Deforming Space:

Creation of a Non-Euclidean Render Engine

Jose L. Redondo Tello

Image Processing and Multimedia Technology Centre, Polytechnic University of Catalonia

Bachelor’s degree in Video Game Design and Development

Mr. Marc Garrigó

June 24, 2021

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Centre de la Imatge i la Tecnologia Multimèdia

Introduction

Abstract

In this document I will explain the development of a real-time non-Euclidean render engine, which will use the ray tracing rendering technique in order to represent deformations of the virtual space in the scene through the use of aberrations.

This project was developed in Visual Studio, using a self-made 3D render engine powered by OpenGL 4.3, and written entirely in C++.

Due to the complexity of the topic and time constrains, this will be a technical demo, focused on showcasing the interesting space/level/puzzle design options that afford non-Euclidean spaces, created and displayed using the scene edition functionality of the rendering engine.

Github release:

Key words

Non-Euclidean geometry, ray tracing, ray casting, ray path tracing, real-time, C++, OpenGL, GLSL, scene editor

Preface

Motivation

Graphics programming is a fascinating subject which I have been studying on my own for the last 2 years, but it wasn’t until I stumbled upon a YouTube video (Non-Euclidean Worlds Engine, 2018) that I became curious about non-Euclidean geometry.

Compressing the space needed to play a game has been a huge design constrain in VR games. To explore big extensions of terrain, a movement mechanic needs to be implemented with the player’s movement in the real world, these mechanics break the immersion and tend to cause motion sickness. Non-Euclidian spaces can be used to cramp an infinite number of rooms in small spaces via portals and space deformation, thus reducing the weight or even removing the space limitation completely, allowing the creation of more immersive experiences.

The main motivation is to create a render engine capable of rendering non-Euclidean spaces using ray tracing, allowing me to see and interact with aberrations in real time.

Goals

The main objective of this paper is to develop a rendering engine using C++ and OpenGL, which will be able to render simple scenes in real time using a ray tracing algorithm.

This rendering engine will allow the user to create aberrations, which deform the way the scene is perceived and traversed, either by compressing or expanding the space in the directions the user introduces to the aberration trough the editor.

- Learn and use OpenGL compute shaders.
- Implement ray tracing algorithm.
- Optimize the algorithm and the process to send scene data to GPU.
- Create an aberration component and implement a method to visualize it in engine.
- Deform the rays using aberrations that will deform what the user sees.
- Make aberrations interact with the camera movement.
- Make the aberrations able to change not only the direction of the rays, but also their position (portals).
- Create a demo to showcase and explore its possibilities.
- Document the development process for future researches.
- Showcase the new possibilities and advantages of using non-Euclidean spaces.

Problem description

The use of non-Euclidean spaces in video games, despite not being new, hasn’t seen a big progression outside of some notable indie games (*Superliminal*) that decided to go a step beyond simple portals that seamlessly teleport the player to a different location.

This can be attributed to the lack of resources and recorded projects that accomplished other forms of non-Euclidean spaces and the lack of knowledge about the possibilities enabled by them.

One example of a video game sector that would benefit from using non-Euclidean spaces is VR games, where the fact that the user is in a closed space obliges the developers to add level traversal mechanics (teleport, movement through joystick…) that may break the immersion and cause motion sickness. Non-Euclidean spaces make possible to compress an infinite amount of space into a single room.

Showcasing the design space that non-Euclidean spaces create and give the tools to developers to test and experiment about the effects and experiences that can be archived may help expand and diversify their uses.

Scope

The objective is to create a demo which will showcase scenes with different non-Euclidean spaces, showcasing some design possibilities and the interactions with these spaces. The demo will not have the ability

Note that the goal is to create a rendering engine, not a game engine, which means that features such as scripting / gameplay features, nor will it have the ability to render complex scenes with hundreds of objects with thousands of polygons.

It will contain basic level editor tools to add, import, rotate, move and scale meshes, and the capability to create, rotate, scale and move aberrations, whose effects on the space will be defined by parameters introduced by the user.

These aberrations will change the direction and the position of the rays thrown by the camera and will affect its movement, thus effectively deforming the space of the scene. Spherical and hyperbolic geometry will not be supported (different types of non-Euclidean geometry), since they require a different implementation and treatment.

Table of contents

List of figures

Glossary

Dictionary

GPU intensive: Task / program that uses a lot of GPU processing power during an extensive amount of time.

Algorithm: An ordered set of instructions recursively applied to transform data input into processed data output, as a mathematical solution, descriptive statistics, internet search engine result…

Real-time / online render engine: Engine that renders the scene at a refresh rate high enough to create the illusion of motion (30 FPS and upwards).

Offline render engine: Engine focused on the quality of the render, can take hours or days to render one frame of a scene (depending on the scene and effects complexity).

To render: To interpret a scene / geometry / object and translate that interpretation to a visual 2D image.

Ray tracing / path tracing / ray casting: Rendering technique based on throwing rays for each pixel of a camera, and painting it with the color of the element the ray intersected with.

Mesh: Representation of a geometric object as a set of finite elements.

High-poly mesh: There is no set number that defines a high-poly mesh, in our case, we will consider a high-poly mesh any mesh with more than 50.000 vertices.

To debug: To stop the execution of the code at certain points, to visualize the state of the variables and how the code operates step – by step. Technique used in programming to fix and locate errors.

Hardcoded variable: A hardcoded variable is one that cannot be changed in the middle of execution, directly written in the code.

Diffuse light: Simulates the directional impact a light object has on an object. This is the most visually significant component of the lighting model. The more a part of an object faces the light source, the brighter it becomes (Vries, 2014).

Aliasing: Effect in which the pixel formations in a computer rendered image become visible.

Acronyms

GPU: Graphics Processing Unit, specialized electronic circuit designed to accelerate the creation of images.

CPU: Central Processing Unit, electronic circuitry that executes instructions comprising a computer program.

ND: N-Dimensional

(a) 2D Two-Dimensional
(b) 3D Three-Dimensional

GPGPU programming: General-Purpose Graphics Processing Unit programming, programming paradigm focused on the possible uses of GPU's outside of rendering (AI, mathematic computations…).

VR: Virtual Reality.

AABB: Axis-Aligned Bounding Box.

FoV: Field of View.

FPS: Frames Per Second.

Project management

Due to this project's high technical difficulty, the planning was understood as a set of deadlines that, in case of not being accomplished, would delay the next implementation, probably causing the need to cut down features.

Planning

September 2021

Month dedicated to investigate and understand not only the contents but also the needs of the project, their scale, the different implementations and the problems and consequences of each of them.

With all the sources gathered, at the end of the month a reunion with the TFG supervisor was held in order to assess the order and importance of the features that must be implemented, the optional ones, and the discarded features, and also inform of the implementations of each one and their respective deadline.

October 2021

Start to implement ray tracing, the first part of the development and the longest one, expected to be completed by the end of February 2022, it was estimated that 4 moths were needed to implement it.

- Set up a compute shader.
- Draw first geometrical shape (sphere) using raytracing algorithm.

November 2021

Start to make the engine "usable", focused on testing that the raytracing algorithm, compute shader and geometric math are set up correctly.

- Ray trace a triangle.
- Set up perspective camera.
- Enable automatic FoV adjustments.

December 2021 – January 2022

The biggest and most important feature of the project is to render a mesh using ray tracing. It is expected to cause a lot of problems due to the need to send a lot of data from the CPU to the GPU. Careful testing will be needed in order to minimize the risk of bugs.

- Send geometry data to GPU through a texture.
- Paint a simple triangle.
- Paint complex meshes.
- Apply transform changes (translate, rotate, scale) form the inspector automatically.

February 2022

Period dedicated to clean the code and improve it, with the intention to optimize it since the next features are expected to be expensive. Low priority tasks related to ray tracing rendering will be implemented during this month, the ones that are not finished will be discarded from the prototype.

- AABB boxes implementation.
- Fast ray/triangle intersection.
- Clean and optimize the code.
- Change mesh color through inspector (Optional).
- Add textures (Optional).
- Illumination (Optional).
- Shadows (Optional).
- Reflections and refractions (Optional).

March – April 2022

Start of the second and final phase of the development, focused on implementing non-Euclidean aberrations.

- Aberration inspector previsualization.
- Aberration intersection.
- Ray trajectory deformation.
- Deformations editable through inspector.
- Ray position change (Portals).

May 2022

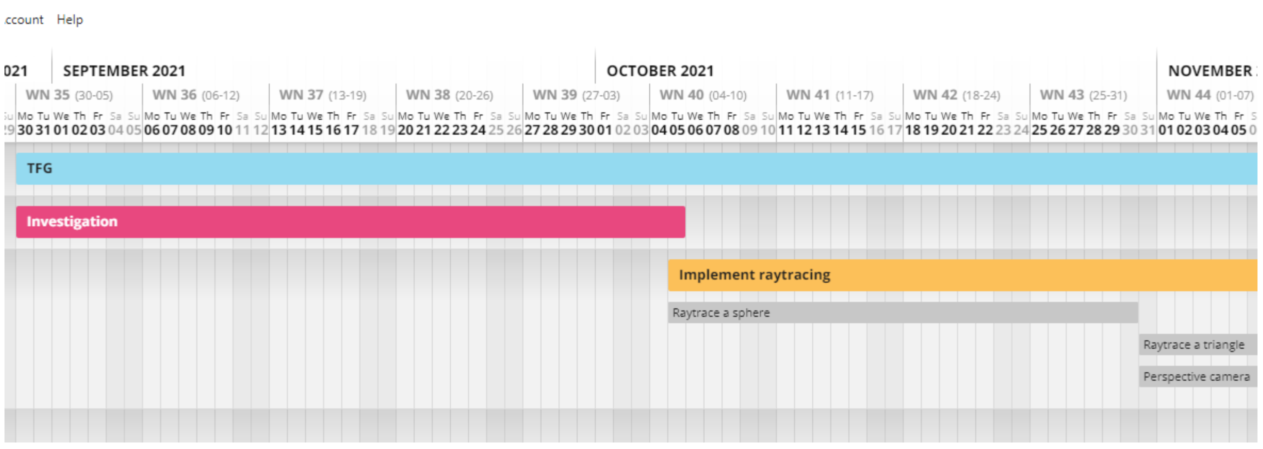
This month will be used as a buffer to finish pending tasks, the tasks planned for it have a low workload.

- Add support to save and load multiple scenes.
- Create test scenes to showcase the features and possibilities.

Organization tools

A Gantt diagram that displays in a high level the state of the project, the order and duration of each task and the deadlines of each feature, created in the application *Agantty* which has a built-in feature to send mails each day/week/month reminding the user of the state of the project.

Figure 1 ***Gantt example***



SWOT Analysis:

Figure 2 ***SWOT***

Strengths	Weaknesses
Previous experience working with shaders. Experience handling long term projects. Self-made engine with free libraries.	Information, algorithms and code has to be adapted to the project. Lack of experience with GPGPU programming. Need a high-end computer to use and work on the project.
Opportunities	**Threats**
First non-Euclidean engine that allows multiple mesh objects edition from the inspector. Student project, which enables the possibility to take risks.	Optimizations may not be enough to allow for real-time rendering. Hard to escalate the project.

Risks and contingency plans

Due to the longevity of this project, taking into account the possible risks during the development was one of the main focuses during the investigation phase, in an attempt to minimize unforeseen problems and have enough resources in case a minor / discarded feature becomes necessary in further iterations of the project.

Logistic problems

Lack of resources: As stated before, the lack of debugging tools and specialized information on the subject make this development prone to experiencing delays due to technical errors and lack of knowledge. This is why a lot of time has been dedicated in researching the topic and learning techniques and utilities outside the scope and objectives of this project, so that any errors or unexpected issues can be addressed without the need to research again. It is also worth noting that, as stated before, there is an entire month dedicated to finish pending features.

Obsolete hardware: My current computer is using a GTX 1050, a low profile GPU that I suspect will be below the requirements of this project. An RTX of the 3000 series would be a great purchase, since they have processing units dedicated specifically to accelerate ray tracing operations. Even though it will probably exceed the minimum requirements of the finished project, it will allow me to finish the entire ray tracing code and then optimize it.

Lack of specialized information: The majority of information related to ray tracing is focused on offline rendering in the CPU and the information on GPU ray tracing is usually about the algorithm itself and using it to render hardcoded shapes. This is the main reason why the ray tracing development is expected to take 4 months since adapting CPU ray tracing information to GPU and discovering how to send large portions of data to the GPU will be the main challenges of the first development phase.

Legacy problems: The base of the project is a raster render engine, created for a university assignment in my third year, due to the lack of knowledge at the time, usability and code structure issues may pose a problem to the development process. Even though it has been revised for the ease of use of the application, and restructured to avoid bugs, it should be taken into account that because of the large code base of the project some problems might go unnoticed for long periods of time.

Programming for the GPU

GPU programming presents a lot of differences in comparison to CPU programming, even though the programming languages used seem similar on a surface level.

Firstly, the code that is being executed in the GPU cannot be debugged, increasing the difficulty of finding problems and slowing the development of the project, since small problems become really hard to locate.

Secondly, communication between the CPU and GPU is not completely transparent, and different hardware might do different conversions of the same data without noticing the user, causing errors and undesired results.

Lastly, optimizing code for the GPU is a very unintuitive process due to certain functions and operations being highly optimized while others are avoided because they slow down the execution (for example, conditionals and loops). One of the most important parts of optimizations comes down to improving and reducing the communication between CPU and GPU.

Ray tracing

Ray tracing is a technique that consists on throwing a ray for each pixel of the screen and painting the pixel depending on the color of the object that collided with the ray (the technique will be explained in detail later).

Nowadays the screen size standard for computers is 1920 x 1080 pixels, therefore, using a raytracing render engine that wants to paint all the screen will need to throw at least 2.073.600 rays. This makes raytracing a very GPU intensive algorithm, which has started to see use in the past 4 years thanks to hardware improvements, but only in small and concrete parts of the render pipeline due to it being very slow compared to rasterization rendering.

Though applying ray tracing in a real time engine is possible, it requires a lot of code optimizations and, more importantly, does not allow for high-poly meshes nor complex scenes with a lot of objects.

Non-Euclidean geometry

Despite a lot of information being available about non-Euclidean geometry, the spaces that can be created and how they interact with rays, there aren’t real time implementations of the concepts described using raytracing rendering and meshes.

Methodology

Methodology type

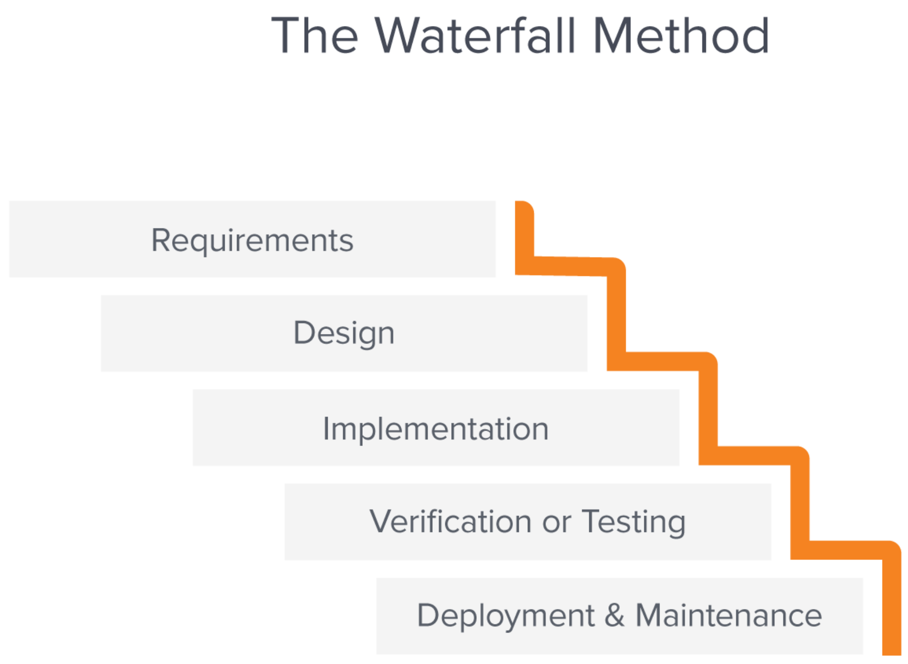
The methodology used for this project is the waterfall methodology. This approach emphasizes a linear progression from beginning to end of a project and relies on careful planning.

The phases considered are:

- Requirements: Detailed understanding of the project's requirements, risks and dependencies.
- Design: Design a technical solution to the requirements.
- Implementation: Create the application.
- Verification: Testing phase to ensure all the requirements have been completed.

Note that the waterfall methodology also considers a maintenance phase, but it will not be taken into account for this project.

Figure 3 ***Waterfall method***



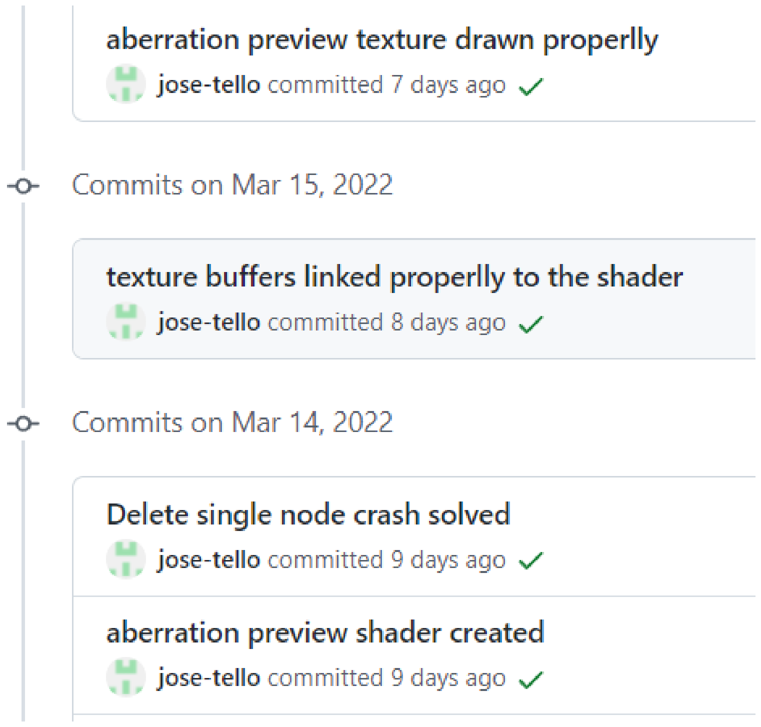
Tools for monitoring the project

GitHub is the main tool used to monitoring the project, this platform is mainly used to store projects and allow big teams to work simultaneously in the same project.

Since this tool has been used extensively during the career, no investigation and learning were needed to start using it to develop the project.

GitHub page: https://github.com/jose-tello/TelloEngine

Figure 4 ***GitHub example***



aberration preview texture drawn properlly
jose-tello committed 7 days ago ✓

Commits on Mar 15, 2022

texture buffers linked properlly to the shader
jose-tello committed 8 days ago ✓

Commits on Mar 14, 2022

Delete single node crash solved
jose-tello committed 9 days ago ✓

aberration preview shader created
jose-tello committed 9 days ago ✓

Here we can observe that GitHub lists all the code actualizations in chronological order, it also allows the user to see the changes done in each commit.

Result validation method

To consider a feature as “completed” it will need to:

- Be tested in different environments and situations alongside the previously implemented features (save / load scenes, create multiple objects…).
- In case that any bug was found, solving it will become a high priority task.
- If the feature slows down the engine below 30 FPS optimization of the feature will become a high priority task.
- Go through an improvement phase where the code will be cleaned and improved to ensure a good base of code and reduce future risks.
- The feature has to be fully implemented with all of its functionalities

Only when the conditions above are accomplished the task will be considered “completed”.

In final stages of the development a usability test will be done to assess interface usability problems and correct them. The correction of this problems will be considered a low priority task.

Theoretical Framework

Ray tracing

Ray tracing is a method of graphics rendering that simulates the physical behavior of light (Nvidia, n.d.).

Most used method in offline rendering due to its impressive and realistic results when rendering emissive, reflexive and refractive materials, lightning and shadows, volumes (smoke, fire, clouds…) and fluids (water, lava…).

Started seeing use in real-time engines due to new GPUs that have started implementing technology dedicated exclusively to this technique. Nowadays, even though it's still too slow to use it in real time, the technique is used alongside the raster render pipeline mainly to create small light reflections, light highlights and clouds.

Algorithm

This method consists on throwing rays from the origin of the camera to the center of each pixel. Then check what objects intersected with the ray and set the color of the pixel with the color of the nearest object that intersected with the ray (L. Cook, Porter, & Carpenter, 1984).

Once a ray intersects with an object, it will throw secondary rays from the point of collision, in order to evaluate different lightning and material-related effects, such as reflections and shadows. Once the secondary rays have been evaluated (secondary rays could create more secondary rays in order to increase visual fidelity) the resulting colors will be mixed and the resulting color will be set as the color of the pixel.

Compute shaders

As stated before, the algorithm itself is not slow, the problem is that it has to be done a lot of times. In a 1920 x 1080 screen, we would need to throw 2.073.600 rays to render the scene without light, light effects or shadows, witch's objects borders would look like the image has low resolution (aliasing). Adding lights (for example) and simple diffuse illumination, would require to double the amount of rays casted into the scene.

Compute shaders are general-purpose shaders that allow to use the GPU for other tasks - GPGPU programming (Gerdelan, 2016). This allows us to use the GPU to execute the same code multiple times in parallel, thus enabling us to calculate various ray intersections at the same time, greatly reducing the execution time.

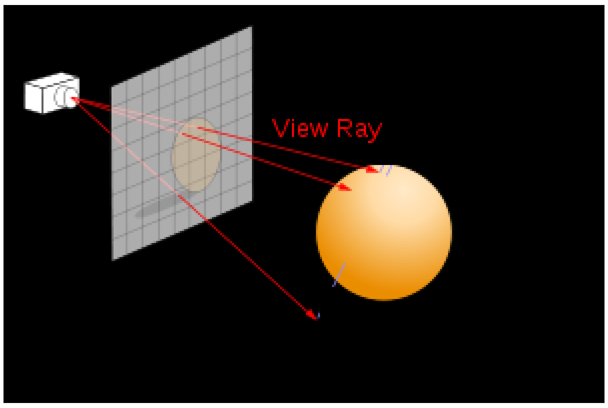
How to implement it

Firstly, we will set a compute shader that will render to a texture, which we will then render to a rectangle in our engine (scene window), the compute shader will calculate each ray independently (one ray for each pixel of the texture).

In the compute shader, the pixel position will be remapped to values between -1 and 1, will be used to calculate the direction vector from the camera position to the current pixel of the texture.

The ray will test if it collided with any triangle in the scene, if it does, it will get the color of the nearest geometry it intersected with and paint that pixel with it. If the ray does not intersect with any geometry, the pixel will be painted black.

Figure 5 ***Ray tracing illustration***

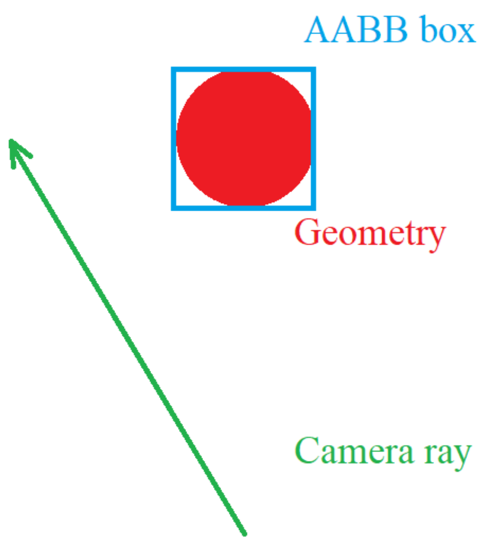


Optimizations

As stated before, this process is really slow but it can be optimized to a usable state in real-time with enough optimizations.

Space partitioning: A mesh can be composed of hundreds or thousands of triangles, and each ray has to check for collisions with every one of them, but as we can see in the next figure, there are situations where the ray does not collide with the geometry, causing to do a lot of unnecessary collision tests:

Figure 6 ***Space partitioning example***



If we simplify each object into an AABB, which are cubes that encapsulate all the geometry of the object. Then we can test if the ray collides with the box of the object, and only testing collision with triangles if the ray intersected with the box, allowing us to quickly discard objects that don’t intersect with the ray.

Fast ray-triangle intersection: A lot of research has been done to find the fastest way to calculate if a ray intersects with a triangle with the objective to speed up the raytracing algorithm. The Möller-Trumbore algorithm will be the one used in the project, this algorithm translates the origin of the ray and then changes the base of that vector which yields a vector (*t*, *u*, *v*), where *t* is the distance to the plane in which the triangle lies, and (*u*, *v*) represents the coordinates inside the triangle (Möller & Trumbore, 1997).

GPU optimizations: GPU programming operates very differently compared to CPU programming at a low level, which mainly turns into avoiding if statements and loops thus reducing the clarity of the code. The most problematic and slow operation, however, is when the CPU has to communicate with the GPU.

Textures are the fastest structures that can be sent to the GPU, and extracting their values is also one of the fastest operations in the GPU, thus making it the best method to send large amounts of information from the CPU to the GPU. In our case that would be the geometry of the scene.

All the vertices, all the indices and the texture coordinates are codified as RGB values into 3 textures and sent to the GPU where the compute shader translates the texture data into geometry data, and then calculates the ray intersections.

Euclidean geometry

Euclidean geometry is a mathematical system based on 5 principles:

- A straight line may be drawn between any two points.
- A piece of straight line may be extended indefinitely.
- A circle may be drawn with any given radius and an arbitrary center.
- All right angles are equal
- If a straight line crossing two straight lines makes the interior angles on the same side less than two right angles, the two straight lines, if extended indefinitely, meet on that side on which are the angles less than the two right angles. (Two parallel lines will never cross).

(Bogomolny, 1996).

Non-Euclidean geometry

Any type of geometry that does not follow one or more of Euclid’s postulates is considered non-Euclidean (Miller, 2018).

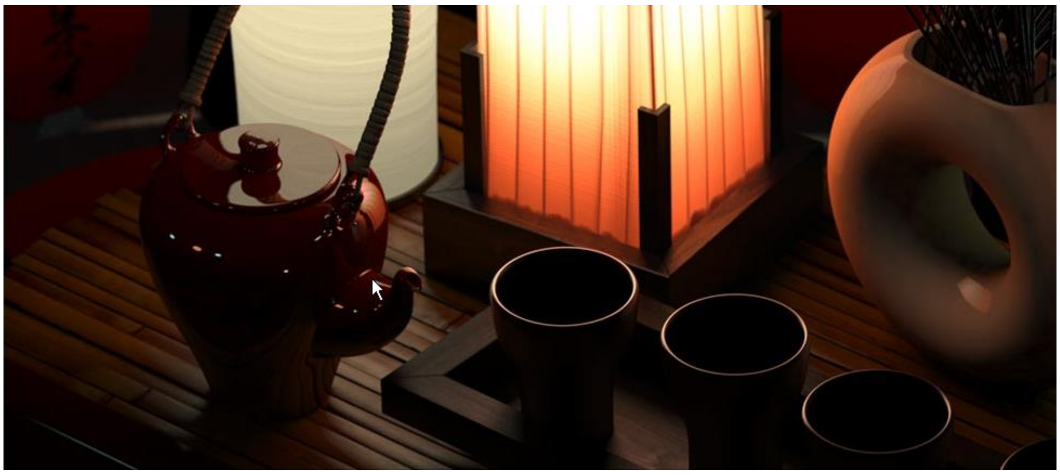
State of the art

Ray tracing

This rendering method is the most widely spread in offline renders due to its photo-realistic results and the capacity to handle difficult substances / elements such as smoke, fire, fluids…

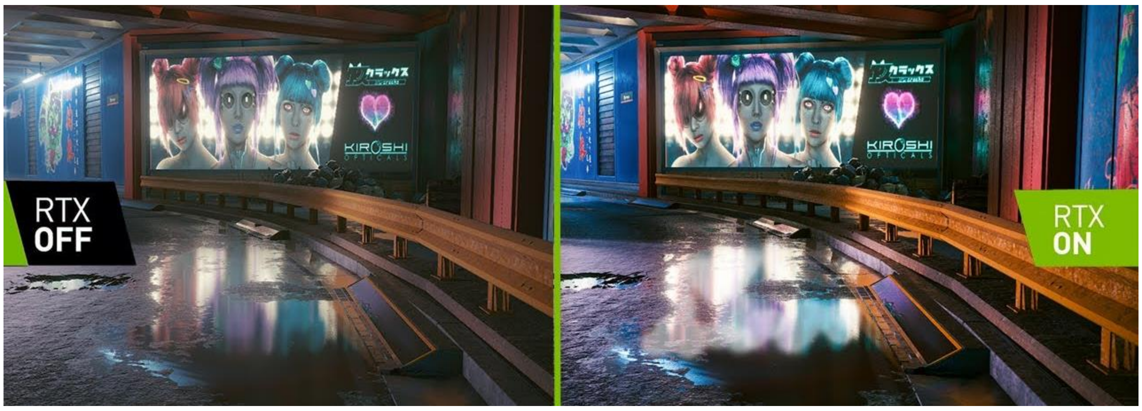
Houdini, 3Ds Max, Maya, Blender, RenderMan (Pixar)… Are some of the most advanced 3D edition tools in the market of Animation, 3D modelling and special effects which use render engines based on this technique.

Figure 7 ***Houdini render example***



In the past years, as GPU's have started adding support for raytracing, the method has become fast enough to be used in real-time in hybrid render pipelines where there are 2 render passes, the first pass renders the scene with the raster algorithm, and the second pass renders certain surfaces of the scene with ray tracing (mainly reflections and ambient illumination).

Figure 8 ***Raster vs. Hybrid render comparison***



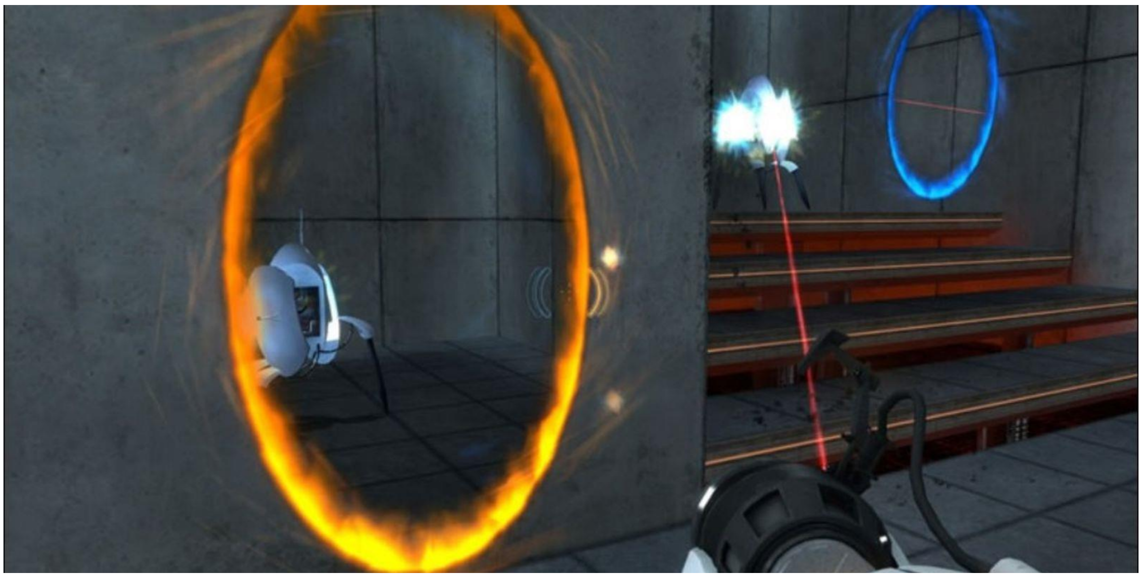
On the left side of the image we can find the scene rendered with the raster algorithm and on the right side the same scene rendered with a hybrid pipeline. We can observe the improved quality of the reflections on the puddles and how the light effects are more realistic and clear.

Non-Euclidean geometry in Games

There are extensive examples of games that implemented in some way or another non-Euclidian spaces (*Portal*, *Antichamber*, *The Stanley Parable*, *Split Gate*...), but most of them opted to fake the effect, severely limiting its possibilities.

I will use the video game *Portal* to explain how non-Euclidean spaces are faked. In this game the player can create portals that connect spaces together and they can traverse this connection to arrive to the other connected space.

Figure 9 ***Portal example***



Note: Here we can observe that the blue portal connects the space with the orange one.

In this case the position of the orange portal is located a texture and a collider, this texture is the render target of a camera situated behind the blue portal, which matches the field of view of the player, creating the illusion that the space is connected, the collider is used to instantly move the player to the blue portal position, fast enough so that the player does not notice it (DigiDigger, 2017).

There is a game still in development that accomplished rendering hyperbolic geometry (type of non-Euclidean geometry not covered in this research) in real time, named *Hyperbolica*.

Non-Euclidean rendering engines

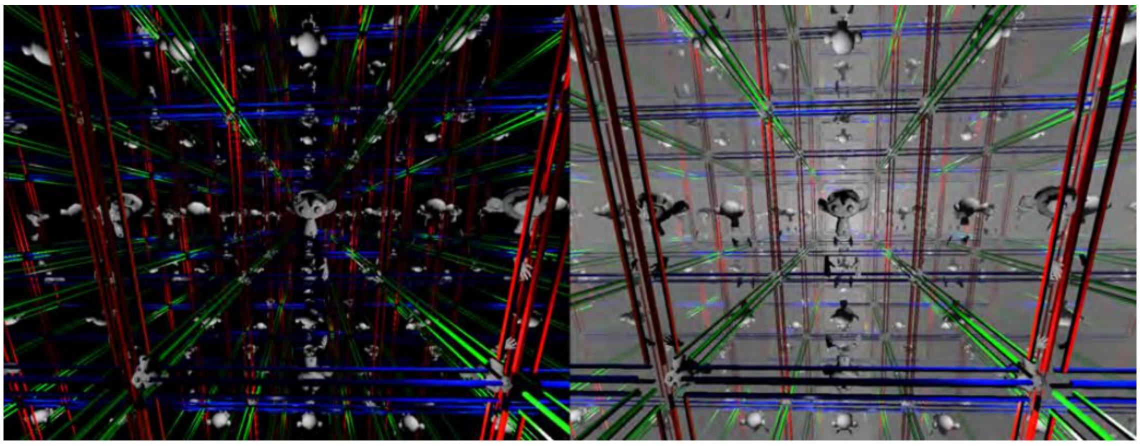
During the investigation process, 3 different non-Euclidean real-time rendering engines stood out, each one with different features and capabilities, but none of them had the tools needed to modify the scene and aberrations in execution through an inspector or a gizmo. It is also worth mentioning that only one of them was able to render meshes, but with a high limitation in number.

Visualization of Non-Euclidean Spaces using Ray Tracing (2019)

Able to render different aberrations with simple geometry placed on the scene, but as the name implies, it is not an engine with the different tools needed to load scenes, meshes, place the objects and edit the aberrations during the execution of the program.

For more information:
https://www.researchgate.net/publication/337472155_Visualization_of_Non-Euclidean_Spaces_using_Ray_Tracing

Figure 10 ***Visualization of Non-Euclidean Spaces using Ray Tracing caption***



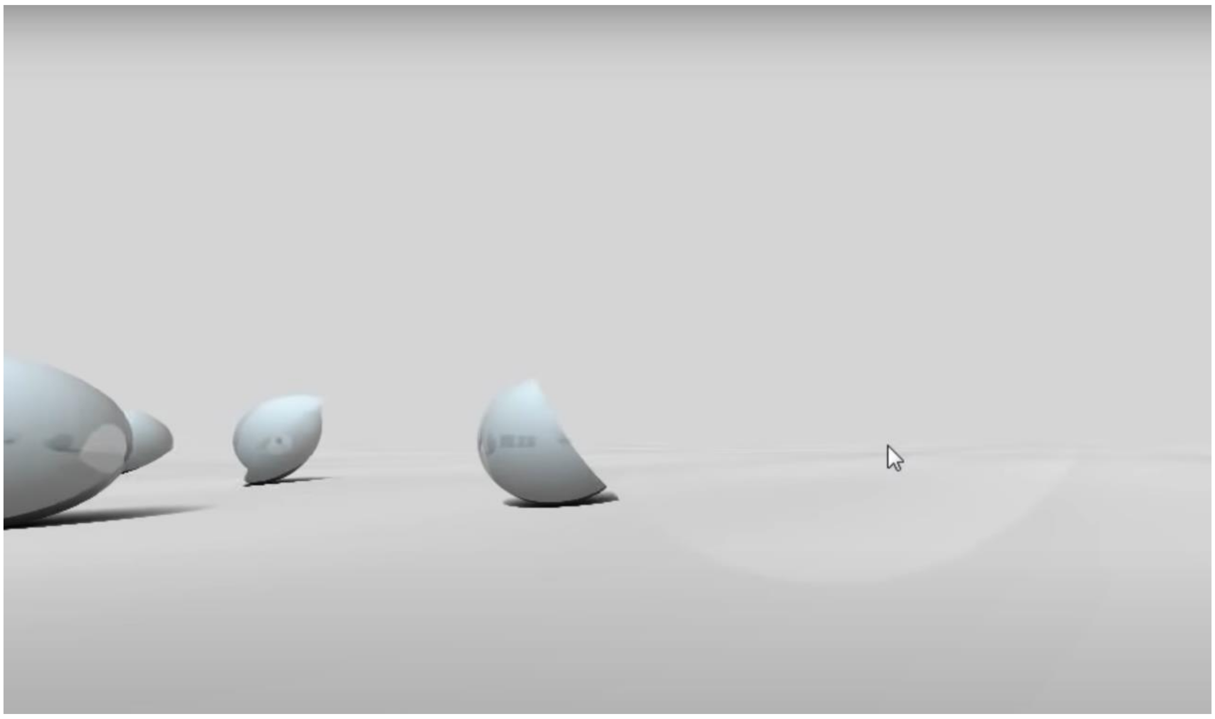
Real-Time non-Euclidean Ray-tracer Demo

Able to render aberrations and portals with different shapes, but has optimization problems (fps drops when going through a portal or aberration) and is unable to render meshes and textures. All the objects are made by stating shape variables and setting the object's position, which limits the possible objects to simple shapes. It is also worth mentioning that the project doesn't have the tools to modify the objects and aberrations during the execution, all of them are hardcoded into the scene (Varun R., 2013).

For more information:

https://youtu.be/YvU-srHhQxw

Figure 11 ***Real-Time non-Euclidean Ray-tracer Demo caption***



Non-Euclidean GPU Ray Tracing Test

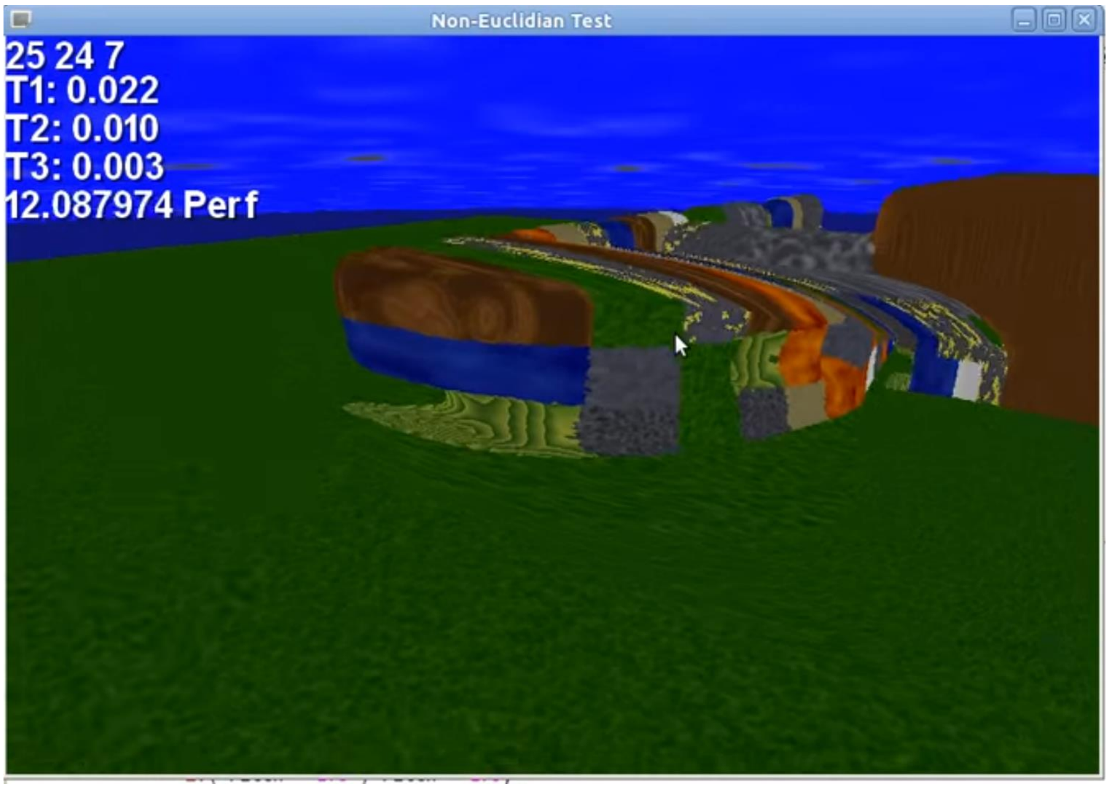
Able to render aberrations that change the direction of the rays and modify the speed of the camera accordingly, thus elongating and compressing space. Can handle multiple objects with different textures, but it cannot render geometry as all the objects shown are made by stating shape variables and setting a position (a sphere may be defined as a point in space with a certain radius) thus limiting the objects to simple shapes (CNLohr, 2011).

For more information:

https://youtu.be/0pmSPlYHxoY

https://youtu.be/tl40xidKF-4

Figure 12 ***CNLohr engine caption***



```
25 24 7
T1: 0.022
T2: 0.010
T3: 0.003
12.087974 Perf
```

Project development

Conclusions

Bibliography

Möller, T., & Trumbore, B. (1997). *Fast, Minimum Storeage Ray/Triangle Intersection*. Retrieved from https://cadxfem.org/inf/Fast%20MinimumStorage%20RayTriangle%20Intersection.pdf

Bogomolny, A. (1996). *Cut the knot*. Retrieved from http://www.cut-the-knot.org/triangle/pythpar/Fifth.shtml

CNLohr. (2011). *Youtube*. Retrieved from https://youtu.be/0pmSPlYHxoY

DigiDigger. (2017). *Youtube*. Retrieved from https://youtu.be/_SmPR5mvH7w

Gerdelan, A. (2016). *An Introduction to Compute Shaders*. Retrieved from https://antongerdelan.net/opengl/compute.html

Khronos Group. (n.d.). *OpenGL Overview - The Khronos Group Inc*. Retrieved from https://www.khronos.org/opengl/

Kuri, D. (2018). *GPU Ray Tracing in Unity*. Retrieved from http://three-eyed-games.com/2018/05/03/gpu-ray-tracing-in-unity-part-1/

L. Cook, R., Porter, T., & Carpenter, L. (1984). *Distributed Ray Tracing*. Retrieved from https://artis.inrialpes.fr/Enseignement/TRSA/CookDistributed84.pdf

Lapere, S. (2015). *Raytracey*. Retrieved from http://raytracey.blogspot.com/2015/10/gpu-path-tracing-tutorial-1-drawing.html

Marschner, S. (2017). *Textures and normals in ray tracing*. Retrieved from https://www.cs.cornell.edu/courses/cs4620/2017sp/slides/06rt-textures.pdf

Miller, G. (2018). *Encyclopedia*. Retrieved from https://www.encyclopedia.com/science-and-technology/mathematics/mathematics/non-euclidean-geometry

Non-Euclidean Worlds Engine. (2018). Retrieved from https://www.youtube.com/watch?v=kEB11PQ9Eo8

Nvidia. (n.d.). *Nvidia developer*. Retrieved from https://developer.nvidia.com/rtx/ray-tracing#:~:text=Ray%20tracing%20is%20a%20method,to%20pioneer%20the%20technology%20since.

Pitici, M. (2008). *Non-Euclidean Geometry Online: a Guide to Resources*. Retrieved from http://pi.math.cornell.edu/~mec/mircea.html

Quilez, I. (n.d.). *Iquilezles*. Retrieved from https://iquilezles.org/index.html

Scratchapixel. (2009). Retrieved from https://www.scratchapixel.com/index.php?redirect

Varun R. (2013). *Youtube*. Retrieved from https://youtu.be/YvU-srHhQxw

Vinícius Silva, Tiago Novello de Brito, Luiz Velho, Djalma Lucio. (2019). *Visualization of Non-Euclidean Spaces using Ray Tracing*. Retrieved from https://www.researchgate.net/publication/337472155_Visualization_of_Non-Euclidean_Spaces_using_Ray_Tracing

Vries, J. d. (2014). *Learnopengl*. Retrieved from https://learnopengl.com/Introduction

Zwan, J. v. (2020). *Linear Interpolation along a Triangle with Barycentric Coordinates*. Retrieved from https://observablehq.com/@jobleonard/linear-interpolation-along-a-triangle-with-barycentric-co