flat, Non-Euclidean Spaces

Mostly Euclidean, flat non-Euclidean spaces are spaces where Euclid's postulates hold except on abrupt space transitions.

2.3.1 Rasterization

Valve's *Portal* (`https://en.wikipedia.org/wiki/Portal_(video_game)`) is a game that contains flat Euclidean rooms with portals connecting different parts of the room, where an object may be visible in two locations of the screen at the same time, which means there are two straight paths from the camera to that object; hence, the space is considered non-Euclidean. This drawing of an object twice was achieved multiplying every object's vertices by an MVP relative to the camera and then drawing each object again on the section of the screen where we find a portal. This second drawing of the object makes use of a different MVP which is relative to the position of the camera relative to the visible portal, and the position of the object relative to the other portal. This means that, even though one matrix yields only a position on the screen per object, but two matrices are used for different parts of the screen, allowing the same object to be in principle rendered in several different places at the same time. If a portal is visible on the second drawing, the process is repeated recursively.

Therefore, the multiple MVP-based rasterization technique behind Portal only works because the space is mostly Euclidean, having few transitions where the Euclid's postulates are broken. Using the same technique in a curved space would be impractical, because it would need an infinity of different matrices for each point in the space between the camera and the objects, as there may be no two points in the space within the same locally Euclidean zone.

2.3.2 Ray tracing

Unlike Portal, *Dygra* (`https://github.com/rameshvarun/Dygra`) is not a video game, but only a ray-tracer demo which renders an $\mathbb{E}^3$ world, where there may be special objects like spherical portals, spherical aberrations, cuboid portals and cuboid aberrations, each breaking Euclid's postulates in a different way. For example, a portal changes the ray position from one location to another, while an aberration stretches a ray in a single direction. Concerning portals, Dygra works similarly to Portal, with the difference that instead of using MVP raterization, its renderer is a ray tracer. In the particular case of cuboid aberrations, the effect of stretching a ray in a single direction inside a cuboid causes the tunnel illusion; that is, it looks shorter when seen from the inside and longer when seen from outside.

Summing up, Dygra encodes a flat Euclidean world, which is populated with objects where Euclid's postulates are broken. Therefore, this type of non-Euclidean space is considered mostly Euclidean. Curved spaces like hyperboloid spaces cannot be drawn using this technique because even though some objects may break Euclid's postulates, they may only be placed in Euclidean space through 3D Cartesian coordinates, having no way of freely (in all directions) bending or stretching the space to make more 'space'.

2.4 Curved Non-Euclidean Spaces

Unlike *flat* non-Euclidean spaces, where there are a few concrete boundaries where the space abruptly breaks Euclid's postulates, in *curved* non-Euclidean spaces, the transitions may be between all points.

2.4.1 Rasterization-based techniques

As described below, hyperbolic spaces are curved non-Euclidean spaces for which rasterization techniques have been developed. Rasterization is possible in hyperbolic spaces because any two geodesics diverging from one point do not converge into another point, meaning that the same point does not appear on different places of the screen, which is one of the limitations of using linear projections as mentioned in Section 2.2.1.

At our best knowledge, the animation clip called "Not Knot" produced by Gunn and Maxwell [GM91] allowed us to visualize the inside of the hyperbolic space for the first time. On the video, the space is tessellated with right-angled dodecahedra. This animation is based on a rasterization-based technique described by Phillips and Gunn [PG92] and Gunn [Gun93]. Firstly, Klein or projective model is used, so that every vertex is projected from $\mathbb{H}^3$ into $\mathbb{D}^3$ (disk-space) where the center of the disk is the point of view. These coordinates are then fed into a renderer which generates images by conventional methods; for example, by projecting the disk-space coordinates onto 2D screen coordinates.

The projection between hyperbolic and disk coordinates is injective; a point in $\mathbb{H}^3$ corresponds to a single point in $\mathbb{D}^3$. The same applies to the projection from $\mathbb{D}^3$ onto screen space. For a more intuitive explanation of rendering $\mathbb{H}^3$ the reader is referred to Weeks [Wee02] and Hart et al. described how to render $\mathbb{H}^3$ [HHMS17a]. Interestingly, Hart et al. also described how to render and $\mathbb{H}^2 \times \mathbb{E}$ [HHMS17b] using the same technique, allowing for an exploration of these spaces using virtual reality.

These algorithms are of course limited to rendering hyperbolic spaces, and although interactive in what is allowed to be placed inside the space, the space itself is not deformable.

2.4.2 Ray tracing-based techniques

By tracing individual rays of light, one can render any space for which geodesics can be calculated. This is true because light closely follows a straight path, and an object is visible when photons travel from its surface to an observer's retina, crossing a focal point. Therefore, the ability to trace the straight paths from the retina through a focal point allows us to know what is within the line of sight of the viewer, and opens up many possibilities for rendering non-Euclidean curved spaces.

The movie *Interstellar* released in 2014 attempted to demonstrate what blackholes and wormholes would look like in reality. Both these spaces cannot be rendered using conventional methods since space around blackholes and wormholes is curved, making it non-Euclidean. To solve this problem, James et al. developed the renderer *DNGR* [JvTFT15a], which uses ray tracing in Boyer-Lindquist polar coordinates. One of the problems with ray-tracing in curved space is that the rays are incremented in discrete steps on a space with smooth curvature. Therefore, if the tracer is naively implemented, we end up getting error accumulation that is more noticeable in locations where the space is more curved. In order to truncate the error from each integration step, the step size must adapt depending on the local curvature of the space. *DGNR* solves this using a customized Runge-Kutta integration. This method is computationally expensive, and because of the polar coordinates used, this solution only works for the specific problem of rendering spaces blackholes and wormholes since space deformations need to be symmetrical relative to the origin of the polar coordinate system. For a better understanding of wormhole space, the reader is referred to James et al. [JvTFT15b].

2.5 Summary

This chapter provides an overview of the state-of-art of rendering and ray tracing techniques for non-Euclidean spaces. These techniques are scarce in science and engineering literature, but we can envision their applications in cosmology, physics, mathematics, and so forth. The main drawback of these techniques is that they are not general, that is, they only apply to specific non-Euclidean spaces.

Chapter 3

Space-Mesh Generation

This chapter goes over the technique and tools developed for the generation of space-meshes which represent piecewise linear approximations of 3D non-Euclidean spaces isometrically embeddable in 4D Euclidian space. This technique is general to any space with a natural topology because it does not depend on whether a coordinate system has been theorized for such space or not. For example, while a Schwarzschild black-hole may be described using Boyer–Lindquist polar coordinates, it is a specific coordinate system for such space. In fact, it does not apply to all spaces. Even worse, it is not possible to describe some less well-behaved spaces through such linear systems (e.g., space deformations by gravitational waves by arbitrary bodies). In contrast, our technique allows us to construct a 3D non-Euclidean space from tetrahedra (i.e., building blocks), which are Euclidean spaces. This space discretization into space-meshes is the ground for the ray tracing algorithm described in the next chapter.

3.1 Introduction

Since we are trying to visualize non-Euclidean spaces embedded in 4D space, and since human brains have an intuitive understanding of 3D space but not of 4D space, we start by creating tools to work with on a lower dimension analogue which is intuitively understood, and then we generalize such tools to higher dimensions.

Recall that our main goal is to render 3D non-Euclidean spaces by isometric embedding, so we will first theorize on 2D non-Euclidean spaces embedded in 3D space, and then we generalize our theory and spatial structures to 3D counterparts embedded in 4D Euclidean space.

3.2 Theoretical Background

The most general definition of a space is a topological space, which is a set of points whose neighborhoods satisfy a number of axioms. This definition includes manifolds, which are topological spaces that are locally Euclidean for each small neighborhood of each point. Metric spaces are another way of defining specific spaces through a distance function (or *metric*) which defines the distance between any two points.

Because one of the goals of the dissertation is to discretize spaces, we will focus on approximating manifold spaces with manifold simplicial complexes, here called space-meshes. For 2D spaces, the space-mesh is composed of triangles connected by their edges, and for 3D space, the space-mesh is composed of tetrahedra connected by their faces.

3.2.1 Euclidean Space

An Euclidean space is a topological space where the postulates of Euclidian geometry apply:

1. A straight line segment can be drawn joining any two points.
2. Any straight line segment can be extended indefinitely in a straight line.
3. Given any straight line segment, a circle can be drawn having the segment as radius and one endpoint as center.
4. All right angles are congruent.
5. If two lines are drawn which intersect a third in such a way that the sum of the inner angles on one side is less than two right angles, then the two lines inevitably must intersect each other on that side if extended far enough. This postulate is equivalent to what is known as the parallel postulate.

If at least one of these postulates does not apply, the space is considered non-Euclidean. There are various types of non-Euclidean spaces, namely the hyperbolic, spherical, toroidal spaces. Because any manifold can be used to describe a space, there is an infinity of possible spaces.

3.2.2 Toroidal Space

A toroidal space is a space described by a torus shape. A torus has positive curvature on its outer rim and negative curvature on its inner rim. The construction of a torus is as follows. When traveling on a finite rectangular 2D surface, we observe that motion is limited by the edges of rectangle. If two of the parallel ends were connected, the space would become a cylinder, and one could travel infinitely in a circle on its surface. Then, by connecting the two remaining circular edges, the shape becomes a torus, which means that, in the outer rim, parallel straight lines converge like in spherical space, and in the inner rim, parallel lines diverge.

3.2.3 Conical Space

Conical spaces have positive Gaussian curvature which increases to infinity as one aproaches its apex, since its radius tends to zero. This means straight lines on its surface, when seen from outside the space, seem to always curve downwards as if a force is pulling

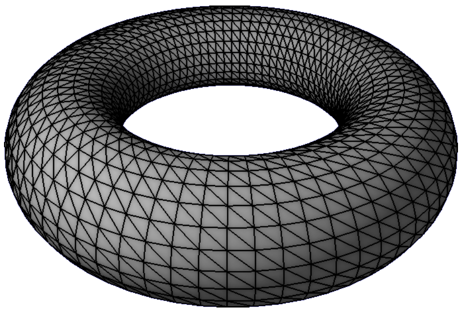


Figure 3.1: Embedded torus space

them down. Because the only edge of a conical space is in its base, all straight lines on its surface, with the exception of the base circle itself, intersect the base circle twice.

3.2.4 Hyperbolic Space

Hyperbolic space is a space described by hyperbolic geometry, which has constant negative curvature.

Hyperbolic geometry was achieved by replacing Euclid's fifth postulate with:

For any given line l and point P not on l, in the plane containing both line l and point P there are at least two distinct lines through P that do not intersect l.

It has been proven that there is a global isometric embedding in 6-dimensional Euclidean space of 2D hyperbolic space, and it has been proven that it is impossible to for a 3-dimensional global isometric embedding to exist. Because of this, even though we model and render negative curvature spaces which are local isometric embeddings of hyperbolic space, a complete global embedding was not tried in this dissertation.

3.2.5 Schwarzschild black-hole Space

Gravity deforms space around mass which causes light traveling on the surface of the space to describe paths which break Euclid's postulates, meaning we cannot render images near large concentrations of mass using conventional rendering techniques. Black-holes are bodies with a particular case of such deformation, where the gravitational effect is so strong that within it, inside what is called the event horizon, nothing can escape from it.

The simplest type of black-hole, which is the one studied in this dissertation, is called static black-hole, or Schwarzschild black-hole, which has no electric charge and has a spherical shape due to it having no angular momentum. The Schwarzschild radius is the distance to the center of the black-hole at which the event horizon is found and can be calculated as:

$$r_s = \frac{2GM}{c^2} \tag{3.1}$$

where G is the gravitational constant, M is the black-hole's mass in kilograms and c is the speed of light in meters per second.

The Schwarzschild space's metric can be described as:

$$c^2dr^2 = \Big(1 - \frac{r_s}{r}\Big)c^2dt^2 - \Big(1 - \frac{r_s}{r}\Big)^{-1}dr^2 - r^2\big(d\theta^2 + \sin^2\theta d\varphi^2\big) \tag{3.2}$$

The deformed space surrounding a Schwarzschild black-hole can be isometrically embedded in a higher dimension and because of the symmetry inherent in the space, one dimension can be simplified out, so the resulting surface, called Flamm's paraboloid, can be visualized without losing any information. To calculate Flamm's paraboloid, we can go through all points of the plane (x, y) with a distance r to the center of the black-hole, and offsetting them in z by w, because we only care about a single slice in time, and because we are generating a 2D simplification of the space, we can solve the Schwarzschild metric for the equatorial slice and with time as a constant ($\theta = \frac{\pi}{2}, dt = 0$), convert the result to cylindrical coordinates which preserve the same distances, giving us the following equation:

$$w = 2\sqrt{r_s(r - r_s)} \tag{3.3}$$

3.2.6 Schwarzschild Wormhole Space

A wormhole, or Einstein-Rosen bridge, is a theoretical concept consistent with general relativity, which can be imagined as a tunnel between two locations in spacetime. Although wormholes have not been proven to exist, and they are theorized to be impossible to exist naturally, they are an interesting concept to visualize.

In this dissertation, the shape created for simulating a wormhole is similar to two Schwarzchild black-holes connected by their event-horizons.

3.3 Geometric Data Structures

To represent any space and eventual objects in such a space, we use a geometric data structure for meshes. In 3D computer graphics, a mesh is a collection of vertices, edges and faces. Using meshes, artists can create approximations of real-life atom-based objects and render nearly photo-realistic images of those objects. In a similar fashion, this dissertation uses meshes. Instead of meshes being used to draw approximations of visible objects, the work developed in this dissertation uses meshes for defining the approximation of topology of manifold spaces.

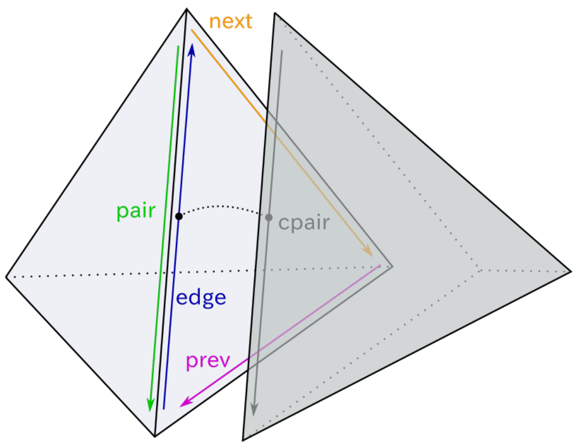


Figure 3.2: All half-edge relationships relative to an $edge$ which is painted blue.

All meshes mentioned in this dissertation are an extension of the half-edge structure. This extension includes half-faces and cells, and edges have an additional reference to the cell-pair, abbreviated as *cpair*. Note that, given an edge e, to get its $cpair$, we traverse the following sequence of relationships: (i) we first obtain the face f of e; (ii) we obtain the pair of f which is on the adjacent cell; (iii) we obtain the edge bounding the pair of f composed of the same vertices as e. These connectivity relationships are illustrated in Figure 3.2, which are encoded below.

```
typedef struct vertex_t
{
    vecN_t pos; /* euclidian position, can be a vec3 or vec4 */
    vec4_t color;
    int half; /* edge_t */
    int tmp; /* auxiliar variable */
} vertex_t;
```

```
typedef struct edge_t /* Half edge */
{
    int v; /* First vertex in edge */
    vec3_t n; /* normal of v */
    vec2_t t; /* texture coord of v */

    int face; /* face_t */
    int pair; /* edge_t */
    int next; /* edge_t */
    int prev; /* edge_t */
#ifdef HALF_FACE
    int cpair; /* edge_t */
#endif

    int tmp; /* auxiliar variable */
} edge_t;
```

```
typedef struct face_t /* Half face */
{
    int e_size;
    int e[4]; /* edge_t[e_size] */

    vec3_t n; /* flat normal, only applicable to triangles */

#ifdef HALF_FACE
    int pair; /* face_t */
    int cell; /* cell_t */
#endif

    int tmp; /* auxiliar variable */
} face_t;
```

```
typedef struct cell_t /* Cell */
{
    int f_size;
    int f[5]; /* face_t[f_size] */
    int tmp; /* auxiliar variable */
} cell_t;
```

```
typedef struct mesh_t
{
        vector_t *verts;
        vector_t *edges;
        vector_t *faces;
#ifdef HALF_FACE
        vector_t *cells;
#endif
        mesh_selection_t selections[16];

        khash_t(id) *edges_hash; /* hash-table of unpaired edges ids */
#ifdef HALF_FACE
        khash_t(id) *faces_hash; /* hash-table of unpaired faces ids */
#endif


        int has_texcoords; /* do edges have texture coordinates */
        int triangulated; /* does it contain non-simplexes */

        mat4_t transformation_stack[4]; /* only works for R3*/

        int changes; /* Has changed since last update */
        int update_id; /* tracks current update version */
        int locked_write;
        int locked_read;
        void *semaphore;
        ulong owner_thread; /* id of thread modifying the mesh */

        char name[256];
} mesh_t;
```

3.4 Geometric Operations

3.4.1 Extrusion

In polygonal modeling contexts, extrusion modifies a mesh by transforming all n-dimensional primitives into their $(n+1)$-dimensional versions (Figure 3.3). Vertices become edges, edges become faces, faces become cells. Each new vertex created from the extrusion is then translated by a user-defined offset. Extruding a piecewise-linear circle is shown in Figure 3.4, whereas the extrusion of a piecewise-linear sphere is depicted in Figure 3.5.

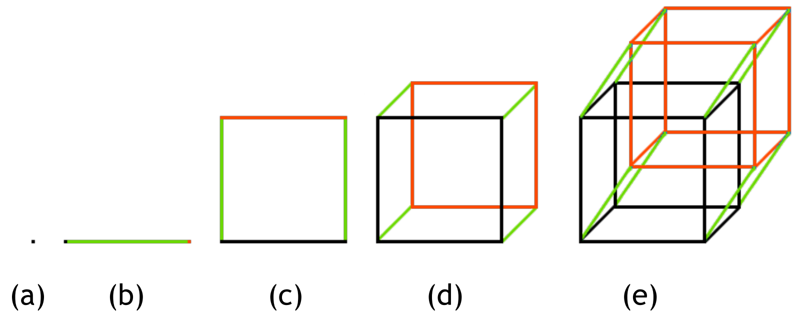


Figure 3.3: A sequence of four extrusions starting with a single point (a). In each extrusion (b) to (e), the input geometry in black is cloned and translated into blue. The space between the black and blue geometry is filled by green geometry. For the sake of simplicity, we omit the faces and solids generated when extruding, showing only the edges and vertices.

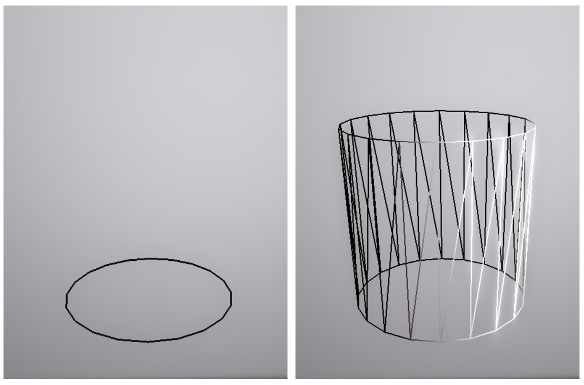


Figure 3.4: Extrusion of a circle made of edges (left) into a cylinder made of triangles (right).

3.4.2 Unpaired geometry removal

Half-edges that do not belong to a face and half-faces that do not belong to a cell need to be removed from the mesh before rendering, because to avoid time consuming unnecessary checks, all half-edges are assumed to have a face and all half-faces are assumed to have a cell.

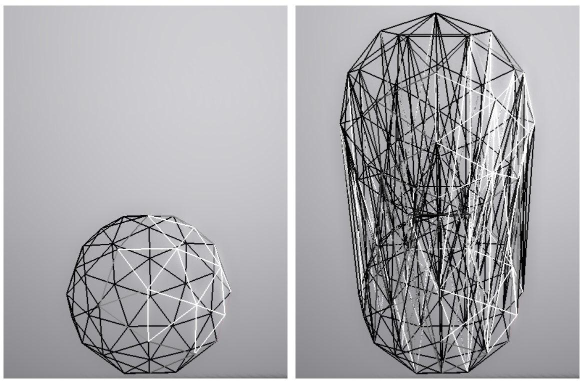


Figure 3.5: Extrusion of a sphere made of triangles (left) into a hyper-cylinder made of tetrahedra (right).

3.4.3 Welding

Welding the process of collapsing all selected vertices into a single one (Figure 3.6). The tool removes each triangle and tetrahedra composed by more than one of the selected vertices, the triangles and tetrahedra that remain need to update their pairs.

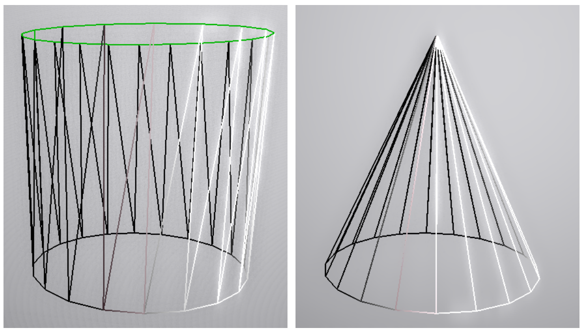


Figure 3.6: Cylinder (left) with green selected edges welded (right).

3.4.4 Triangulation

Because the space traversal algorithm only works on simplexes, and because extrusions generate quads and prisms which are not simplexes, they have to be split into triangles and tetrahedra respectively, so we end up obtaining a triangulation.

3.4.4.1 Quad triangulation

Triangulation in a polygonal mesh with quads is straightforward, each quad can be split in one of two ways, but either way works. Table 3.1 represents the possible configurations for triangulation of a quad, which are shown in Figure 3.7.

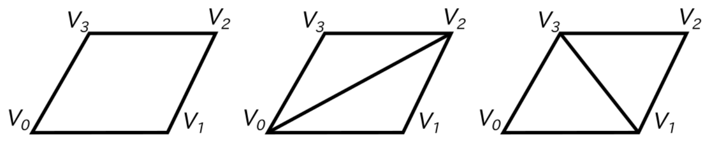


Figure 3.7: Original quad and two possible configurations for subdivision.

configuration	triangle 0	triangle 1
0	V0 V1 V2	V0 V2 V3
1	V0 V1 V3	V1 V2 V3

Table 3.1: The two possible configurations for subdividing a quad composed by the vertices V0 V1 and V2.

3.4.4.2 Prism triangulation

A prism is made of two triangles and three quads (Figure 3.8), while tetrahedra are only made of four triangles. To split a prism into tetrahedra, each of its three quads has to be split into two triangles, as stated before, each quad has two possible ways it can be split into triangles, therefore, there are eight combinations of how the prism's quads may be subdivided. It is impossible to create tetrahedra out of the triangles created from subdividing all of the quads with the exact same configuration [DLVC99], which means two of the eight configurations do not yield tetrahedra and must be discarded. Table 3.2 is used to generate the tetrahedra given a specific configuration. The resulting tetrahedra for each configuration is shown in Figure 3.9.

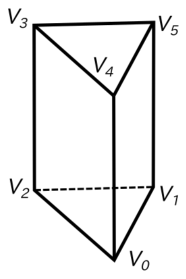


Figure 3.8: Prism.

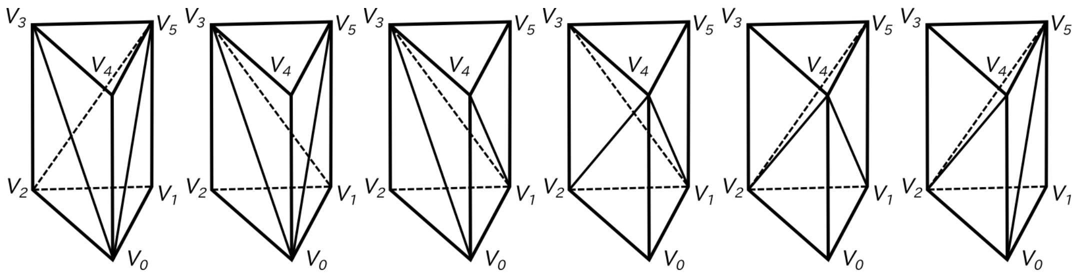


Figure 3.9: The six possible configurations for subdivision of a prism into tetrahedra.

configuration	tetrahedron 0	tetrahedron 1	tetrahedron 2
000	-	-	-
001	V0 V1 V2 V5	V0 V1 V5 V4	V0 V4 V5 V3
010	V0 V1 V2 V4	V0 V4 V2 V3	V3 V4 V2 V5
011	V0 V1 V2 V4	V0 V4 V5 V3	V0 V4 V2 V5
100	V0 V1 V2 V3	V1 V2 V3 V5	V1 V5 V3 V4
101	V0 V1 V2 V5	V0 V1 V5 V3	V1 V3 V4 V5
110	V0 V1 V2 V3	V1 V4 V2 V3	V2 V4 V5 V3
111	-	-	-

Table 3.2: The six possible and two impossible configurations for subdividing a prism composed by the vertices V0 V1 V2 V3 V4 and V5, modified from [DLVC99]. Each bit in the configuration represents which of the two possible configurations each quad of the prism will use in its subdivision.

3.4.4.3 Half-face pair congruence

Although triangulating a mesh of quads paired up by half-edges, each quad's triangulation does not depend on its pair faces, since the edges themselves do not get altered, the same cannot be said for triangulating a prism mesh, because the triangulation modifies the quads of the prism, which are only half-faces that may be paired up with a quad of a connected prism, since the resulting triangles of the triangulation of both paired half-faces must be paired up as well, the way one quad is subdivided is going to enforce that its paired quad must be subdivided in the inverse way.

Because a prism's subdivision determines how one of the connected prism's face will subdivide, and because not all configurations of how a prism's quads may be triangulated are valid (since some do not yield tetrahedra), a problem emerges that if the subdivision configuration for each prism is chosen arbitrarily, they may force other prisms into non-valid configurations. This means the subdivision configuration for each prism must be planned taking into account the whole mesh.

3.4.5 Circles and Spheres

Many of the spaces to be modeled in this dissertation can often be derived from the result of applying extrusions to a circle or a sphere, because of this, some generators with parametric fidelity were developed.

Since in two dimensions, all natural numbers are a valid number of edges to a regular convex polygon, the higher the number, the closer the polygon is to a circle as seen in Figure 3.10.

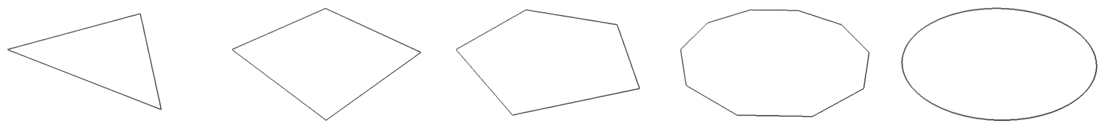


Figure 3.10: Convex regular polygons with different number of segments.

Because in three dimensions, the number of regular polyhedra possible to create is five, it is impossible to create a sphere with parametric precision composed by equilateral triangles of same size. A way around this, is to start with a regular polyheron made of triangles, commonly an icosahedron, then perform consecutive smooth subdivisions, the higher the subdivision, the closer the solid is to a sphere. Although this technique does not generate perfectly equal triangles, the sphere is not regular, but since it would be impossible otherwise, the results are acceptable (Figure 3.12).

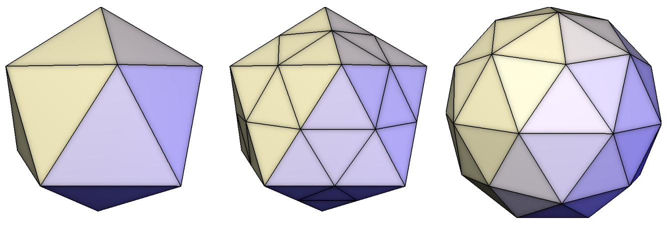


Figure 3.11: No subdivision on the left; new triangles on the middle; normalized radius of new vertices on the right.

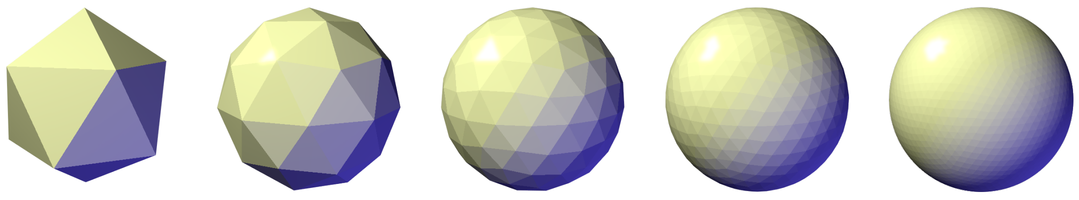


Figure 3.12: Icosahedron on the left followed by consecutive subdivisions.

Subdivisions are made by creating a new vertex in the middle of all edges, and replacing each face with four new triangles using the new vertices as shown in Figure 3.11.

Because we want the subdivision to approximate spheres, the distance from each vertex to the origin has to be normalized and scaled by the desired radius.

3.5 Modeling the Schwarzschild black-hole

3.5.1 Space-Mesh for 2D black-hole embedded in $\mathbb{E}^3$

To create the 2D space deformed by a Schwarzschild black-hole, we can start with a circle made of edges. Each edge is extruded into quads. At the end of each extrusion step, the new vertices' position is scaled down so the space gets generated towards the center of the circle until we reach the Schwarzschild radius, depending on the distance of each vertex to the center of the black-hole, we calculate the offset deformation in the z dimension which is perpendicular to the starting circle, using the Flamm's paraboloid offset. By scaling each step of the extrusion along an equation, we get an approximated shape of Flamm's paraboloid, the accuracy of this shape depends on the number of segments on the original circle and the number of steps of the extrusion (Figure 3.13). Each quad is split into triangles (triangulation). The original edge circle is removed since its edges are not associated with any face.

3.5.2 Space-mesh for 3D black-hole embedded in $\mathbb{E}^4$

Because of the spherical nature of Schwarzschild's metric, and because of spherical symmetry, the *w* offset depends only on the distance to the center of the black-hole, which means we can use the same equation to generate a 3D black-hole as well, only instead of starting with a circle made of edges, we start with a sphere made of triangles, and instead of offsetting the vertices on the third dimension, we offset in the dimension perpendicular to the sphere, which, will in this case be the fourth dimension, since the sphere's coordinates will all be contained in the volume $(w = 0)$ (Figure 3.14).

By concentrating more geometry in the location with higher curvature, we get higher fidelity, as shown in Figure 3.15. This can be achieved by allocating more extrusion steps with lower increments where more geometry is necessary.

3.6 Modeling the Schwarzschild wormhole

If we extend the black-hole generation by applying a second extrusion with the event horizon selected, and reverse the percentage fed to the scaling and offset function, we can transform the blackhole into a wormhole, whose results can be seen in Figure 3.16.

3.7 Summary

In this chapter, we have introduced the theoretical background (mathematics) behind non-Euclidean spaces, as well as their decomposition (discretization) into building blocks. We use simplices as the fundamental building blocks of such decomposition or triangulation. As shown above, we use an extended half-edge data structure to represent space decompositions. The resulting mesh is of paramount importance not only because they apply to general non-Euclidean spaces, but also because these decompositions are the basis for calculating geodesics, as necessary in ray tracing (see next chapter for further details).

2D

```
float flamm(float radius)
{
    return 2.0f * sqrtf(RS * (r - RS
        ));
}
mesh_t *blackhole2d()
{
    mesh_t *space = mesh_new();
    mesh_circle(space, 1, 20);
    mesh_extrude_edges(space, 15,
            vec4(0, 1, 0, 0), RS,
            flamm);
    mesh_remove_lone_edges(space);
    mesh_triangulate(space);
    return space;
}
```

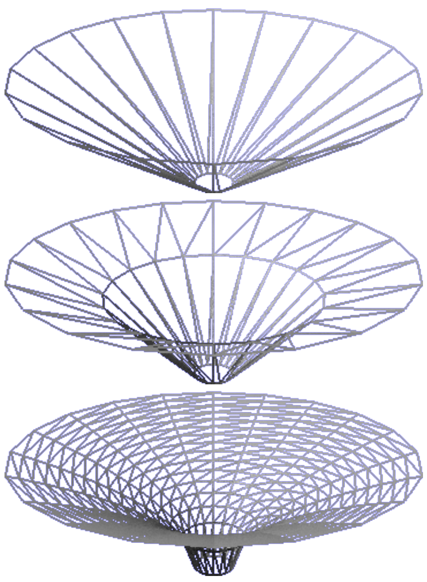


Figure 3.13: 2D black-hole models with differing number of steps, first one, then two, then fifteen.

3D

```
float flamm(float radius)
{
    return 2.0f * sqrtf(RS * (r - RS
        ));
}
mesh_t *blackhole3d()
{
    mesh_t *space = mesh_new();
    mesh_icosphere(space, 1, 2);
    mesh_extrude_faces(space, 15,
            vec4(0, 0, 0, 1), RS,
            flamm);
    mesh_remove_lone_faces(space);
    mesh_triangulate(space);
    return space;
}
```

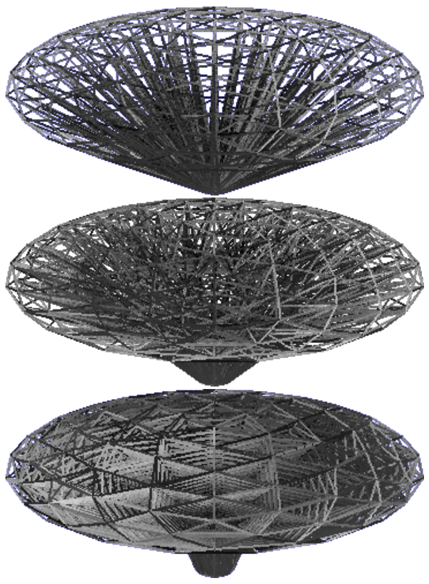


Figure 3.14: 3D black-hole models with differing number of steps, first one, then two, then fifteen.

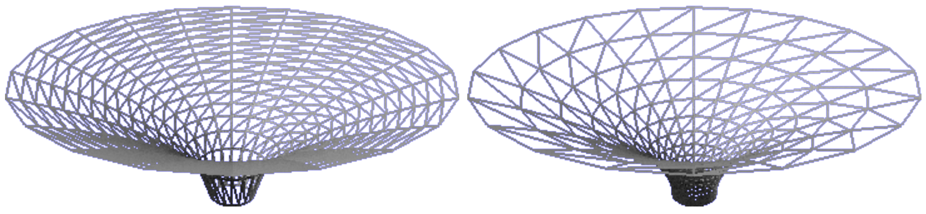


Figure 3.15: On the left, linear scale increment by step, on the right, quadratic. Same number of steps, but on the left, the steps are more concentrated in the most vertical zone.

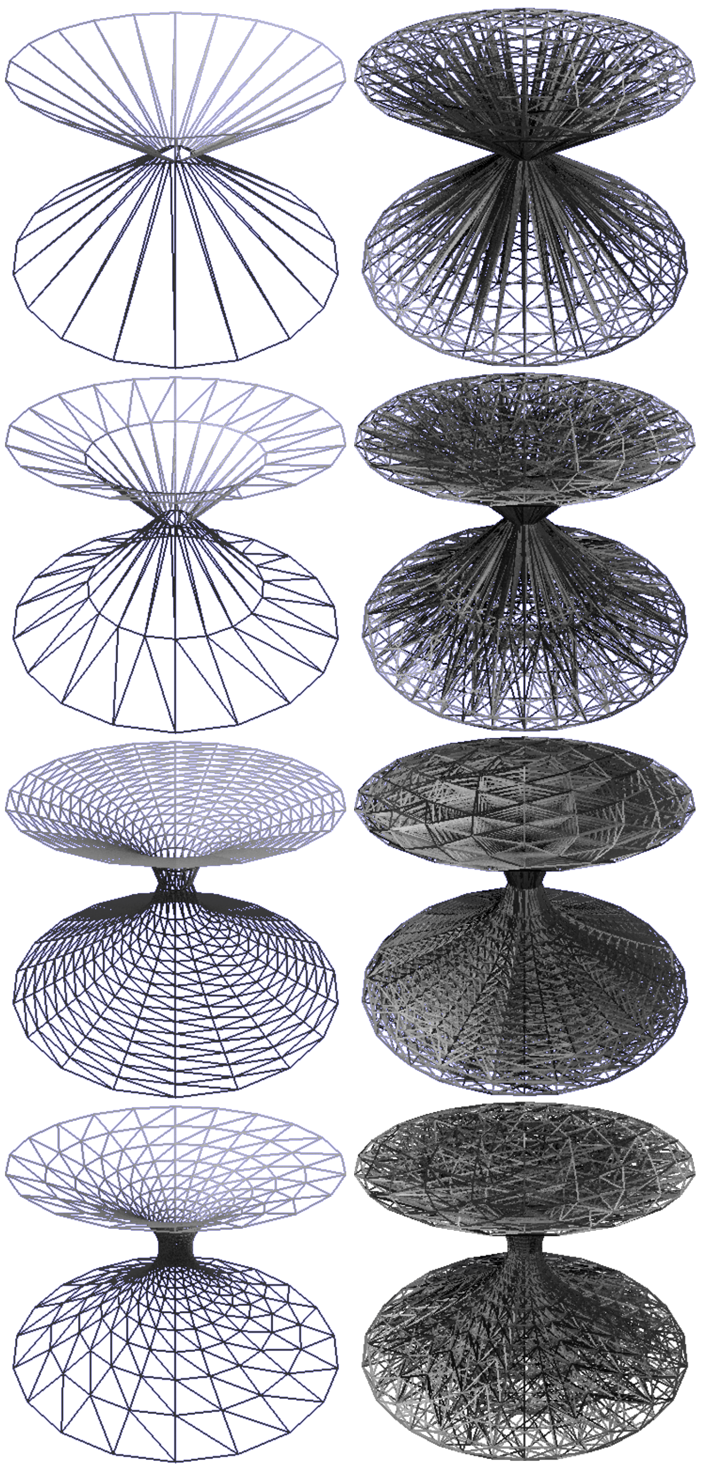


Figure 3.16: 2D and 3D wormhole models with two steps, then four, then thirty, then thirty with quadratic scale increment.

Chapter 4

Ray-tracing Non-Euclidean Spaces

This chapter describes an algorithm to render *general* curved non-Euclidean spaces (e.g., the Schwarzchild space that describes a black hole). For that purpose, a non-Euclidean space first has to be discretized as detailed in the previous chapter. That is, we need to generate a space into a triangulated mesh, here called space mesh. Recall that, as seen in Chapter 2, there are algorithms to render *specific* curved non-Euclidean spaces, but not to render *general* curved non-Euclidean spaces like the one here proposed.

4.1 Introduction

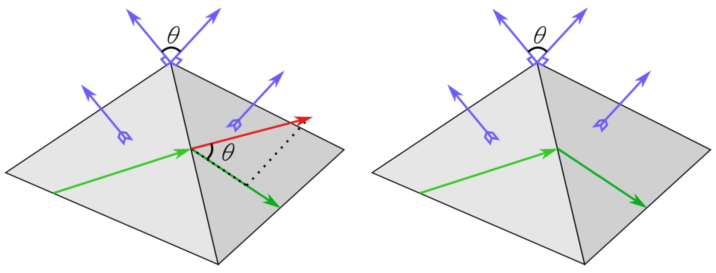
Unlike the straight light rays used in Euclidean spaces to render 3D scenes using conventional rendering techniques, including ray casting and ray tracing, the light paths in non-Euclidean spaces are geodesics, which are considered straight inside the space, but from outside, they might look curved depending on the curvature of the space. Intuitively, what we are trying to do is to render scenes using curved rays (i.e., geodesics). Note that an observer inside a non-Euclidean space sees a geodesic ray as straight. Therefore, by calculating geodesics emanating from a surface point of a non-Euclidean space, one can render an image of what an observer (or camera) sees when such observer is located within the space. Each geodesic (or non-Euclidean straight line) effectively represents a possible path taken by light until reaching the camera.

Let us now see how this geodesic-based rendering works in two cases:

- 2D non-Euclidean spaces embedded 3D space.
- 3D non-Euclidean spaces embedded 4D space.

The second case can be seen as a generalization of the first case, which helps to understand what is going on in higher-dimensional non-Euclidean spaces. Essentially, our geodesic-based rendering algorithm generalizes the calculation of geodesics on triangle meshes in 3D to tetrahedral meshes in 4D. Recall that the triangulation of each non-Euclidean space is represented by a half-face data structure, so the connectivity between simplices can be easily retrieved whenever necessary.

Section 4.2 describes how to calculate geodesics in 2D space-meshes. 4.3 describes how to calculate geodesics in 3D space-meshes. Section 4.4 shows how to use geodesics to trace light paths to the camera and render an image. Section 4.7 shows how to move along a geodesic with a non-infinite velocity.



(a) (b)

Figure 4.1: Illustrating how a geodesic path (in green) would leave the surface of the mesh if not rotated by θ. The triangle normals appear in blue.

4.2 Geodesic Rays on Triangle Meshes

In conventional ray casting and ray tracing, a light ray is defined by a point (observer's eye) and a 3D directional vector that is supported by a straight line because the rendering occurs in the 3D Euclidean space. Similarly, in geodesic-based ray casting and tracing, a light ray is also defined by a 3D point and a 3D directional vector, with the difference that we are using geodesic rays instead of straight rays. However, in this section, we are calculating geodesics on triangle meshes, therefore, a ray's position and direction are 2D.

4.2.1 Door-In-Door-Out

Our algorithm uses a door-in-door-out method; that is, the ray enters at a point of the triangle boundary and finds its way out on other boundary point. Therefore, one possible method to calculate a geodesic path (or ray) on a triangle mesh in 3D is as follows:

1. Find the triangle where the ray point lies in; if there is no such triangle, there is no geodesic because the point is outside of the mesh. The directional vector is perpendicular to triangle normal (or tangent to the triangle).
2. Calculate where the ray exits the triangle by calculating the intersection of the ray with the three planes defined by its edges and triangle's normal.
3. Move the point to the intersection point on the boundary of the triangle.
4. Calculate the angle θ between the current triangle and the next triangle. These triangles share the intersection point in one of their boundary edges.
5. Rotate the direction by θ on the axis which is the cross product of the two normals.

This process in Cartesian coordinates is illustrated in Figure 4.1. However, this process have two main shortcomings. First, the rotation θ does not ensure that the path gets on the surface mesh because the computation of θ involves floating-point calculations whose errors easily sum up. This means that we need extra computations to assure

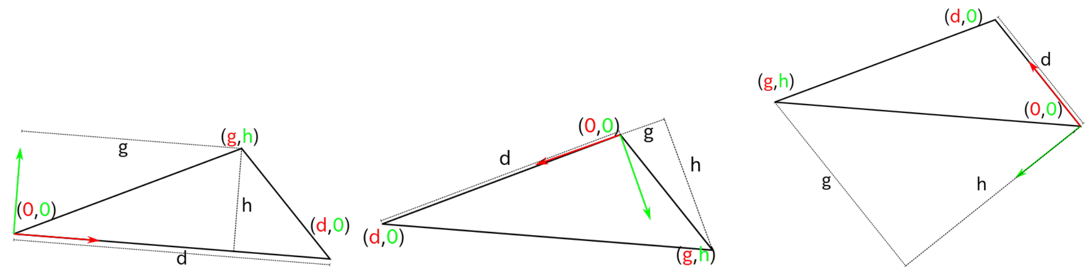


Figure 4.2: 2D Cartesian coordinates of each triangle vertex depending on the edge working as the base. The red axis is x, the green axis is y

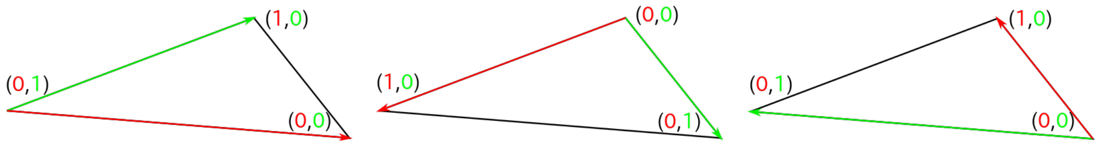


Figure 4.3: 2D barycentric coordinates of each vertex depending on which edge serves as base. The red axis is u, the green axis is v

the geodesic path vector always remains in each traversing triangle, yet within a given error threshold. Second, this process is time-consuming for long paths because of the evaluation of many multiplications and trigonometric functions (i.e., cosine and sine functions of rotation matrix).

To solve these problems, we use the coordinate system of the surface mesh itself, that is, 2D barycentric or Cartesian coordinates rather than 3D Cartesian coordinates. This solution a priori guarantees that the light ray does not escape from the surface mesh. In other words, there is no need for rotations. Therefore, a geodesic path is determined as follows:

1. Start with a ray composed of a 2D point and 2D direction using local 2D barycentric or Cartesian coordinate frame of a triangle in the space-mesh.
2. Determine the edge intersected by the ray.
3. Move the point to the intersection locus.
4. Convert the point coordinates and direction to a local frame of the next triangle.

This method is agnostic to normals or any global coordinates, all that matters is the shape of the triangles, which means there are no rotations or bends to fix. In practice, we opted by using barycentric coordinates because they make ray-edge intersection and coordinate conversion between adjacent triangles easier to calculate than 2D Cartesian coordinates, as illustrated in Figures 4.2 and 4.3. But, barycentric coordinates take advantage over Cartesian coordinates because computations are confined to each triangle's interior and boundary.

In 2D barycentric coordinates, there are two axes, u and v. Each axis starts at the same point and is aligned with one of the triangle edges. Any triangle vertex can work as the origin of the barycentric coordinates. After pinning the origin at a vertex, there are two possible configurations for the frame of barycentric coordinates (Figure 4.3). This means there are six possible barycentric coordinate systems per triangle.

To keep things even more simple and topologically consistent, we take advantage of the half-edge data structure to represent not only the surface mesh but also the barycentric coordinates of each triangle. Therefore, the u axis will always be aligned with the half-edge whose vertex is the origin, and the v axis will always be aligned with the half-edge whose *next* vertex is the origin. This means there are only three unambiguous barycentric coordinate systems, one per half-edge (or base), for each triangle. Note that the half-edge representation allows us to know which is the next triangle to be travelled by the light ray; the next triangle is retrieved using the twin of the current door-out half-edge crossed by the light ray. Summing up, to describe a point in the surface mesh, one needs only to store its 2D barycentric coordinates and triangle's *base*, so describing not only the coordinate system being used, but also which triangle the point is currently in.

4.2.2 Coordinate conversions

Every point in the simulated 2D space can be uniquely represented using either:

- 2D barycentric (u, v) local coordinates and a *base*. These coordinates are necessary to determine the geodesic paths or rays.
- 2D Cartesian (x, y) local coordinates and a *base*. These coordinates are necessary to perform orthogonal operations (e.g., ray-circle intersection). Recall that barycentric coordinates are not orthogonal, i.e., they are skewed.
- 3D Cartesian (x, y, z) global coordinates of the Euclidean space in which the space-mesh is embedded. For example, these coordinates are necessary to compute edge lengths, and draw the space itself as seen from outside. These global 3D coordinates are not sent to GPU.

In order to convert Cartesian coordinates to and from barycentric, we need to calculate the (d, g, h) values relative to each possible base edge, as illustrated in Figure 4.2, where d is the length of the base, or the x Cartesian coordinate of the base's next vertex, h is the height of the triangle or the y Cartesian coordinate of the point which is not on the base, and g is the x Cartesian coordinate of the point which is not on the base. These variables are stored in a new structure *base_t* as follows:

```
typedef struct
{
    float d;
    float g, h;
    int next, prev, pair; // half-edges
} base_t;
```

Note that the next, previous, and the pair half-edges also go with the *base_t* for navigation across the half-edge data structure. The C functions to convert from local Cartesian to (local) barycentric coordinates, and vice-versa, and (local) barycentric to global Cartesian coordinates are as follows:

```
void cartesian(base_t *base, vec2_t *b)
{
    b->x = base->d * b->u + base->g * b->v;
    b->y = base->h * b->v;
}

void barycentric(base_t *base, vec2_t *c)
{
    c->u = (c->x - (c->y * base->g / base->h)) / base->d;
    c->v = c->y / base->h;
}

vec3_t barycentric_to_global(vec2_t pos, edge_t *base)
{
    vec3_t v0 = e_vert(base)->pos;
    vec3_t v1 = e_vert(e_next(base))->pos;
    vec3_t v2 = e_vert(e_prev(base))->pos;
    return vec3_add(
            v0,
            vec3_scale(vec3_sub(v1, v0), pos.u),
            vec3_scale(vec3_sub(v2, v0), pos.v));
}
```

Because there is one *base_t* (on GPU side) for each half-edge, it is possible to insert these values directly into *edge_t* (on CPU side). However, this is not done because *edge_t* contains much more data than needed for calculating a ray's path, and since the GPU needs to have access to (d, g, h) values, sending the *edge_t* vector directly would be inefficient in terms of memory usage.

Figure 4.4 shows the GPU inputs (OpenCL and OpenGL inputs) and their original host structures on CPU side. While OpenGL is used to model and draw meshes from the outside point of view, OpenCL is used for raytracing *within* those spaces.

The *mesh_t* structure contains the list of geometrical primitives (vertices, edges, and faces) with information needed for space modelling like half-edge relations, primitive selections, synchronization semaphores and others, as seen in Section 3.3 of the previous chapter. On the other hand, *mesh_gl_t* fetches information from *mesh_t*, selecting only the information (list of triangles) needed for OpenGL rendering of the mesh on screen. *mesh_cl_t* contains the inputs to the raytracer, a list of edges containing only the variables needed for calculating a ray's path, and the color of each vertex on the outer boundary of the mesh.

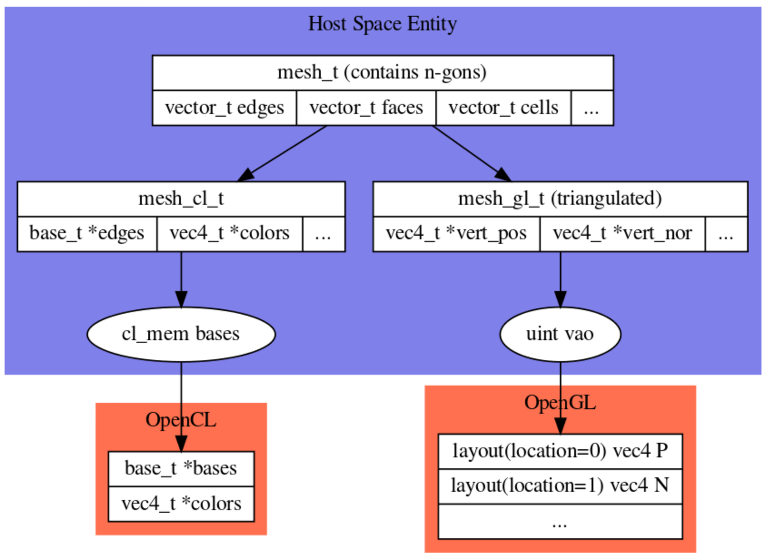


Figure 4.4: Memory occupancy overview on CPU and GPU sides. In blue is the host memory, in red is GPU device inputs.

4.2.3 Ray-Edge Intersection

A ray travelling on a triangle always crosses either a door-out edge or vertex. By calculating the distance at which a ray with position (u_0, v_0) and direction (u_Δ, v_Δ) intersects any of the edges' line equations, and checking which distance is the smallest, we know where the ray exits the triangle.

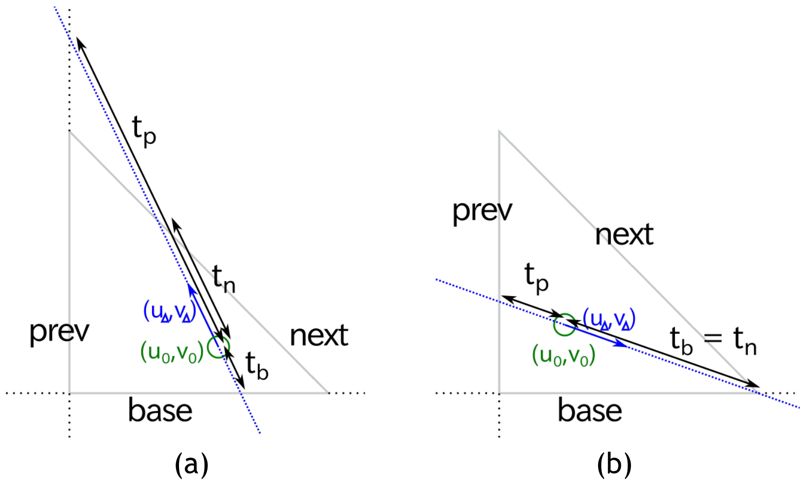


Figure 4.5: Determining the door-out edge of a ray laying on a triangle as the closest edge to point (u_0, v_0) in the direction of a ray passing: (a) through an edge; (b) through a vertex.

In barycentric coordinates, we know that the base's line equation is $v = 0$, the line equation of the base's previous edge is $u = 0$ and the equation of the base's next edge is $u + v = 1$. To calculate the distance from (u, v) to each of those edge (Figure 4.5(b)), we have the following system:

$$\begin{cases} v_0 + t_b.v_\Delta = 0 \\ u_0 + t_p.u_\Delta = 0 \\ u_0 + v_0 + t_n.(u_\Delta + v_\Delta) = 1 \end{cases} \tag{4.1}$$

or, equivalently,

$$\begin{cases} t_b = -\frac{v_0}{v_\Delta}, & \text{if } v_\Delta < 0 \\ t_p = -\frac{u_0}{u_\Delta}, & \text{if } u_\Delta < 0 \\ t_n = \frac{1-u_0-v_0}{u_\Delta+v_\Delta}, & \text{if } u_\Delta + v_\Delta > 0 \end{cases} \tag{4.2}$$

As shown in Figure 4.5(a), the ray intersects with:

- the base if t_b has the smallest positive value.
- the base's previous edge if t_p has the smallest positive value.
- the base's next edge if t_n has the smallest positive value.

Otherwise (see Figure 4.5(b)):

- the base's vertex if $t_b = t_p$.
- the base's next vertex if $t_b = t_n$.
- the base's previous vertex if $t_n = t_p$.

The distance at which the ray exits the triangle is the smallest positive $t_\bullet$.

A simplified code concerning ray-edge intersection is as follows:

```
float tv, ta, tu;
tv = (ray.d.v < 0.0f) ? -ray.p.v / ray.d.v : INFINITY;
tu = (ray.d.u < 0.0f) ? -ray.p.u / ray.d.u : INFINITY;
ta = (ray.d.u + ray.d.v > 0.0f) ?
    (1.0f - ray.p.u - ray.p.v) / (ray.d.u + ray.d.v) : INFINITY;

if(tv < tu && tv < ta)
    /* ray intersects the base */
else if (tu < tv && tu < ta)
    /* ray intersects the base->prev edge */
else if (ta < tv && ta < tu)
    /* ray intersects the base->next edge */
else
    /* ray intersects a vertex */
```

Note that if the ray crosses a door-out vertex (Figure 4.5(b)), meaning that the distance to two of the edges is equal and positive, we chose one of the edges incident on the vertex to proceed with propagation.

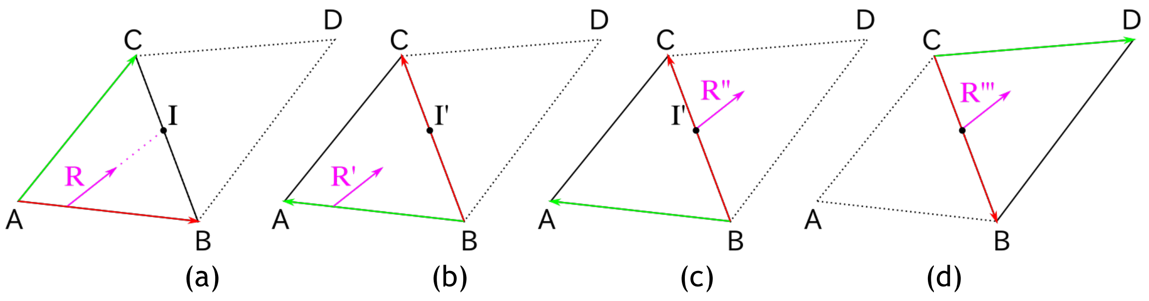


Figure 4.6: Red and green edges represent the u, v barycentric axes being used during propagation: (a) I is the point of intersection (as calculated in Section 4.2.3) between the door-out edge $\overline{CB}$ and the ray R on triangle $ABC_\triangle$ with $\overline{AB}$ as the base; (b) the frame is rotated to have $\overline{BC}$ as the base, while R' represents the updated coordinates relative to the new base; (c) the ray is moved to the point of intersection I', and is represented as R'' without changing direction; (d) the frame passage is performed by changing the base from the halfedge $\overline{BC}$ to its pair $\overline{CB}$ of triangle $CBD_\triangle$, and R''' is the ray (position and direction) relative to the new base.

4.2.4 Frame Propagation

When a ray crosses an edge from one triangle to another, the local coordinates of the ray have to be converted from the current coordinate system, defined by an edge of the current triangle, to one defined by one of the three possible edges of the next triangle. This is performed in two steps: *frame rotation* and *frame passage*.

The first step involves *rotating* the frame (or coordinate system) of the current triangle, which amounts to rotating the base of the current triangle; that is, converting from one coordinate system to another on the same triangle, as illustrated in Figure 4.6(a)-(b). A ray is composed of a position (u, v) and a direction $(u_\triangle, v_\triangle)$, each has to be updated when rotating the coordinate system.

```
vec2_t position_rotate_edge(base_t *base, int crossed_id, vec2_t p)
/* convert to another base of same the face */
{
    if(base->prev == crossed_id)
        return vec2(1.0f - p.u - p.v, p.u);
    if(base->next == crossed_id)
        return vec2(p.v, 1.0f - p.u - p.v);
    return p;
}
vec2_t direction_rotate_edge(base_t *base, int crossed_id, vec2_t d)
/* convert to another base of same the face */
{
    if(base->prev == crossed_id)
        return vec2(-d.u - d.v, d.u);
    if(base->next == crossed_id)
        return vec2(d.v, -d.u - d.v);
    return d;
}
```

Frame passage consists in moving the frame of the current triangle across its door-out

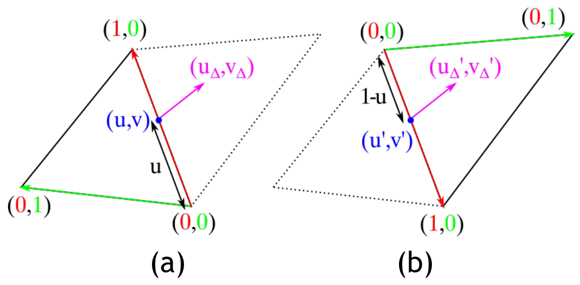


Figure 4.7: Updating position (u, v) and direction (u_Δ, v_Δ) to (u', v') and direction (u'_Δ, v'_Δ) during frame passage from a triangle to another across the door-out edge of the first (or door-in of the second). Because the ray is already positioned on the base, $v = 0$ and $v' = 0$.

edge to the corresponding adjacent triangle (Figure 4.6(c)-(d)); that is, the door-out edge of current triangle becomes the door-in edge of the next triangle. Therefore, this operation amounts to convert the frame base of the current triangle to frame base of the next triangle, as detailed in Figure 4.7. The ray position (u', v') in the new frame is given by $u' = 1 - u$ and $v' = v = 0$; the value of v' is 0 because the ray is positioned on the u-axis (frame base). Furthermore, since the new base is pair of the previous base (paired half-edges), and because the base of each frame works as the u-axis, and taking into consideration that both bases have length 1, we conclude that $u' = 1 - u$.

Concerning the computation of direction of the ray in the frame, we have to be aware that barycentric coordinate systems are not orthogonal, so it is not enough to simply negate such direction. Doing so, the resulting ray's direction would get skewed. To solve this problem, the direction (u_Δ, v_Δ) (barycentric coordinates) has to be converted into Cartesian coordinates of the current frame, which are converted into Cartesian coordinates of the next frame by negation, and then converted to barycentric coordinates. Note that in the process of frame passage, the position updating does not depend on the shape of both triangles, but the direction updating does.

```
vec2_t position_cross_edge(vec2_t p)
{
    return vec2(1.0f - p.u, 0);
}
vec2_t direction_cross_edge(base_t *crossed, base_t *next_base, vec2_t d)
{
    cartesian2D(crossed, &d);
    d = vec2(-d.u, -d.v);
    barycentric2D(next_base, &d);
    return d;
}
```

The frame passage from one triangle to the next one is then as follows:

```
ray_t ray_cross_edge(base_t *bases, base_t *crossed, ray_t ray)
{
    base_t *next_base = &bases[ crossed->pair ];
    ray.p = position_cross_edge(ray.p);
    ray.d = direction_cross_edge(crossed, next_base, ray.d);
    ray.base = crossed->pair;
    return ray;
}
```

4.3 Geodesic Rays across Tetrahedral Meshes in 4D Space

This section generalizes the ray tracing algorithm described in Section 4.2 to 4D, so that we can render images of 3D non-Euclidean spaces embedded in the 4D Euclidean space.

The method is similar and can be broadly described by the following steps:

- Start with a ray composed of a 3D point and 3D direction using barycentric coordinates relative to one tetrahedron of the space-mesh.
- Determine which face is intersected by the ray.
- Move the point to the intersection.
- Convert the point coordinates and direction to a local frame of the next tetrahedron.

As noted in the previous section, this method is agnostic to normals or any global coordinates, so we do not need to calculate any 4D normals for tetrahedra. Also, a tetrahedron has four barycentric axes, but only independent 3D barycentric coordinates, labeled u, v and w. Each axis starts at the same apex and is aligned with one of the edges incident on that point. Taking into account that a tetrahedron has four triangles, each with triangle defined by three vertices, we end up having twelve possible coordinate systems or frames. Therefore, a triangle is not enough to describe one particular coordinate system. To do that, in principle, we need to store one tetrahedron, one triangle and one edge. Fortunately, because we are using the half-edge data structure, we only use a half-edge to unambiguously specify the coordinate system, because the half-edge refers to a single half-face, which in turn refers to a single tetrahedron. Note that the number of half-edges of a tetrahedron is equal to the number of possible coordinate systems or frames, specifically twelve frames. Bear in mind that a ray may enter a tetrahedron coming from any triangle, and using any of the three possible coordinate systems of such triangle, so we cannot discard any of those twelve coordinate systems.

4.3.1 Coordinate conversions

Every point in the discretized 3D non-Euclidean space (i.e., space-mesh) can be uniquely represented by either:

- 3D local barycentric (u, v, w) coordinates and a *base*;
- 3D local Cartesian (x, y, z) coordinates and a *base*;

- 4D global Cartesian (x, y, z, w) coordinates of the 4D Euclidean space where the space-mesh is embedded.

Recall that 3D barycentric coordinates are needed when comes the time of computing geodesic paths or rays; 3D local Cartesians are instrumental in orthogonal operations (e.g., ray-line intersection); and 4D global Cartesians are important to determine, for example, edge lengths, and draw the space itself from the outside point of view.

To convert Cartesian coordinates to and from barycentric coordinates, we first calculate (d, g, h) values for all edges of each triangle of the tetrahedron, as illustrated in Fig. 4.2. In addition, we calculate (i, j, k) which are the Cartesian coordinates of the fourth corner of the tetrahedron that lays outside the base triangle relative to each of those edges which are used as a base for the twelve coordinate systems of the tetrahedron. Hence, we extended the structure *base_t* as follows:

```
typedef struct
{
    float d;        /* x      of the second vert of the base edge */
    float g, h;     /* x,y    of the third vert of the base triangle */
    float i, j, k; /* x,y,z of the fourth vert of the base tetrahedron */
    int next, prev, pair, cpair; /* (pair, cpair) are matching half-edges
        belonging to superimposed triangles of adjacent tetrahedra*/
} base_t;
```

Note that k is the height of the tetrahedron relative to the half-edge's triangle. Instead of including the same k on all three edges of that triangle, we could store a per-triangle structure and include an integer index of the triangle in *base_t*. However, this is not done in this way because it would spend more memory space since the index would occupy the same number of bytes as the k variable in each *base_t*. Besides, the execution would be slower because of the added pointer indirection, where instead of accessing k as $base \to k$ we would need to fetch it as $faces[base \to face] \to k$. The relationship (`pair`,`cpair`) between half-edges of adjacent tetrahedra is shown in Figure 3.2. This topological relationship is necessary to move from a tetrahedron to the next one; that is, to navigate through the space-mesh.

The following code snippets concern the conversion from barycentric to local Cartesian coordinates, and vice-versa, as well as from barycentric to global Cartesian coordinates, respectively:

```
void cartesian(base_t *base, vec3_t *b)
{
    b->x = base->d * b->u + base->g * b->v + b->w * base->i;
    b->y = base->h * b->v + b->w * base->j;
    b->z = b->w * base->k;
}
void barycentric(base_t *base, vec3_t *c)
{
    float hk = base->h * base->k;
    c->u = c->x / base->d - (c->y * base->g) / (base->d * base->h)
        - c->z * (base->h * base->i - base->g * base->j) / (base->d * hk);
    c->v = c->y / base->h - c->z * base->j / hk;
    c->w = c->z / base->k;
}

vec4_t barycentric_to_global(vec3_t pos, edge_t *base)
{
    vec4_t v0 = e_vert(base)->pos;
    vec4_t v1 = e_vert(e_next(base), mesh)->pos;
    vec4_t v2 = e_vert(e_prev(base), mesh)->pos;
    vec4_t v3 = e_vert(e_prev(e_pair(base)), mesh)->pos;
    vec4_t result = vec4_add(
            v0,
            vec4_scale( vec4_sub(v1, v0), pos.u),
            vec4_scale( vec4_sub(v2, v0), pos.v),
            vec4_scale( vec4_sub(v3, v0), pos.w));

    return result;
}
```

4.3.2 Ray-Triangle Intersection

A ray going through a tetrahedron always crosses either a face, an edge or a vertex. If no face is crossed, that is, only either a single vertex or an edge is on the way of the ray, one chooses any triangle incident on such vertex or edge to proceed, which simplifies the number of rules for propagation from fourteen (four triangles, six edges, and four vertices) to four because the intersection test will only return one out of four triangles.

Assuming that a ray with position (u, v, w) and direction $(u_\Delta, v_\Delta, w_\Delta)$ inside a tetrahedron, we need to calculate the distance from (u, v, w) to each triangle's plane in the direction $(u_\Delta, v_\Delta, w_\Delta)$, checking then which distance is smallest to decide which triangle the ray exits the tetrahedron from. This distance is calculated as follows:

$$\begin{cases} t_b = -\frac{w}{w_\Delta}, & \text{if } w_\Delta < 0 \\ t_{bp} = -\frac{v}{v_\Delta}, & \text{if } v_\Delta < 0 \\ t_{pp} = -\frac{u}{u_\Delta}, & \text{if } u_\Delta < 0 \\ t_{np} = -\frac{1-u-v-w}{u_\Delta+v_\Delta+w_\Delta}, & \text{if } u_\Delta + v_\Delta + w_\Delta > 0 \end{cases} \tag{4.3}$$

So, the ray intersects with:

- the $base \to face$ if t_b has the smallest positive value.
- the $base \to pair \to face$ edge if t_{bp} has the smallest positive value.

- the $base \to prev \to pair \to face$ if t_{pp} has the smallest positive value.
- the $base \to next \to pair \to face$ if t_{np} has the smallest positive value.
- otherwise, the ray intersects an edge or a vertex.

Note, if $w_\Delta >= 0$ it means the ray is either moving away or stationary relative to the base, meaning w_Δ has to be negative in order to intersect the base.

The distance at which the ray exits the triangle is the smallest $t_\bullet$. The following code snippet illustrates one inefficient way of implementing the intersection test:

```
float tbp, tnp, tpp, tb;
tb = (ray.d.w < 0.0f)  ? -ray.p.w / ray.d.w : INFINITY;
tbp = (ray.d.v < 0.0f) ? -ray.p.v / ray.d.v : INFINITY;
tpp = (ray.d.u < 0.0f) ? -ray.p.u / ray.d.u : INFINITY;
tnp = (ray.d.u + ray.d.v + ray.d.w > 0.0f) ? (1.0f - ray.p.u - ray.p.v - ray.p.
    w) / (ray.d.u + ray.d.v + ray.d.w) : INFINITY;

if (tb < tnp && tb < tpp && tb < tbp)
    /* ray intersects with base->face */
else if (tbp < tpp && tbp < tnp && tbp < tb)
    /* ray intersects with base->pair->face */
else if (tpp < tnp && tpp < tb && tpp < tbp)
    /* ray intersects with base->prev->pair->face */
else if (tnp < tb && tnp < tpp && tnp < tbp)
    /* ray intersects with base->next->pair->face */
else
    /* ray intersects an edge or vertex */
```

Since we always need a triangle to move to the next tetrahedron, and not an edge or vertex, we can decide which triangles will be chosen by simplifying the above conditions as follows:

```
if (tb <= tnp && tb < tpp && tb < tbp)
    /* ray intersects with base->face */
else if (tbp <= tpp && tbp < tnp)
    /* ray intersects with base->pair->face */
else if (tpp <= tnp)
    /* ray intersects with base->prev->pair->face */
else
    /* ray intersects with base->next->pair->face */
```

4.3.3 Frame Propagation

When a ray crosses a triangle from one tetrahedron to another, its local coordinates have to be converted from the coordinate system (or frame) of the current tetrahedron, defined by an half-edge of the current tetrahedron, to the coordinate system (or frame) of the next tetrahedron, defined by one of its twelve half-edges (see Fig.3.2).

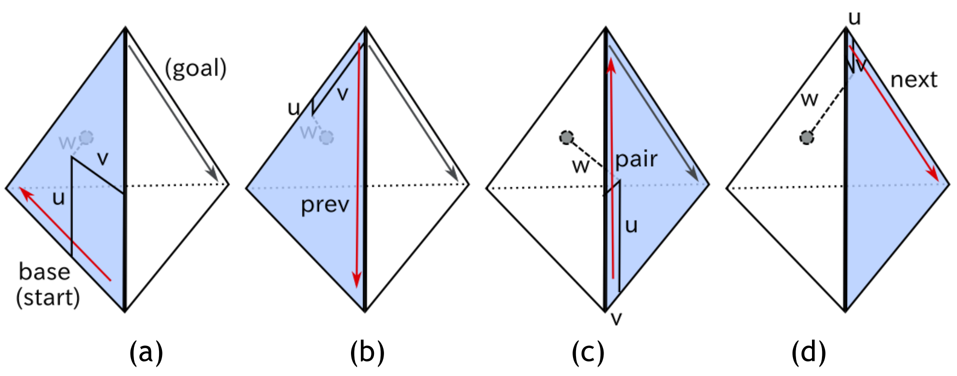


Figure 4.8: Frame rotations on the same tetrahedron applied to one of its interior points, starting on a *base* half-edge (in red) and ending on another half-edge labeled as *goal*; the changing base triangle appears in light blue: (a) $\to$ (b) by counterclockwise rotating the base to its *prev* half-edge, the *prev* becomes the base, though the base triangle stays the same; (b) $\to$ (c) by flipping the current base (or prev in (b)), we get the new base (pair in (c)), so that triangle base changes; (c) $\to$ (d) the final rotation is performed from the current base (in (c)) to its next half-edge (the new base in (d)), so now the point coordinates are relative to the *goal*'s coordinate system;

As before, the problem of *frame propagation* from one coordinate system to another can be subdivided into two steps: *frame rotation* and *frame passage*.

Considering that a ray may exit a tetrahedron through any of its four triangles, we need to convert its current frame to one out of the three frames of its door-out triangle that is intersected by the ray. This operation of changing the coordinate frame of a tetrahedron is called *frame rotation*, which affects both the barycentric coordinates of position and direction of each ray, though the global coordinates of the ray are the same (Figure 4.8). Therefore, the frame rotation involves the following operations:

```
vec3_t position_rotate(base_t *from_base, int to_base_id, vec3_t p)
/* convert to another base of same the tetrahedron */
{
    if(from_base->prev == to_base_id)
        return vec3(1.0f - p.u - p.v - p.w, p.u, p.w);
    if(from_base->next == to_base_id)
        return vec3(p.v, 1.0f - p.u - p.v - p.w, p.w);
    if(from_base->pair == to_base_id)
        return vec3(1.0f - p.w - p.u - p.v, p.w, p.v);
    return p;
}

vec3_t direction_rotate(base_t *base, int to_base_id, vec3_t d)
/* convert to another base of same the tetrahedron */
{
    if(from_base->prev == to_base_id)
       return vec3(-d.u - d.v - d.w, d.u, d.w);
    if(from_base->next == to_base_id)
       return vec3(d.v, -d.u - d.v - d.w, d.w);
    if(from_base->pair == to_base_id)
       return vec3(-d.w - d.u - d.v, d.w, d.v);
    return d;
}
```

Note that these two operations of *frame rotation* take place on the same tetrahedron. Moreover, in spite of using rotations to perform the *frame rotation*, no angles and trigonometric operations are required, as shown in the code snippets above.

On the other hand, *frame passage* consists in transforming the frame of the current tetrahedron across its door-out triangle into the frame of its adjacent tetrahedron; that is, the door-out triangle of current triangle is congruent with the door-in triangle of the next triangle. This operation reduces to convert the frame base (i.e., the `base` half-edge) of the current tetrahedron to frame base (i.e., the `cpair` half-edge) of the next tetrahedron, as illustrated in Fig. 3.2.

Similar to ray moving shown in Fig. 4.7, we assume now that the ray is already positioned at the door-out triangle. The frame passage of barycentric position (u, v, w) (door-out triangle) into (u', v', w') (door-in triangle) can be done in one step. In this case, $w' = 0$ because the distance to the base is zero, that is, it is already on the base triangle. Also, $v' = v$ because the barycentric distance of the point to u-axis is the same as the barycentric distance to u'-axis. Finally, $u' = 1 - u - v$ because u'-axis is symmetric to u-axis. This is encoded by the following snippet:

```
vec3_t position_passage(vec3_t p)
{
    return vec3(1.0f - p.u - p.v, p.v, 0.0f);
}
```

As before, because the barycentric coordinate system is not orthogonal, directions get skewed if the shape of both tetrahedra are not taken into account. To solve this problem, directions have to be converted to (local) Cartesian coordinates of the current tetrahedron, converted into (local) Cartesian coordinates of the next tetrahedron by negating u_Δ and w_Δ, then converted to the barycentric coordinates of the next tetrahedron. Note that v_Δ does not get inverted because the v-axis and v'-axis point in the same barycentric direction (away from the base half-edge). This direction passage process is encoded as follows:

```
vec3_t direction_passage(base_t *crossed, base_t *cpair, vec3_t d)
{
    cartesian(crossed, &d);
    d = vec3( -d.u, d.v, -d.w );
    barycentric(cpair, &d);
    return d;
}
```

Since the current base face represents $w = 0$ and the ray reached it, w_Δ is negative, once it crosses into the new coordinate system, it will be pointing away from the base triangle, not toward it, that is, $w'_\Delta = -w_\Delta$. Once again, recall that while position passage does not depend on the shape of the triangles, direction passage depends on the shape of both triangles.

Summing up, the ray passage from one tetrahedron to the next one can then described as follows:

```
ray_t ray_passage(base_t *base, ray_t ray)
{
    ray.p = position_cross_face(ray.p);
    ray.d = direction_cross_face(base, cpair(base), ray.d);
    ray.base = base->cpair;
    return ray;
}
```

4.4 Tracing a single path

Now that we know how to propagate a point along a direction on the interior of a mesh. We can draw a geodesic by propagating a ray on the mesh until reaching a boundary of the mesh (i.e., when a door-out simplex has no pair connected to it). For example, in Fig. 4.9, the boundary of the 2D wormhole-like space consists of two discretized circles (i.e., 1D spheres) made of 1D simplices; also, in Fig. 4.10, the boundary of the 3D wormhole-like space consists of two discretized 2D spheres made of triangles.

If we want to draw any geodesic path, it suffices to calculate the global Cartesian coordinates of each intersection point on door-out simplices and appending such point to a line mesh, as described by the following code snippet:

```
ray_t trace_path(base_t *bases, ray_t ray, mesh_t *line, mesh_t *mesh)
{
    int i;
    int e = mesh_add_vert(line, barycentric_to_global(ray.pos, mesh));
    mesh_add_edge(line, e, -1);
    for(i = 0; i < MAX_ITERATIONS; i++)
    {
        if(i == 0)
            ray = intersect(bases, ray);
        else
            ray = intersect_from_border(bases, ray);
        mesh_append_edge(line, barycentric_to_global(ray.pos, mesh));
        base_t *crossed = &bases[ray.base];
        if(crossed->pair < 0) break; /* end of mesh */
        /* coordinates relative to the new base */
        ray = ray_passage(bases, crossed, ray);
    }
    return ray;
}
```

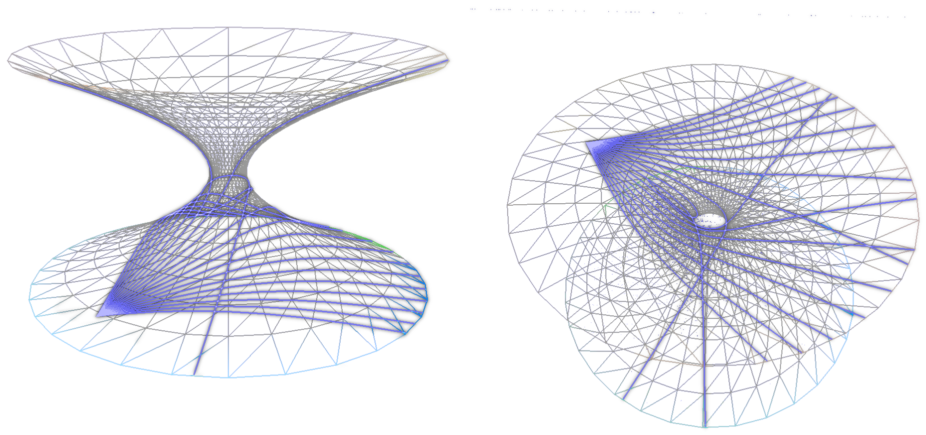


Figure 4.9: Two traces from different camera positions and angles on a 2D wormhole-like space.

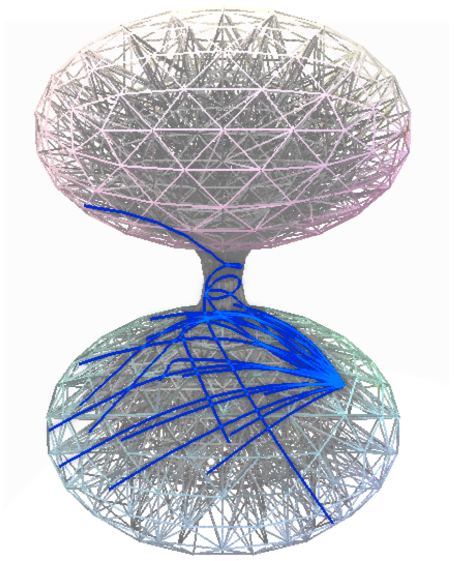


Figure 4.10: A trace on a 3D wormhole-like space. Although hard to see, the blue rays are traveling inside the volume composed of tetrahedra.

The function *intersect* calculates the door-out simplex the ray intersects, and returns the updated ray at the intersection point relative to one of the coordinate systems of the door-out simplex. This function is only used in the first iteration. For subsequent iterations, we use the function *intersect_from_border*. While the function *intersect* checks for an intersection on all door-out simplices, the function *intersect_from_border* does not check for an intersection with the door-in simplex because, after the first iteration, the ray will be moving away from such simplex.

The *trace_path* function only traces a single geodesic ray. To simulate a camera's view, we need to emit many rays from the camera's position and with a direction within its field of view, as seen in Fig. 4.9.

4.5 Geodesic Ray Tracing

In the previous section, we show how to trace a path along a geodesic ray. These geodesic rays in non-Euclidean spaces are homologous to straight rays in conventional ray tracing in Euclidean spaces. In either case, Euclidean or non-Euclidean, the ray travels in a straight manner, i.e., it does not bend on any direction inside the space. Let us now describe how the geodesic ray tracer works.

As seen in the previous section, the rays stop on the space boundary, i.e., when they hit the space boundary. Here, we assume backward ray tracing, so the rays travel from the camera to the light source. However, in a non-Euclidean space, it does not suffice to find geodesic path from the hit (boundary) point to the light because there may be several geodesic rays to the light, each contributing to the illumination of the hit point. Because our main purpose was to see how a non-Euclidean space looks like, we opt by automatically assigning a unique color to each boundary vertex representing its global position.

A camera has an horizontal field of view FoV, a position P in local barycentric coordinates relative to half-edge base B, and an orientation. This orientation is defined by two vectors F (front) and S (sideways) which point respectively in the camera's forward and sideways direction in local Cartesian coordinates. As illustrated in pseudocode of the ray caster below, these vectors are important to not only calculate the up vector of the camera, but more importantly to define the direction of emission of rays.

In practice, the ray tracer works as usual by emitting rays from the camera position within a frustum. The difference is that the rays are geodesic. For that purpose, the ray tracer calls the function `trace`, which is similar to the `trace_path` described in the previous section. However, instead of drawing a line, it only returns the ray's final hit point.

Given a grid of W by H pixels, for each pixel, we calculate its $RGBA$ color by the following method:

```
if $W > H$ then
    $ratio_x \leftarrow tan(FoV) * 2 * (W/H)$
    $ratio_y \leftarrow tan(FoV) * 2$
else
    $ratio_x \leftarrow tan(FoV) * 2$
    $ratio_y \leftarrow tan(FoV) * 2 * (H/W)$
end if
$up \leftarrow F \times S$
for all $(x, y) \in pixels$ do
    $D_{side} \leftarrow S * (x/W - 0.5) * ratio_x$
    $D_{up} \leftarrow up * (y/H - 0.5) * ratio_y$
    $D_{front} \leftarrow F * 1$
    $D \leftarrow D_{side} + D_{up} + D_{front}$
    $D \leftarrow barycentric(B, D)$
    $(base, U, V, W) \leftarrow trace(D, B, P)$
    $(r, g, b, a) \leftarrow base.color * (1 - U - V) + base.next.color * U + base.prev.color * V$
end for
```

This tracer allows for real-time rendering and interaction. The rationale behind this efficiency is twofold. Firstly, the ray traversal is quite efficient using barycentric coordinates. Secondly, we use GPU multi-threading to process each ray independently of other rays. Taking into consideration that the algorithm discretizes the space into a mesh, we can thus edit such mesh in real time. For example, in Fig. 4.11, we can observe the results of interactively changing the height of the extrusion of a wormhole-like space; this amounts to stretching the space in the fourth dimension. In Figs. 4.12-4.13, we see several ray-tracing images of a cubic wormhole.

As a post-processing step, because each color on the output of the raytracing step describes an unique position, we can use it as the coordinates on a texture lookup. Texturing is illustrated in Fig. 4.15, where the four channels of the color represent the 4D global coordinates of the end of each ray; these coordinates are then transformed into spherical coordinates to be used to sample from two sphere-map textures, one is the milky-way, the second is the planet earth.

4.6 Objects embedded in non-Euclidean spaces

So far, the only visible elements are the boundary faces of the space. We have not considered objects embedded in the space. Let us now see how to render such objects.

4.6.1 Space-Mesh Aligned Objects

A space-mesh aligned object is described by triangles already existent in the space-mesh's geometry. In this case, it is trivial to render it, since we know when triangles of the space-mesh are intersected by the rays. This requires to identify the building blocks of the mesh that fit the object. However, we cannot move the object because that would require deforming the space-mesh itself. That is, there is no independence between the space the objects within such space.

4.6.2 Single-Cell Objects

Since the space within a single cell is Euclidean, any object contained within a single cell can be rendered by Euclidean ray-tracing techniques. This means we can render triangle meshes as well as implicitly-defined objects (e.g. spheres, planes, boxes). However,

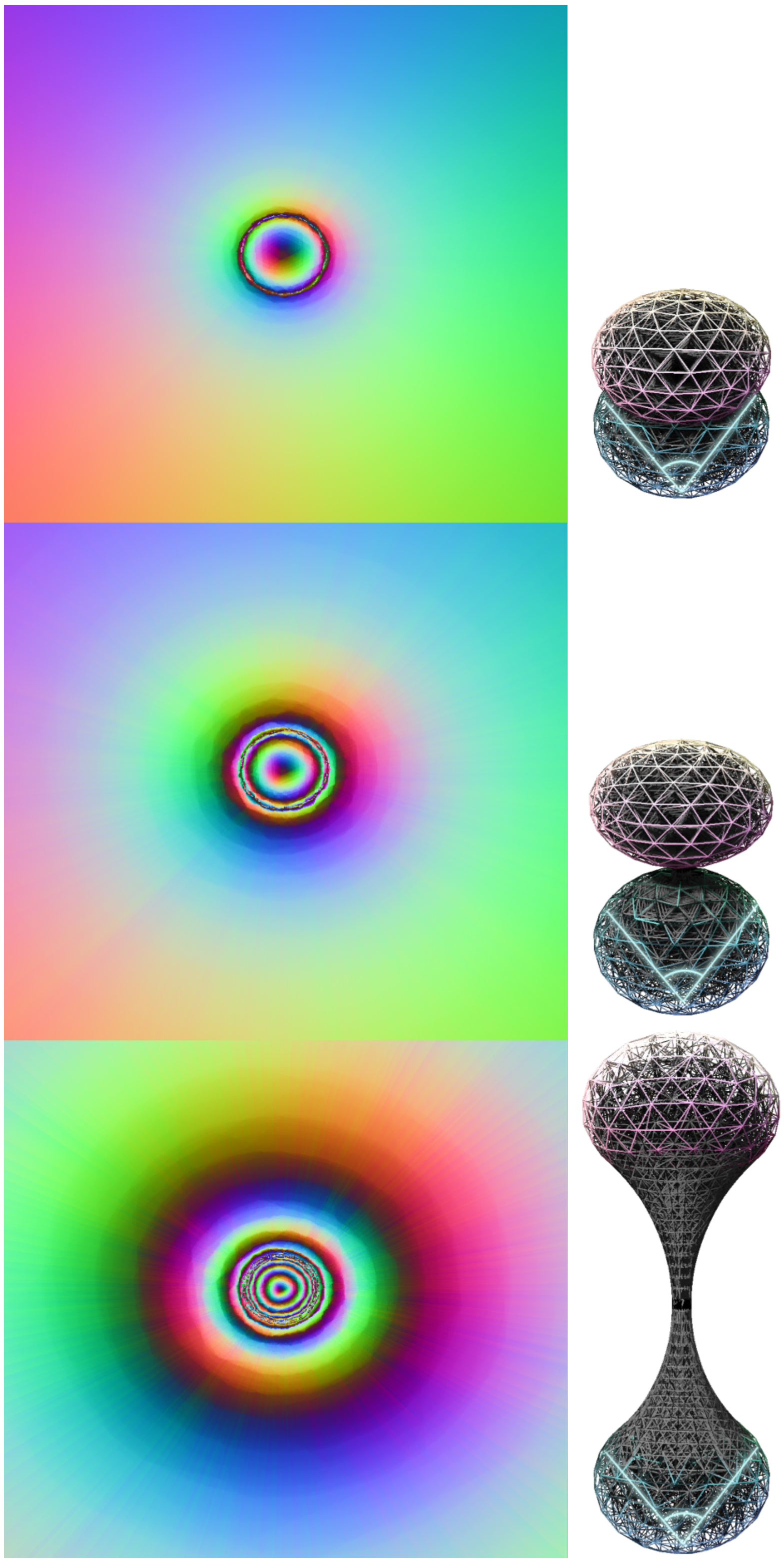


Figure 4.11: A wormhole-like space with varying extrusion sizes. The longer the wormhole, the more rings appear; each repeated color in the rings represent the same position but different paths to such position. On the left-hand side, we see ray-traced images (internal view) of the space; on the right-hand side, we observe the mesh in which the rays are traced, which corresponds to the external view of the space. The cyan lines indicate the camera's position and field of view.

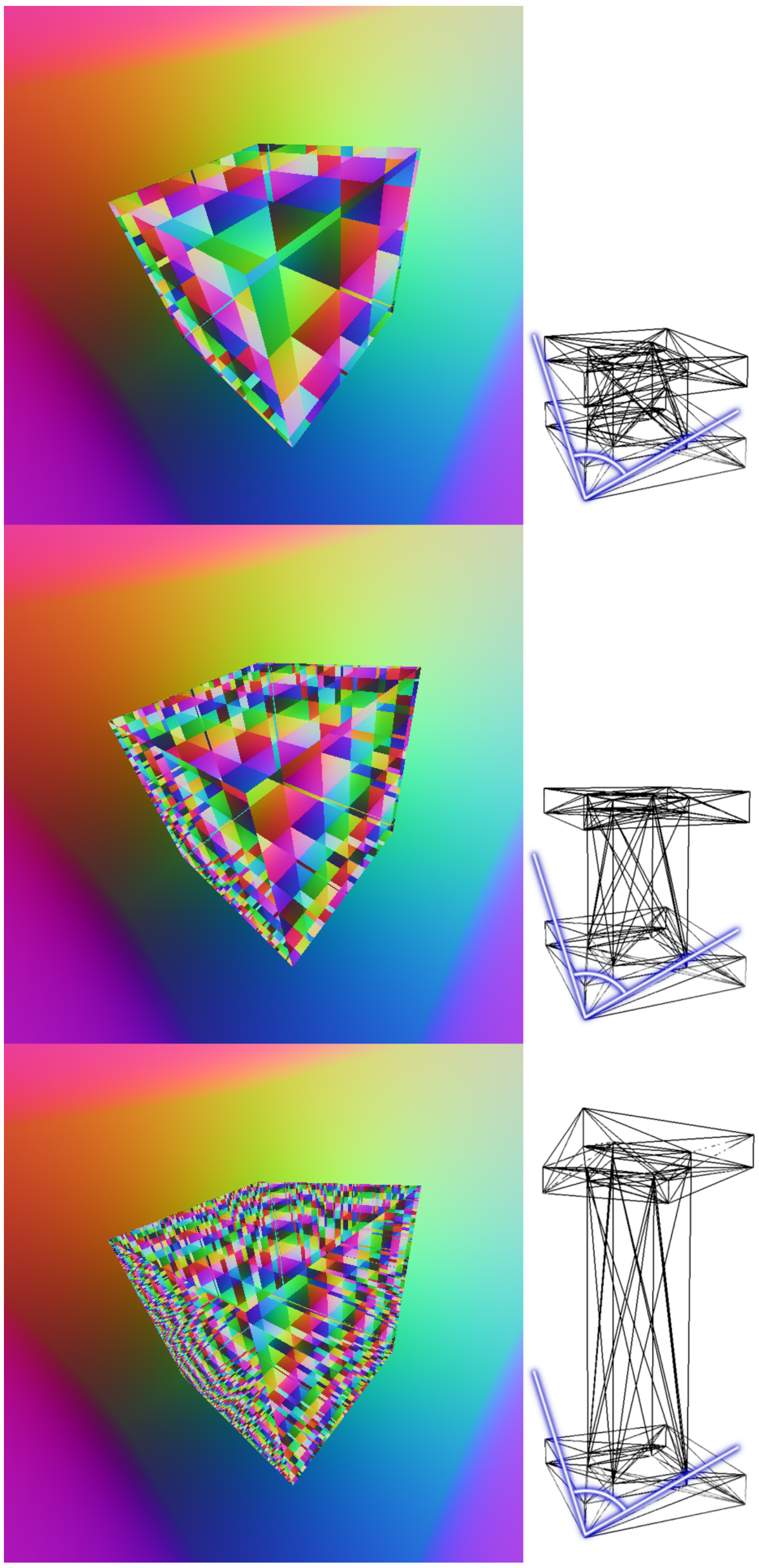


Figure 4.12: A cubic wormhole, where instead of extruding a sphere, we extrude a cube; the center tunnel is a tesseract (i.e., hypercube) without two caps. Also, each image represents a different size in the extrusion step as seen from outside the tunnel, but within the space.

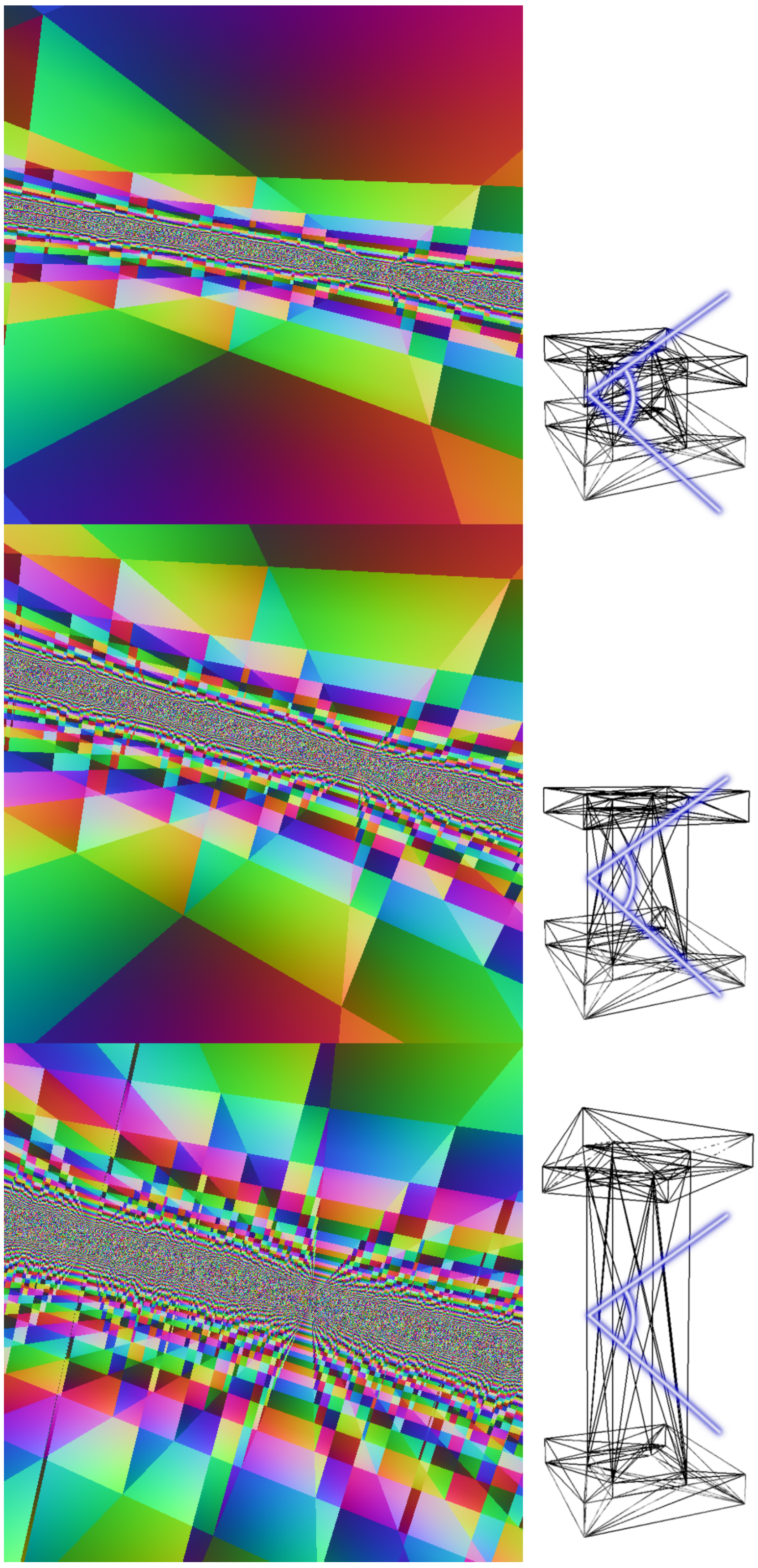


Figure 4.13: Cubic wormhole viewed from inside the tunnel. Because the tesseract's cubes are connected, the light loops through them, giving an impression that the tunnel is infinite.

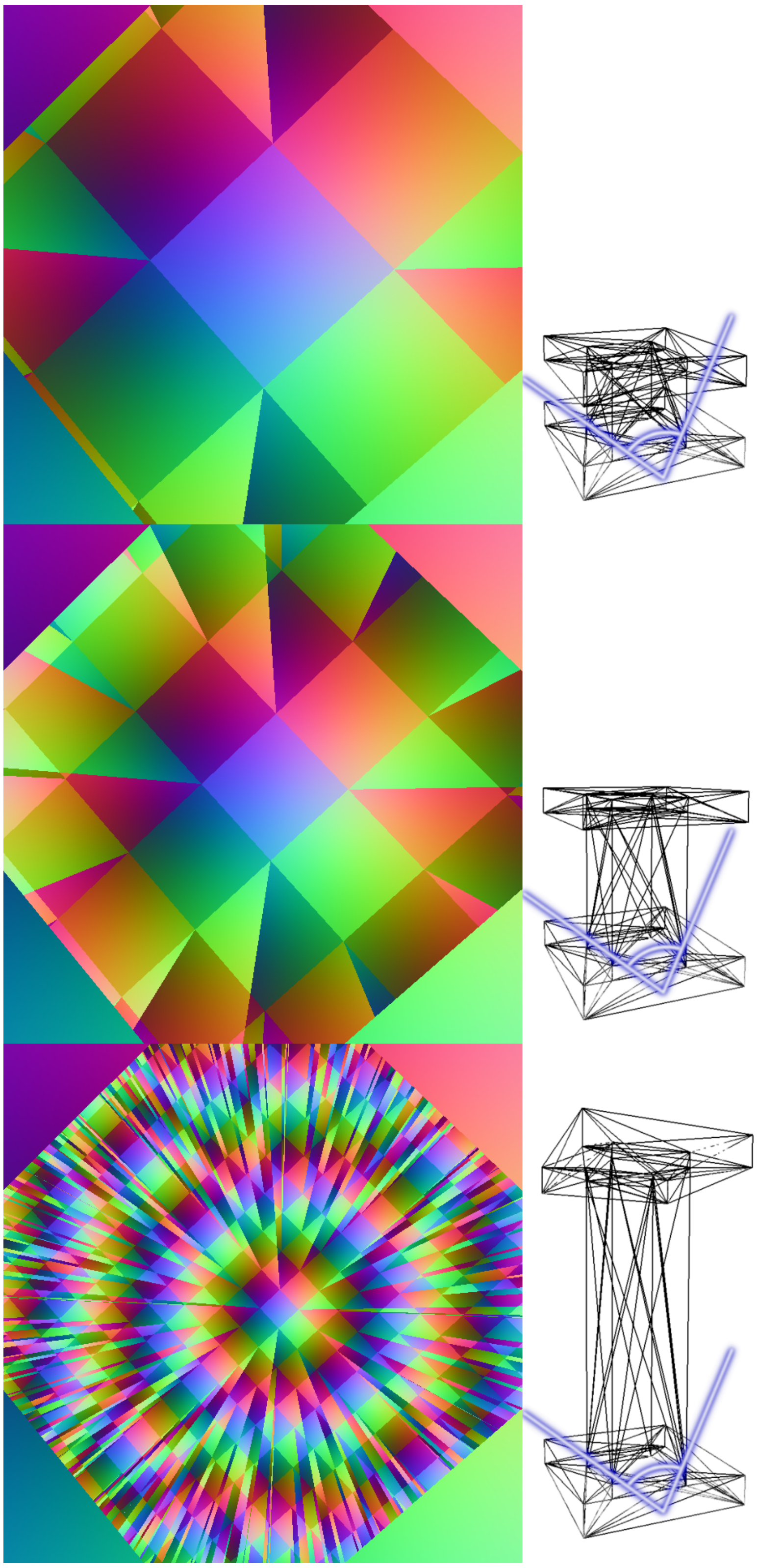


Figure 4.14: As the tunnel gets deeper, rings start to be noticeable similar to spherical wormhole.

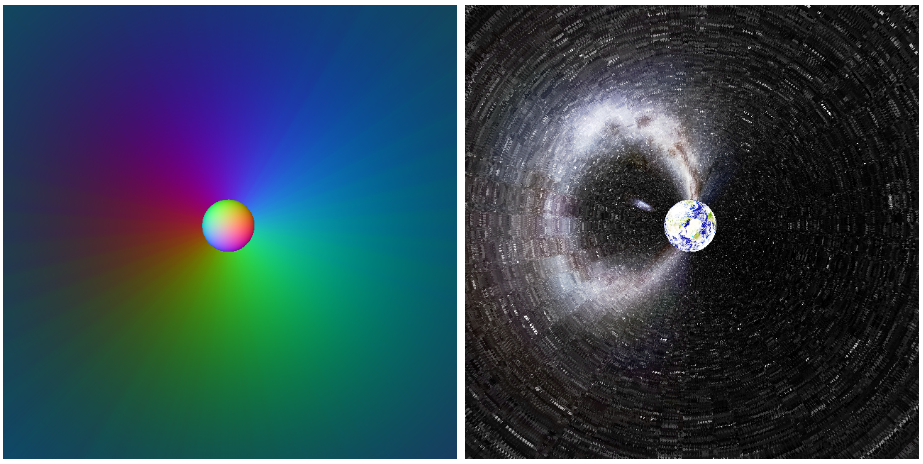


Figure 4.15: Raytracing on a hyper-conical space. On the left, raytracing step, on the right, post processing step.

because each cell has twelve possible coordinate systems, when creating an object within a cell, its coordinates must be converted and registered to all possible coordinate systems of that cell. Recall that we can convert from barycentric to Cartesian and vice-versa.

4.6.3 Multi-Cell Objects

Because there may be no global coordinate system inside a Non-Euclidean space, there is no single method for positioning vertices of a mesh on that space in a way in which all properties of the object hold (e.g. volume, edge lengths, face areas). Because of this, rigid, nondeformable meshes may not be directly positioned on arbitrary non-Euclidean space. To understand the previous assertion, let us imagine a deformable body moving in a non-Euclidean space. The soft body deforms in conformity with the local Gaussian curvature of the space. On the contrary, imagine trying to position a rigid body on any non-Euclidean space, the body would need to already have the same curvature as the space, and it could not freely move to different positions on that space with differing curvature.

A way of approximating a solution is with soft-body physics. A soft-body spring-damper system allows us to try to get as close to the original properties of the object in a weighted manner, by applying a force to correct edges which are longer or shorter than they should, and a force to try to maintain the same volume the original mesh had in Euclidean space.

A spring-damper system relies on calculating the distance between vertices, which is not trivial in non-Euclidean space since there may be several straight lines (geodesics) between two points. Consequently, one needs to compute the geodesics between every two connected vertices.

To trace multi-cell objects embedded in the space-meshes we need to be able to perform the following tasks:

- To trace a point inside a cell.
- To a straight line inside a cell.
- To trace a polygon inside a cell; with this task, we can already trace an entire mesh as long as it does not overflow the cell.
- To calculate geodesics between any two points spanning multiple cells. This may involve path-finding or string pulling. Moreover, this task allows us to visualize distorted the wireframe of a given object.
- To calculate all single-cell polygons of the object that fill the empty space between the three geodesics (edges) that delimit triangles of such object.
- Taking into account the shape deformations in non-Euclidean space (i.e., a straight triangle may look curved in non-Euclidean space), we need to a spring-damper system (including motion integration) that applies forces that satisfy physical constraints and approximate the shape of the object in Euclidean space.

4.7 Navigating across space-meshes

Navigation across space-meshes is performed by moving the camera (or observer), as usual in 3D graphics. To allow interactive and smooth positioning of the camera in the tetrahedral space-mesh, its 3D barycentric velocity V is stored.

On each graphics frame inside an update loop, if the user is pressing any directional key, the velocity is incremented by an acceleration in the direction programmed for the key (e.g., wasd combination of keyboard keys). To keep interaction under control, the velocity is multiplied by a dampening factor to prevent the camera from slipping indefinitely.

Although not physically accurate, we calculate the new position of the camera $P' \leftarrow P + P_\Delta$ with $P_\Delta \leftarrow V$ (Fig.4.16). If P' is inside the coordinate system of the current tetrahedron $(P'_u \geq 0 \wedge P'_u \leq 1 \wedge P'_v \geq 0 \wedge P'_v \leq 1 \wedge P'_u + P'_v \leq 1)$ we set $P \leftarrow P'$; otherwise, we calculate the vector I so that $P + I$ is the position where the ray defined by position P and direction V intersects the boundary of the current tetrahedron. We then move the camera to the start of the new tetrahedron, updating its position, velocity, and direction vectors to the new coordinate frame. We then repeat the process but for $P_\Delta \leftarrow (P_\Delta - I)$ to account for the velocity not yet added to the camera's position.

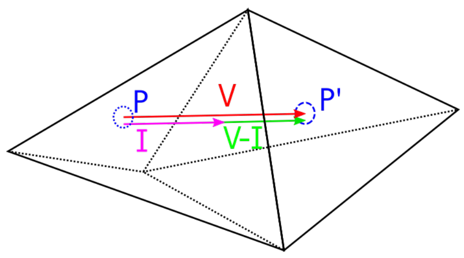


Figure 4.16: Steps in moving a point P by velocity V if there is a cell crossing in the path.

4.8 Summary

This chapter describes the geodesic ray tracing of non-Euclidean spaces and eventual objects inside them. As usual in ray tracing, a bunch of rays are triggered from the observer's position (i.e., camera), with the difference the rays are geodesic rays. When compared to the Euclidean ray tracing, we see that geodesic ray tracing presents the same difficulties in finding the hit points on the boundaries of space and objects. Nevertheless, the grand difficulty was to generalize 2-dimensional to 3-dimensional non-Euclidean.

Chapter 5

Conclusions and Future Work

This dissertation focuses on the modeling and visualization of non-Euclidean spaces through ray tracing. This chapter presents the main conclusions resulting from the research work described above. Also, some hints are put forward for future work.

5.1 Results and Conclusions

As seen in Chapter 2, the literature is scarce in terms of rendering techniques for non-Euclidean spaces. These techniques are mostly based on rasterization, and only apply to specific spaces like, for example, the hyperbolic space. That is, rendering a specific space requires a specific solution.

On the contrary, the main contribution of this dissertation is a general ray tracer for non-Euclidean spaces, no matter if they curved on not. What makes this ray tracer general is that it uses simplicial building blocks. These simplices allow for the interactive modeling of the space without the need for a specific formula or algorithm for testing how the space looks like. The resulting tool (ray tracer and tetrahedral modeler) is especially useful for physicists and mathematicians that may have imagined the shape of an arbitrary space but did not have the means to visualize it. The tool may also be used to by students and teachers, to give a better and intuitive understanding of geometry.

So, recalling the research hypothesis:

> *Is that feasible to come up with a* general *renderer for non-Euclidean spaces?*

we have to conclude that the main goal of the research has been attained. In the context general non-Euclidean spaces, it is not feasible to have a one-to-map between a non-Euclidean space and the 2D screen space. This is equivalent to say that we cannot rasterize general curved spaces. To circumvent this difficulty, we need to take advantage of ray tracing. This fact end up being the main conclusion of the present dissertation.

5.2 Future Work

Looking at the work we have done, we can envision the following improvements:

- To include more modeling operations; for example, automatic tetrahedrization of the space between two surfaces.
- To visualize the effects of non-Euclidean space on multi-cell objects.
- Optimize real-time modelling further; for example, by migrating the calculation of the per-base variables (d, h, g, i, j, k) —used to convert from barycentric to local Cartesian coordinates— to the GPU.
- Calculate smooth geodesics. Much like rasterizers use smooth interpolated normals on triangle meshes to make the geometry seem smoother, perhaps it is possible to smooth out the geodesics, making the resulting space look more curved without the need for further subdivisions.
- Calculate the global illumination on non-Euclidean space using path-tracing.
- Simulate sound deformations in non-Euclidean space such as, for example, echo and reverberations.

Bibliography

[CNS+11] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. Interactive indirect illumination using voxel cone tracing. *Computer Graphics Forum*, 30(7):1921–1930, September 2011. 6

[CSK+17] Alessandro Dal Corso, Marco Salvi, Craig Kolb, Jeppe Revall Frisvad, Aaron Lefohn, and David Luebke. Interactive stable ray tracing. In *Proceedings of High Performance Graphics* (HPG'17), pages 1:1–1:10, New York, NY, USA, 2017. ACM Press. 6

[DLVC99] Julien Dompierre, Paul Labbé, Marie-Gabrielle Vallet, and Ricardo Camarero. How to subdivide pyramids, prisms and hexahedra into tetrahedra. Technical Report CERCA R99–78, Centre de Recherche en Calcul Appliqué (CERCA), October 1999. 19, 20

[GM91] Charlie Gunn and Delle Maxwell. Not knot, 1991. 8

[Gun93] Charlie Gunn. Discrete groups and visualization of three-dimensional manifolds. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques* SIGGRAPH'93, pages 255–262, New York, NY, USA, 1993. ACM Press. 8

[HHMS17a] Vi Hart, Andrea Hawksley, Elisabetta A. Matsumoto, and Henry Segerman. Non-euclidean virtual reality i: explorations of $\mathbb{H}^3$, February 2017. 8

[HHMS17b] Vi Hart, Andrea Hawksley, Elisabetta A. Matsumoto, and Henry Segerman. Non-euclidean virtual reality ii: explorations of $\mathbb{H}^2 \times \mathbb{E}$, February 2017. 8

[HvDF+14] John F. Hughes, Andries van Dam, James D. Foley, Morgan McGuire, Steven K. Feiner, David F. Sklar, and Kurt Akeley. *Computer Graphics: Principles and Practice*. Addison-Wesley, 2014. 5

[JvTFT15a] Oliver James, Eugénie von Tunzelmann, Paul Franklin, and Kip Thorne. Gravitational lensing by spinning black holes in astrophysics, and in the movie Interstellar. *Classical and Quantum Gravity*, 32(6):065001:1–41, February 2015. 9

[JvTFT15b] Oliver James, Eugénie von Tunzelmann, Paul Franklin, and Kip Thorne. Visualizing Interstellar's wormhole. *American Journal of Physics*, 83(6):486-499, June 2015. 9

[Kaj86] James T. Kajiya. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques* (SIGGRAPH'86), Dallas, Texas, pages 143–150, New York, NY, USA, August 1986. ACM Press. 6

[Laf96] Eric Lafortune. *Mathematical Models and Monte Carlo Algorithms for Physically Based Rendering*. PhD thesis, Department of Computer Science, Faculty of Engineering Katholieke, Universiteit Leuven, February 1996. 6

[PG92] Mark Phillips and Charlie Gunn. Visualizing hyperbolic space: Unusual uses of 4x4 matrices. In *Proceedings of the 1992 Symposium on Interactive 3D Graphics* I3D'92, pages 209–214, New York, NY, USA, 1992. ACM Press. 8

[RdSCP17] Nuno T. Reis, Vasco Alexandre da Silva Costa, and João António Madeiras Pereira. Coherent ray-space hierarchy via ray hashing and sorting. In *Proceedings of the 12th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications*. INSTICC Press, January 2017. 6

[Wee02] Jeff Weeks. Real-time rendering in curved spaces. *IEEE Computer Graphics and Applications*, 22(6):90–99, November/December 2002. 8