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.

UI: User interface

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

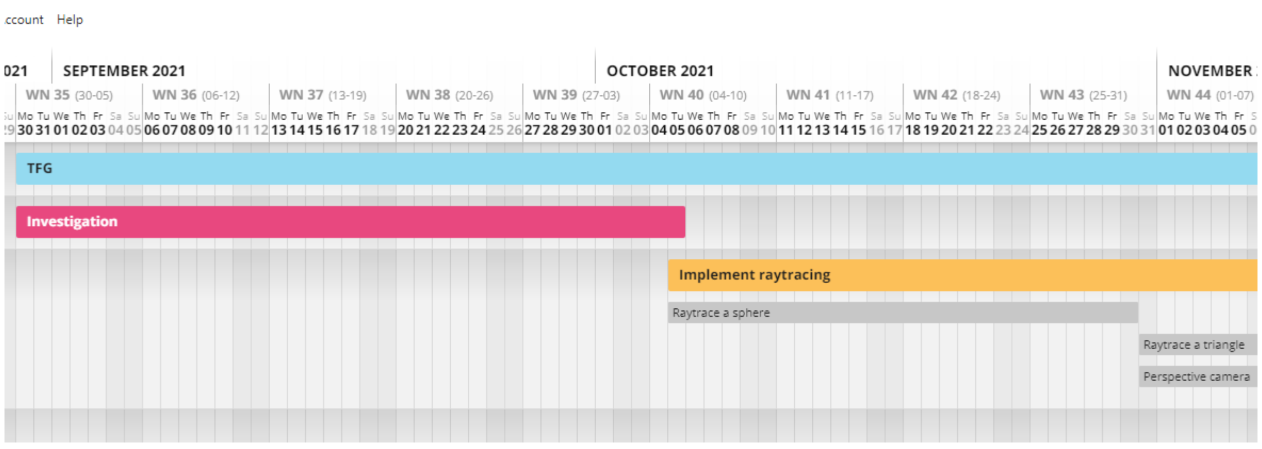
This month will be used as a buffer to finish pending tasks, the tasks planned for it have a low workload.

- 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.	Information, algorithms and code has to be adapted to the project.
Experience handling long term projects.	Lack of experience with GPGPU programming.
Self-made engine with free libraries.	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.	Optimizations may not be enough to allow for real-time rendering.
Student project, which enables the possibility to take risks.	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

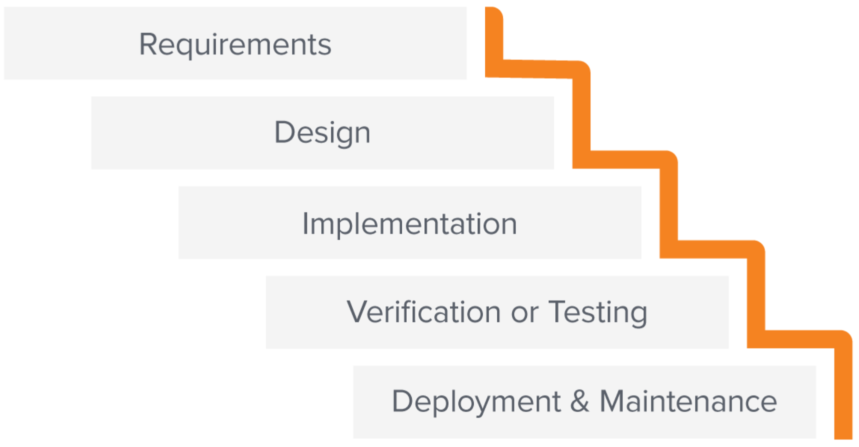
The methodology used for this project is the waterfall methodology. This approach emphasizes a linear progression from beginning to end of a project and relies on careful planning.

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.
- Cleaning & optimization: Clean the code and optimize it in order to facilitate the implementation of future features.
- Verification 2: Test that the optimizations are working properly and have not generated bugs.

Note that the waterfall methodology also considers a maintenance phase, but it will not be taken into account for this project. Instead two new phases were created to ensure a good and optimized code and that the project runs in real-time in as many computers as possible.

Figure 3 ***Waterfall method***



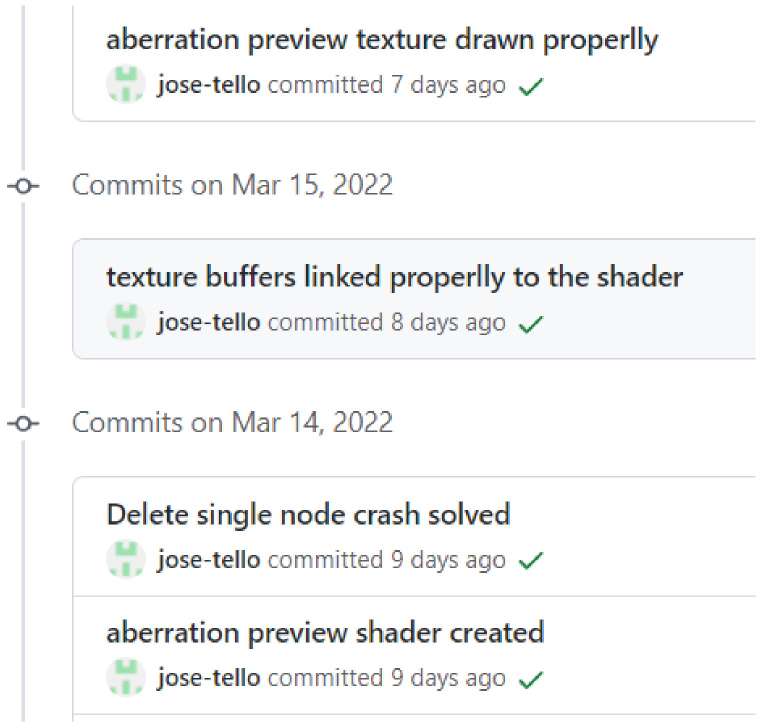
Tools for monitoring the project

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.

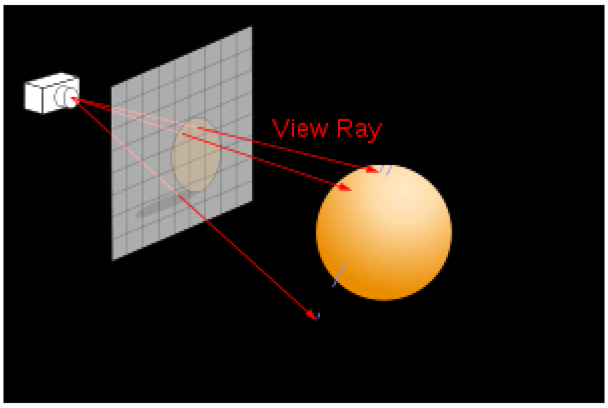
How to implement it

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***

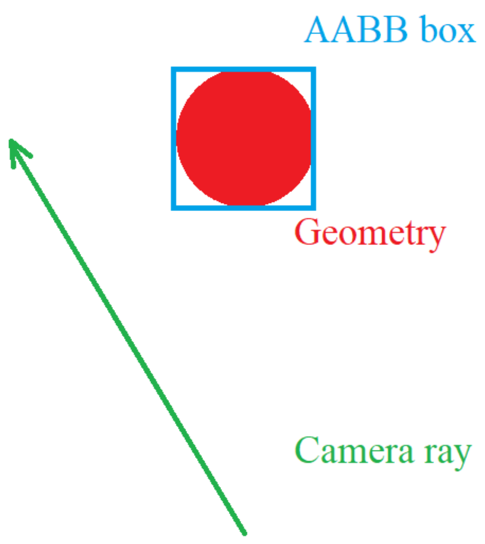


Optimizations

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).

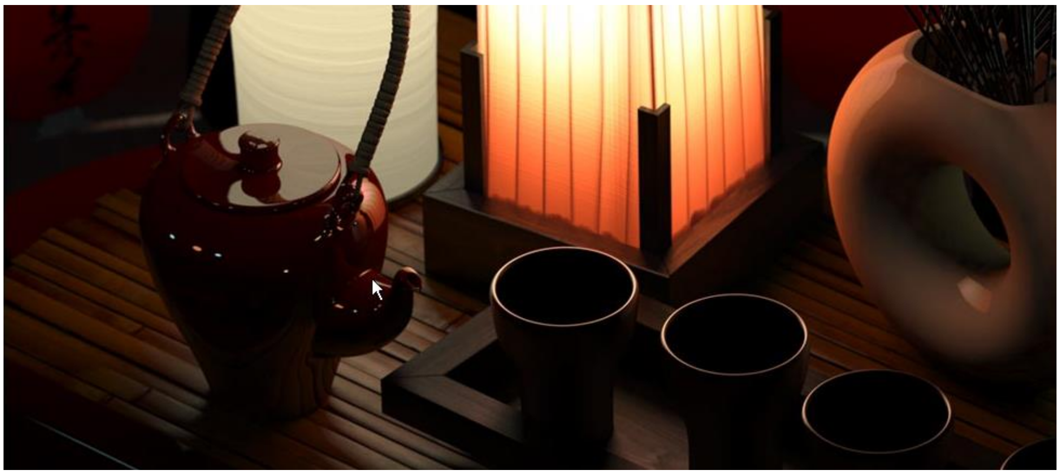
State of the art

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…

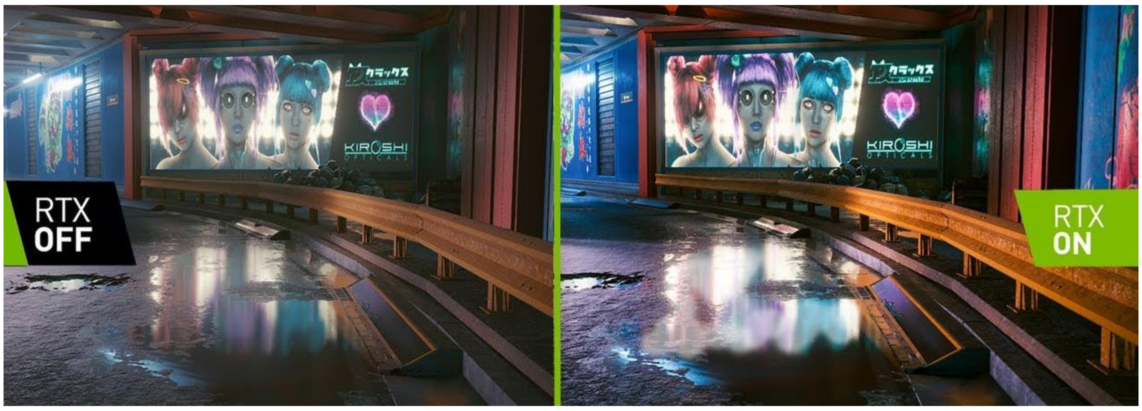
Houdini, 3Ds Max, Maya, Blender, RenderMan (Pixar)… Are some of the most advanced 3D edition tools in the market of Animation, 3D modelling and special effects which use render engines based on this technique.

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***



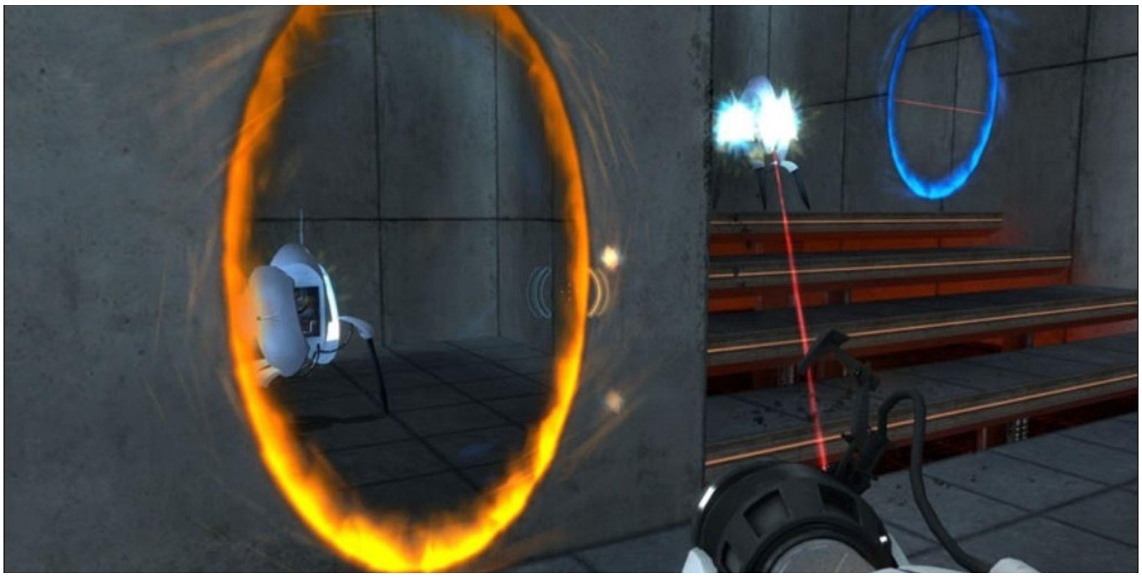
On the left side of the image we can find the scene rendered with the raster algorithm and on the right side the same scene rendered with a hybrid pipeline. We can observe the improved quality of the reflections on the puddles and how the light effects are more realistic and clear.

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

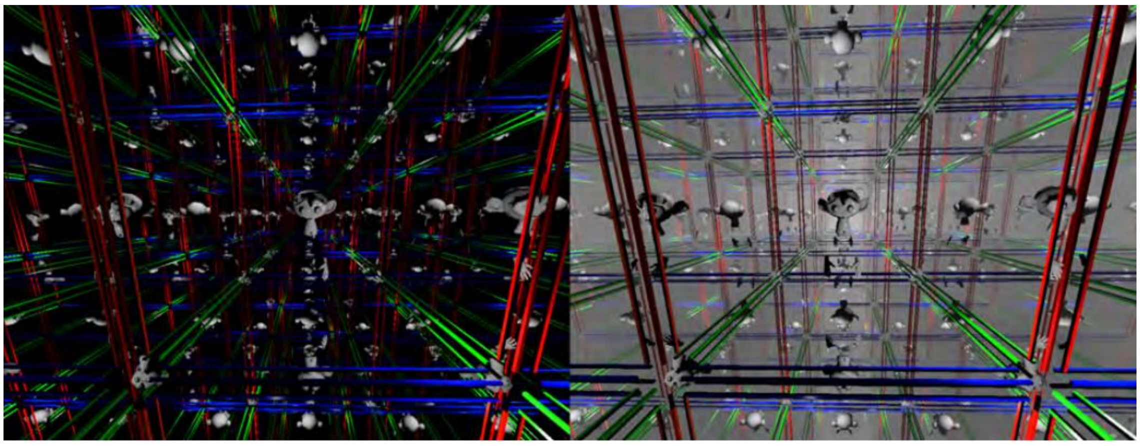
During the investigation process, 3 different non-Euclidean real-time rendering engines stood out, each one with different features and capabilities, but none of them had the tools needed to modify the scene and aberrations in execution through an inspector or a gizmo. It is also worth mentioning that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



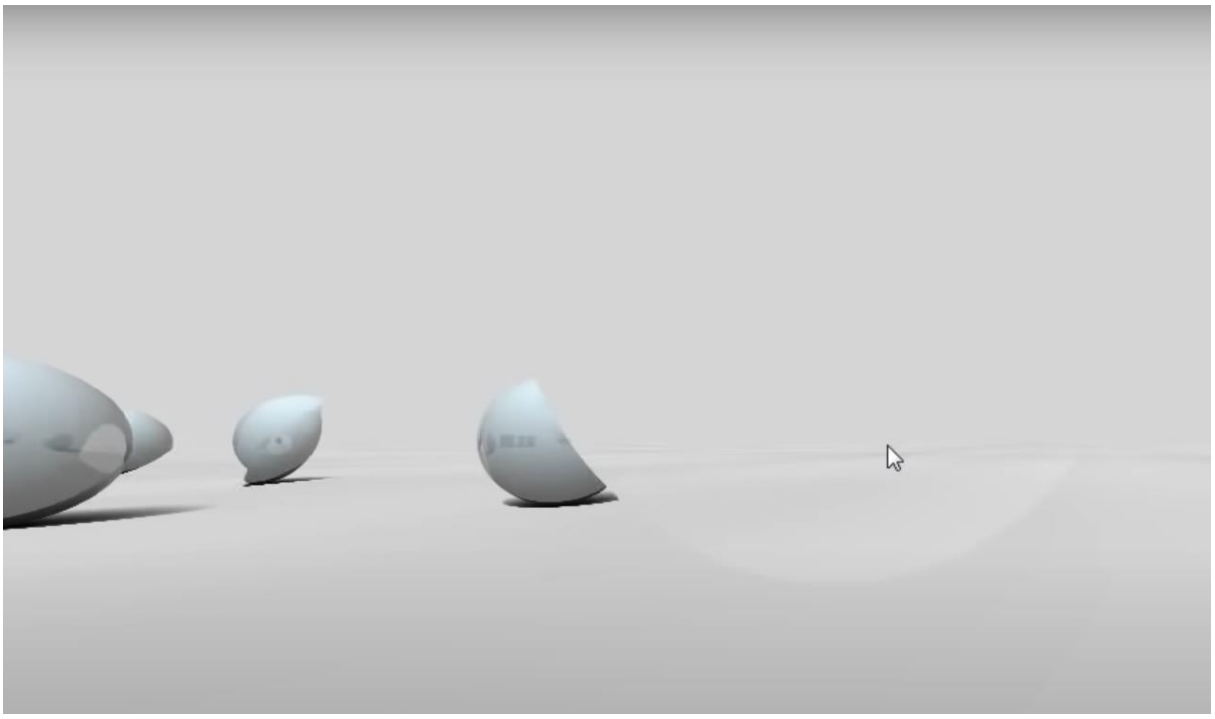
Real-Time non-Euclidean Ray-tracer Demo

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

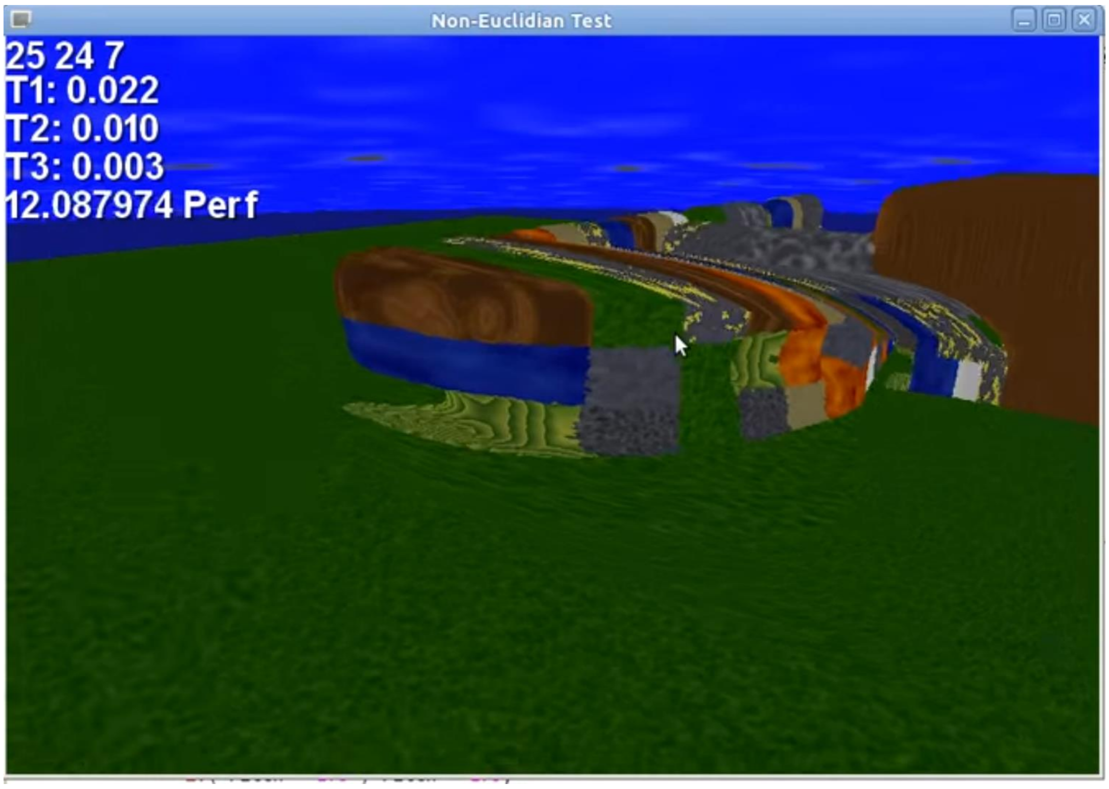
Able to render aberrations that change the direction of the rays and modify the speed of the camera accordingly, thus elongating and compressing space. Can handle multiple objects with different textures, but it cannot render geometry as all the objects shown are made by stating shape variables and setting 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

To have a better understanding of the process of developing this project I will first define the list of features needed to complete the project (requirements) and then explain the design, implementation and verification process of each one in chronological order.

The verification paragraph will contain the verification and verification 2 steps.

Requirements

Real-time GPU ray tracing algorithm with meshes

As stated before, this rendering method will allow us to deform the way elements are visualized with a level of freedom that cannot be archived using the raster rendering technique. Mesh rendering will be needed to create simple scenes that will showcase the engine capabilities. A lot of optimization investigation will be needed in order to archive a stable frame rate that will allow users to edit the scene and see the aberrations effects in real-time.

Aberrations (edition and previsualization tools)

In order to allow users to experiment with non-Euclidean design experiences, a set of aberration edition tools and a method to visualize their location and area of effect will be needed, it will also allow us to test edge cases once we start deforming the rays.

Aberration ray deformation and collision detection

Functionality that will displace and change the direction of the camera rays once they interact with an aberration, allowing us to create the illusion that the space is being expanded or compressed.

Camera-Aberration interaction

Reduce or increase the camera movement speed to create the illusion that the space is being deformed and apply the deformation to the rays if the camera is inside an aberration.

Development: Real-time GPU ray tracing algorithm with meshes

Design

Compute shaders: To start the implementation of the ray tracing algorithm, first we need to decide on how will we calculate the collisions with the camera rays with the scene. This is a simple calculation that will be repeated 640.000 times each frame (one ray for each pixel in an 800x800 camera).

This kind of scale problem can be subsided by using the GPU to do this calculations (compute shaders), in our case we will be using OpenGL version 4.3.

CUDA and OpenCL (parallel GPU computing platforms) were quickly discarded due to the lack of experience and the need to rewrite a major part of the base engine in order to implement them which would delay the start of the project. It should be noted that CUDA and OpenCL would provide with a faster execution time thanks to the access to low level hardware administration.

Ray tracing algorithm: The implementation is very straightforward as it makes use of basic math vector calculations in order to find the closest collision point in a triangle in a given direction, the Möller-Trumbore algorithm will be the one implemented since it is widelly used in benchmarks to compare other algorithms in concrete cases due to its polivalence.

Passing data to the GPU: Passing data to the GPU in the fastest way possible will be a determining factor to reach real-time frame rate. The transform data (view camera matrix, mesh world matrices…) will be send as uniforms (easy to set and flexible, but very slow) and the scene vertex data (vertices, indices, texture coords…) will be send as textures to the GPU.

Uniform buffers are faster than uniform variables at the cost of using more memory, but the method could not be implemented due to time constrains. It consists in writing all the uniform data directlly in the GPU memory and interpret it in the shader.

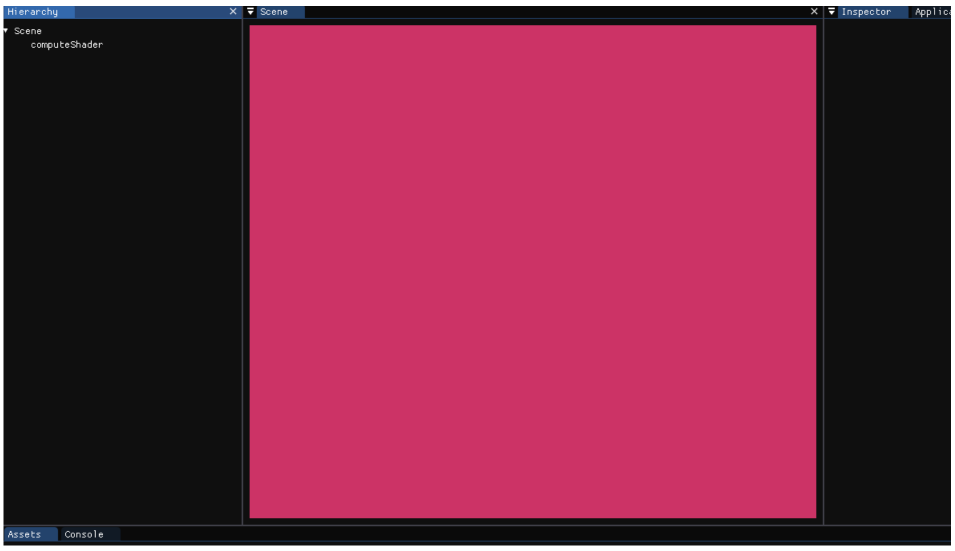
Implementation

Compute shaders: Firstly, I created the option to select the rendering method (raster / ray tracing) in the editor, then prepared the output texture for the compute shader, and passed it to the ImGui window so that it could be displayed.

Then prepared the shader importer module to load compute shaders and to allow shaders without vertex and fragment components. I wrote a simple compute shader that painted each pixel pink and checked that it was loaded correctly without errors.

Lastly the compute shader was set up and dispatched in the ray tracing render function, and got the following output.

Figure 13 ***Compute shader output***



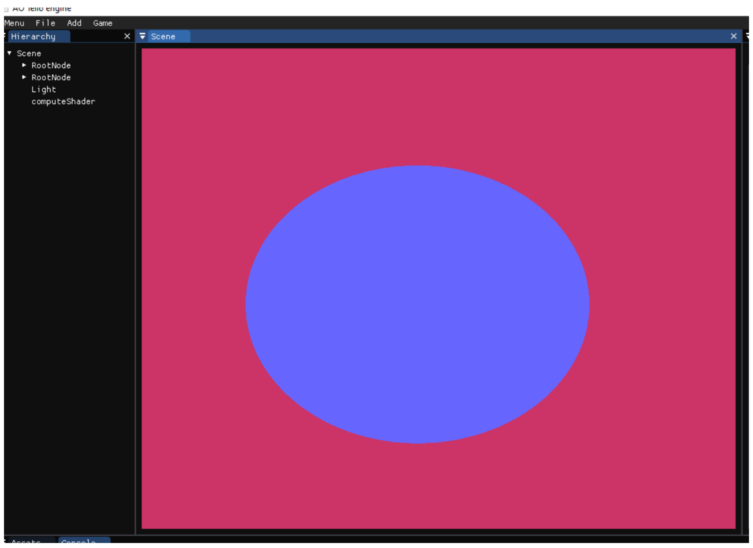
This simple output assures us that the compute shader is loaded correctly, it's output is correct and the output texture is being handled properly.

Ray tracing algorithm: Various steps were done in order to test various aspects of the code (camera ray's direction, distance test, compilation speed…).

One ray would be calculated for each pass of the compute shader and this ray would paint one pixel of the texture. The direction of the ray would be determined by the camera position and the pixel of the texture in the OpenGL coordinate system. Since the texture pixel coordinates range from 0 to pixel-size, we need to transform it to a range between -1 and 1. This would allow us to have a perspective camera which will help with depth and distance perception.

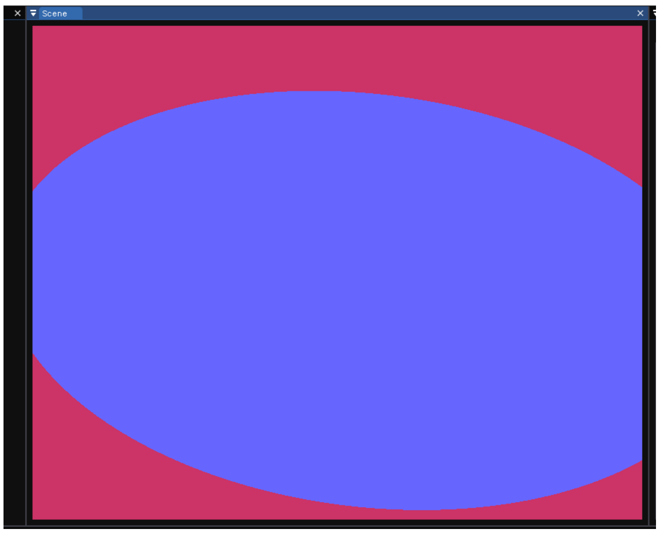
The first shape that was tested was a sphere, a simple geometric shape that is fast to implement.

Figure 14 ***Sphere ray tracing test***



The results seem satisfactory, but once the camera was moved sideways, the sphere would deform, indicating that something was not being calculated correctly.

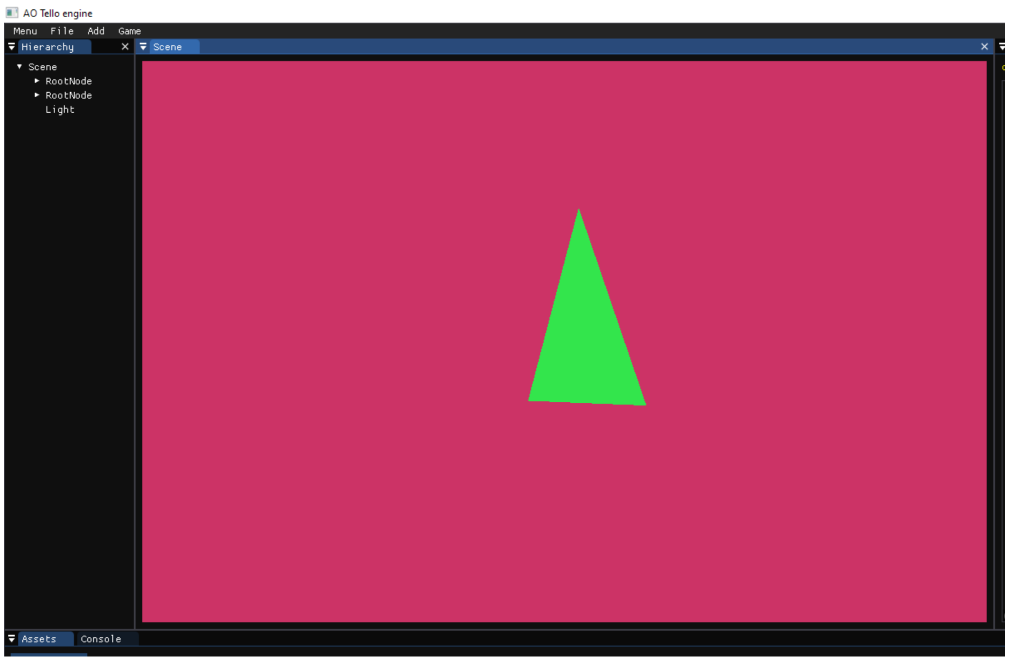
Figure 15 ***Sphere deformation***



The problem was solved by checking if the point of collision is behind the camera, and calculating the camera rays using the camera's FOV and aspect ratio.

The next shape that was tested was a triangle, which is the basic shape that constructs meshes. The first implementation used an analytical solution to compute if the ray intersected with the triangle, this implementation is way too slow to be used alongside meshes (take into account that the difference with one triangle is negligible, but once we have a mere 200 triangles in the scene which we will have to test collisions with every triangle for every ray we cast) and was replaced by the Möller-Trumbore algorithm.

Figure 16 ***Triangle ray tracing test***



Passing data to the GPU: As stated before, we will need to codify the mesh vertex data as a texture, we will need to indicate OpenGL to not normalize the texture values.

In the vertex texture each pixel will represent a point, and its RGB values will represent the xyz coordinates.

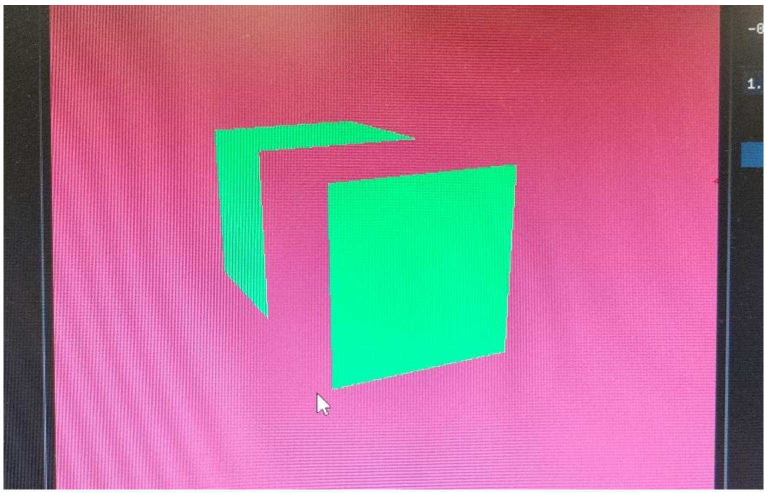
In the texture coords texture each pixel will represent a 2d point (uv), and its RG values will represent the x and y coordinates.

In the indice texture each pixel will represent a triangle, and its rgb values will represent an iterator to the vertex texture (point).

The mesh data (world transform, vertex offset, vertex count, color and texture) will be send using uniforms.

Once the data was correctly send to the GPU small adaptations were done in the shader code so that it could translate the geometry textures and find the nearest collision point to the camera position.

Figure 17 ***Cube mesh error***



An error in the calculation of vertex collisions caused certain vertex of the mesh to be displaced in the Z direction, the problem was caused because of a sign error in a mathematic calculation.

Figure 18 ***Multiple cube meshes***

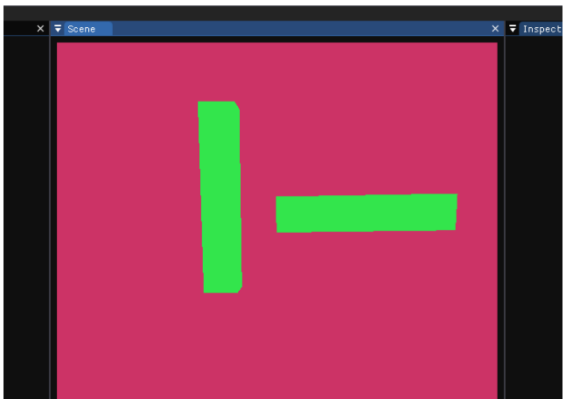
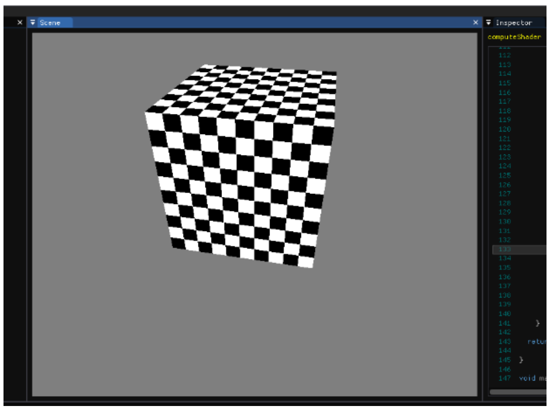


Figure 19 ***Textured Cube***



In the left image we can see two cube meshes that have been moved, rotated, and scaled in different axis. The right image shows a cube with the default texture.

Cleaning & Optimization

As stated before, each ray is testing all the triangles in the scene for collision, which is very inefficient. To solve this issue AABB boxes were implemented with the objective to avoid unnecessary triangle tests.

A system to automatically add an AABB box to each mesh was already implemented (used in the raster render), so the only thing needed was to send the AABB box of each mesh to the GPU using uniform variables, and create a method that checks ray-AABB box intersections.

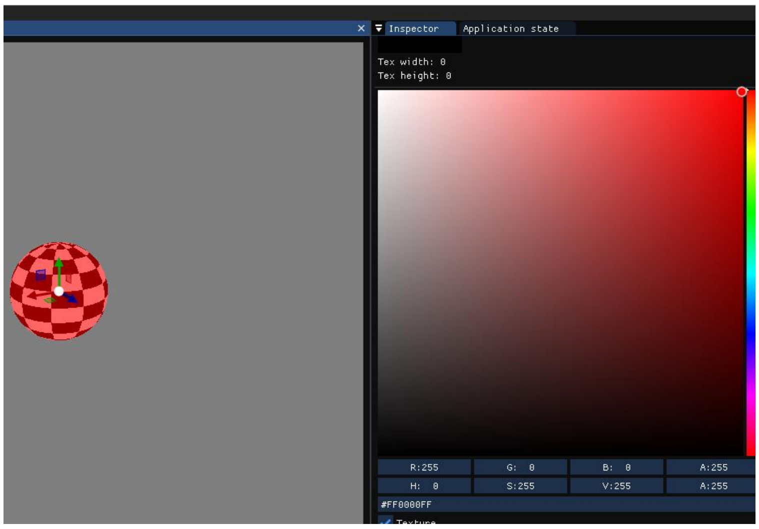
The next step was to clean the code, which involved renaming variables, reusing the variables that only needed to be saved, reduce and substitute the maximum number of *if* statements as possible (branching is a huge efficiency factor in GPU programming). This changes made the code less readable but in the GPU code the speed was deemed as the most important aspect.

Validation

At this point, it was possible to add multiple meshes to the scene which can be translated, rotated and scaled using the inspector. The user is also able to import meshes in .obj format. The meshes have a default texture that helps the user judge depth and distances.

The mesh can also be colored using a color picker in the inspector.

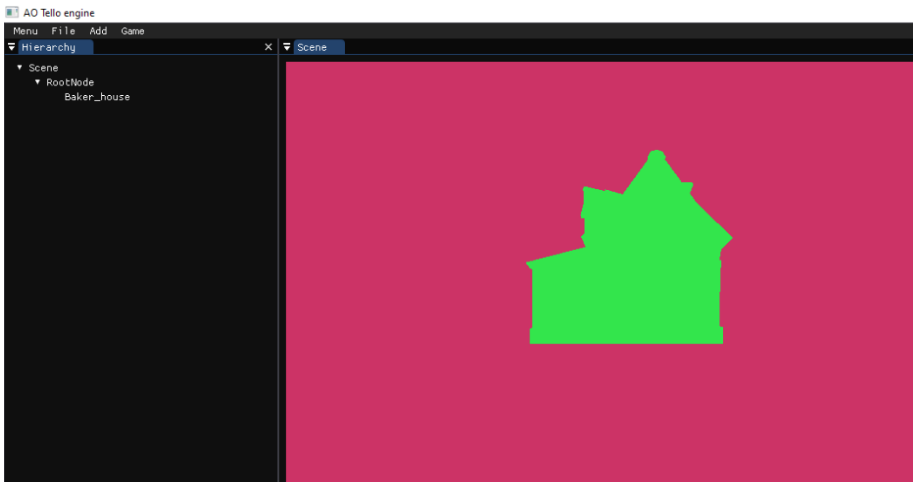
Figure 20 ***Mesh color picker***



```
Tex width: 0
Tex height: 0
R:255   G: 0    B: 0    A:255
H: 0    S:255   V:255   A:255
#FF0000FF
```

The engine is currently working at around 150 – 200 FPS in scenes with 10 cubes in a RTX 1050Ti GPU's. And at around 350 – 500 FPS in the same scene in a RTX 3060Ti.

Figure 21 ***House mesh***



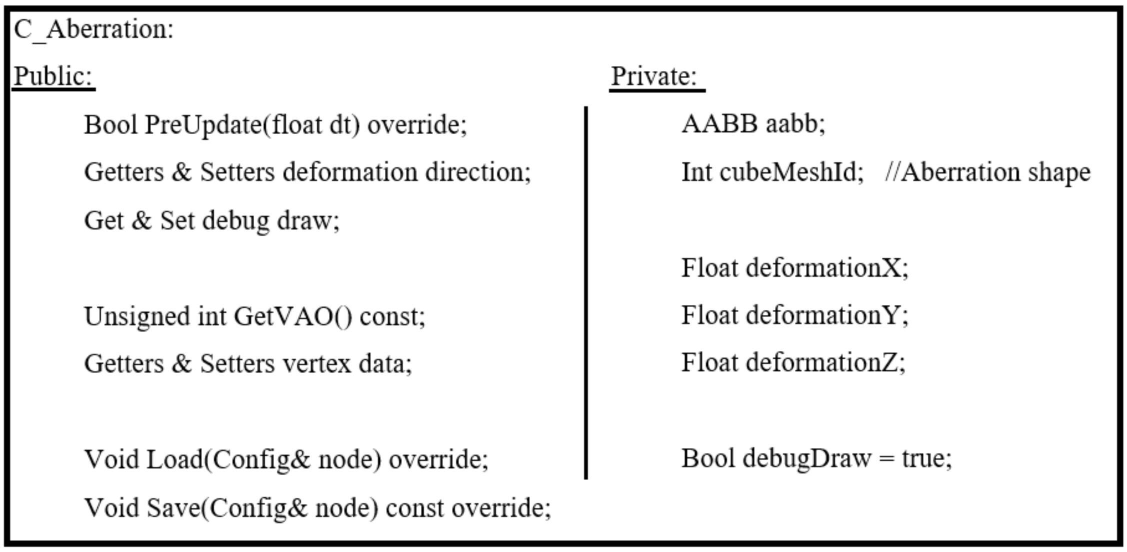
A scene with a house mesh (5500 vertices) would run at around 60 FPS in a RTX 3060 Ti, this was estimated as a valid result since the rendering method was expected to be slow, but this limit will allow to create scenes with shapes more complex than simple cubes.

Development: Aberrations edition and previsualization tools

Design

Create aberration component class: Create a class with all the variables and base functionality needed: Save and load, initialize, update, mesh (the shape of the aberration), deformation direction, activate / deactivate debug draw….

Figure 22 ***Component aberration class UML***



```
C_Aberration:
Public:                                          Private:
        Bool PreUpdate(float dt) override;              AABB aabb;
        Getters & Setters deformation direction;        Int cubeMeshId;   //Aberration shape
        Get & Set debug draw;
                                                        Float deformationX;
        Unsigned int GetVAO() const;                    Float deformationY;
        Getters & Setters vertex data;                  Float deformationZ;

        Void Load(Config& node) override;               Bool debugDraw = true;
        Void Save(Config& node) const override;
```

Add editor functionality: A button to add a scene node with an aberration component will be placed in the *Add* drop down menu, and the inspector will display and allow the user to modify the aberration data. Thanks to the ImGUI library (User interface library used in the render engine to manage all of the UI) these changes can be added very quickly.

Aberration previsualization: It is not possible to draw a mesh outline using the compute shader, so another render pass is needed in order to draw the aberration previsualization.

Directly drawing the outline of the shape using the raster rendering method would cause that the outline would be always completely drawn independently of the other meshes (if a box was placed in front of the outline, the outline would be drawn regardless as if it was in front of the box) which would reduce the usability of the level editor.

To solve the depth problem, using the raster render we can draw all the meshes of the scene in a completely transparent color with the depth buffer enabled, and then draw the aberration outlines. Since OpenGL does not check if the object is transparent before discarding it in the depth test, the outline will be drawn correctly when a mesh is placed in front of it.

Implementation

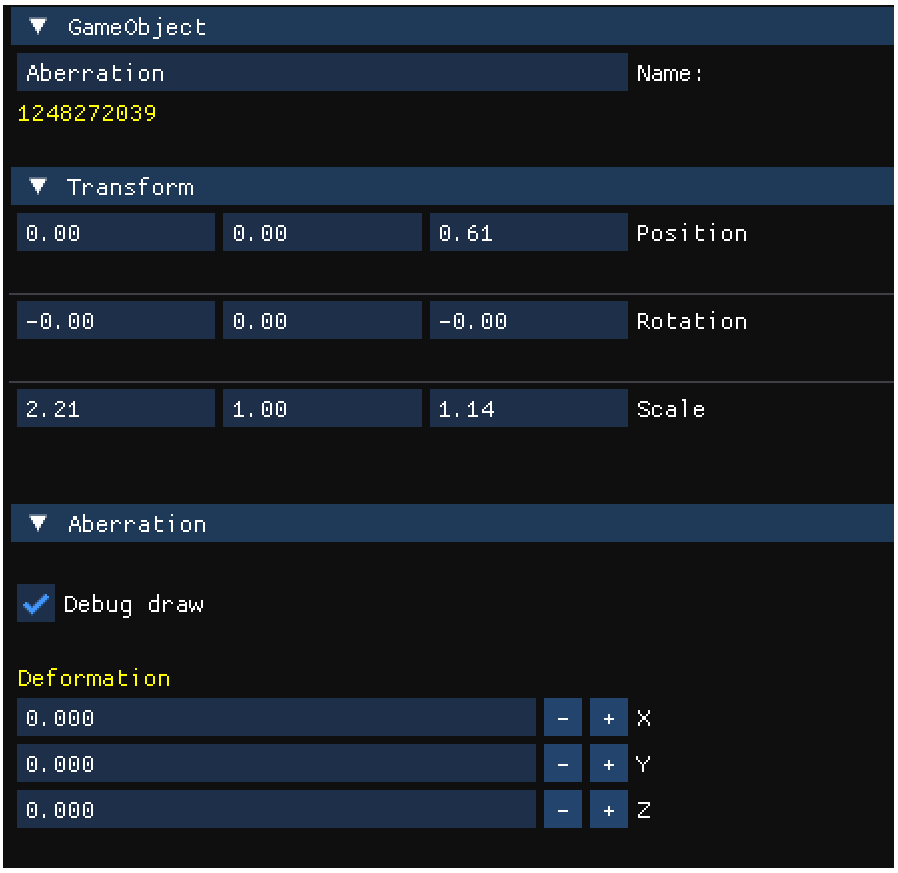
Create aberration component class: Created as a child of the component class, has an ID to the mesh resource that represents its shape (cube), this method can be easily changed in order to allow the user to select the shape of the aberration, but was not implemented due to time constrains and the inability to ensure FPS stability (aberration shape vertex count could impact the frame-rate more than the meshes).

The aberration would send a pointer of itself to the render module each frame which would be saved in a vector that is regenerated each frame in order to avoid accessing a destroyed aberration. This would be done in the *PreUpdate* method of the aberration component, because researching al the scene nodes to get all the aberration information was considered to create a considerable performance downgrade.

The aberration shape will be affected by the scene node transformation the same way a mesh is affected, allowing to move, rotate and scale it using gizmos or changing these values in the inspector.

Add editor functionality: Using the ImGUI functionality, the button to add a scene node with an aberration component was added, and a new collapsing header that displays and allows the edition of the aberration deformation direction and a checkbox to activate or deactivate the debug draw.

Figure 23 ***Aberration editor***

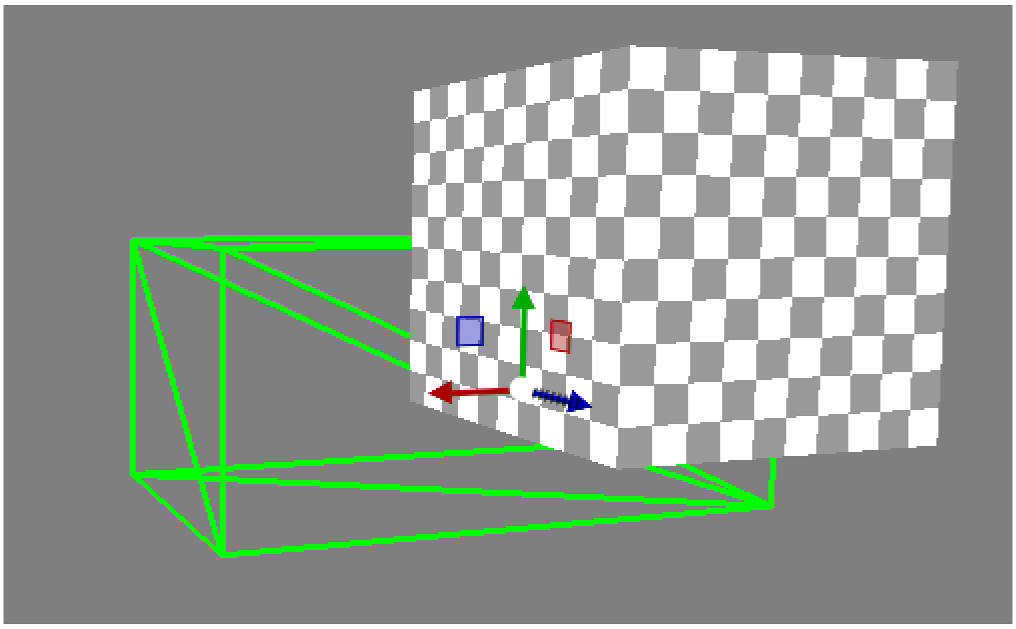

▼ GameObject
Aberration Name:
1248272039
▼ Transform
0.00 0.00 0.61 Position
-0.00 0.00 -0.00 Rotation
2.21 1.00 1.14 Scale
▼ Aberration
✓ Debug draw
Deformation
0.000 - + X
0.000 - + Y
0.000 - + Z

Here we can observe the inspector displaying the data of the game object that contains the aberration, it contains:

- Input text component to rename the game object
- 3 drag float components that allow to modify the x, y and z position.
- 3 drag float components that allow to modify the x, y and z rotation (in degrees).
- 3 drag float components that allow to modify the x, y and z scale.
- Checkbox to enable/disable debug draw.
- 3 input float components that allow to modify the x, y and z ray deformation.

Aberration previsualization: After the compute shader render pass is finished, the same texture is set as the target of a raster rendering pass in which the depth buffer is activated, in this render pass all the meshes of the scene are rendered with a shader that makes them transparent, and then the aberrations are drawn using the OpenGL line polygon model.

Figure 24 ***Aberration previsualization***



The area affected by an aberration is represented by the green lines, and we can appreciate that the depth perception is working properly since the upper-right corner of the aberration is not being drawn because there is a mesh in front of it.

Cleaning & Optimization

Since the aberration previsualization is using the raster rendering method, known for being the fastest rendering method, and the shaders used only calculate the positions of the vertices and directly paint the texture with the set colors (transparent for meshes and green for aberrations) the process is extremely fast and the performance has not been affected, even when using meshes of 5.500 polygons.

This result was expected because nowadays raster rendering engines are able to render scenes with 5 million polygons in real-time (Note that each of the meshes of the described scenes have around 3 to 6 textures, light, reflection and shadow calculations), so it is not unusual that it has no problems rendering small scenes with a low polygon count.

Validation

The engine currently has the features needed to create, save and load aberrations in the scene. These aberrations can be moved, rotated and scaled using the gizmos, and their information can be set through the inspector.

The previsualization method of the aberrations is working as expected and hasn’t affected the performance of the application.

Furthermore, the code is ready to work with meshes, the only functionality needed is to be able to change the mesh resource that the aberration is using. Despite being a feature out of the scope of the project it is a possible future expansion.

Development: Aberration ray deformation and collision detection

Design

Pass aberration data to the GPU: The aberration data will be treated similarly to a mesh, the aberration mesh data will be codified in the mesh textures (vertex and index textures), and the aberration data (transform, AABB info, vertex offset and deformation direction) will be send using a uniform variable.

Find nearest ray collision: Unfortunately, it will be needed to first test ray-mesh collisions and then test ray-aberration collisions, after comparing the distance between the two collided points, the closest one will be the collision point.

In case that the ray intersects with an aberration, a new ray shall be cast from the collision point, so we will need to calculate the collision point using the distance to the collision and the ray direction.

Deform the rays that collided with an aberration: Once the rays collide with the aberration, we will cast another one from the collided point with the deformed direction.

Even though there is no information about how the rays should be deformed, the question stands: *how are the rays deformed?* There are a variety of approaches that could be taken and "be correct" so that is not the focus of this investigation, the focus is in finding how can the rays be deformed in a way that will create the illusion that the space and the objects inside of it are elongated or compressed.

This led to the conclusion that a variety of methods should be implemented, documented and tested in order to select the most suitable to accomplish our desired effect.

- Once the ray collides with the aberration, cast the ray from the collision point, and add the deformation values directly to the ray direction, then normalize the vector.
- Once the ray collides with the aberration, cast a ray from the collision point, and compute a radial interpolation that makes the rays direction to get closer or further away from the center ray.
- When the camera is inside the aberration and the ray exits the aberration:
    - Deform the ray in the same direction.
    - Apply to the ray the inverted deformation.
    - Set the original ray direction previous to any deformation.

Implementation

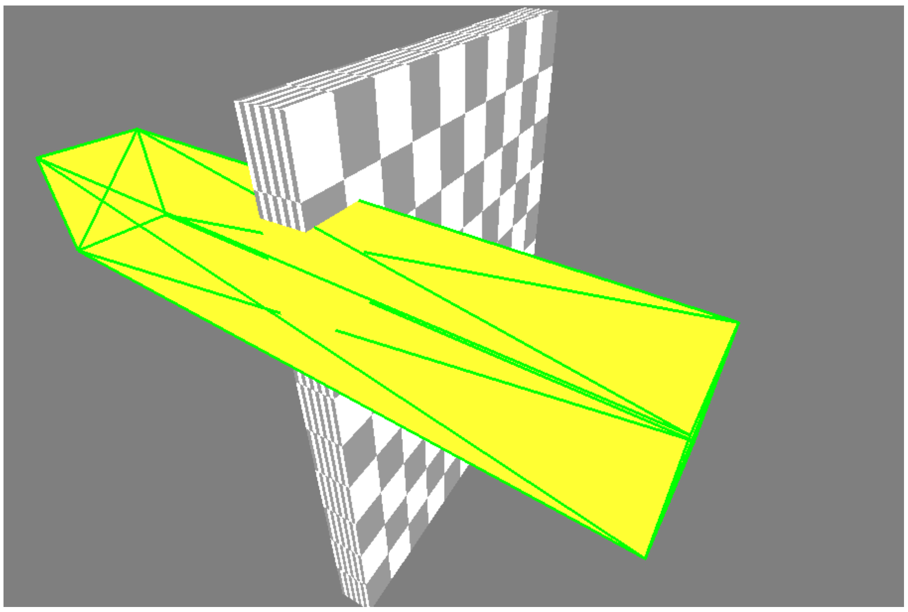
Pass aberration data to the GPU: Same approach as the meshes, the vertex and index data will be codified in the same textures as the meshes data, and their transform, vertex offset, triangle count and deformation direction will be send using uniform variables.

Find nearest ray collision: The method used to test ray-mesh collisions was adapted to create the method to check for aberration collisions.

Once the nearest collisions of each type are calculated, we will use the *step* GLSL function to multiply by 0 the values of the farthest collision. Even though this could be done using an if statement, those are to be avoided in order to minimize branching delays, calculating all values for 2 situations and multiplying by 0 the discarded one is a common practice in GPU programming because is faster than using an if statement to avoid calculating the discarded case.

To test that the nearest collision was being calculated correctly, if the closest collision was the aberration the output was set to be yellow.

Figure 25 ***Aberration and mesh collision test***

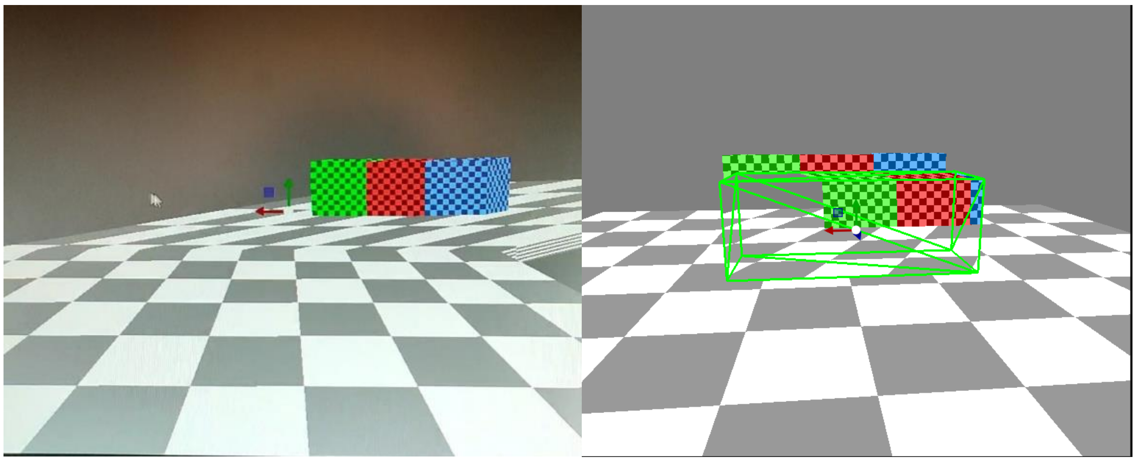


In this image we can observe that the collision detection is working properly, since the aberration (yellow) is completely enclosed by its AABB box and is being painted in front of the mesh, but the mesh is being drawn on top when it intersects with the aberration.

Deform the rays that collided with an aberration: As stated before, different approaches were tested in order to choose the one that produced the desired results and visual effects.

The first approach was to, once the rays enter the aberration, the aberration deformation would be added to the ray direction.

//TODO: add ilustration title

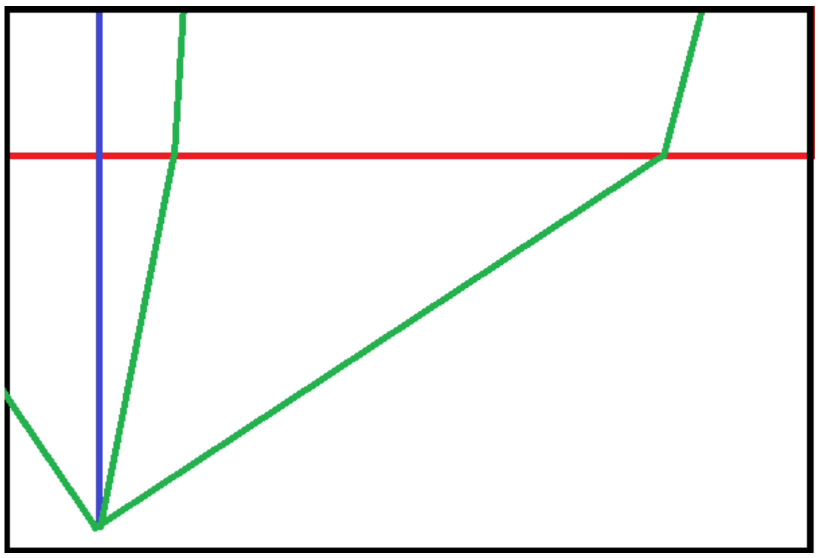


In this image we can appreciate how the aberration is unidirectionally deforming the space in the x direction, making the illusion that the elements of the scene are moved to the right.

Even though this approach does not allow the user to compress or expand the space, it creates an interesting effect that should be considered as possible addition to the application.

The second approach consisted on deforming the rays outwards or inwards relatively to the distance to the central camera ray.

//TODO add illustration title



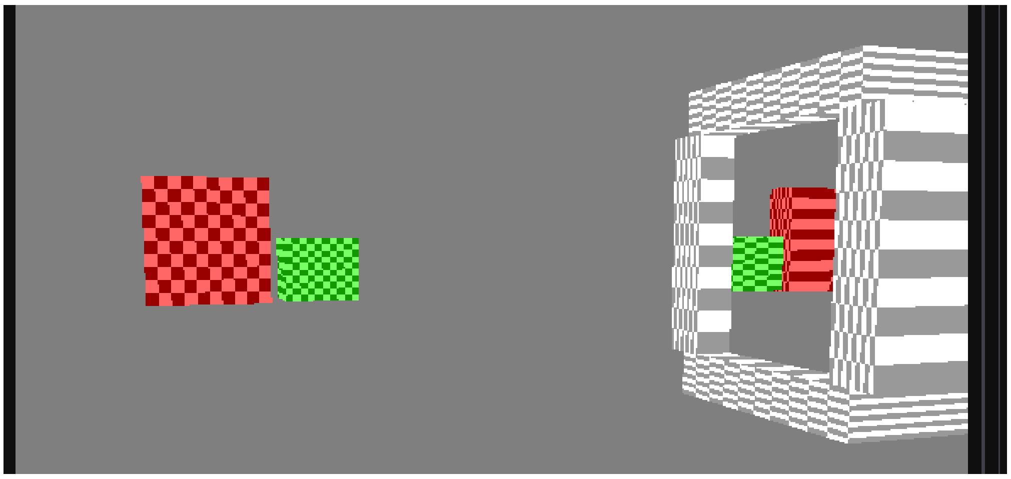
In the previous image we can see some camera rays (green) and the central camera ray (blue), which collide with an aberration (red) that is compressing the space. The deformation will not affect the central ray, but the deformation will increase the further away the ray is relatively to the central ray.

TODO: Add simple deformation image in x axis and y axis

In the left image we can observe an aberration compressing the space in the x axis, making the cube look slim, and in the right direction we can appreciate the same scene but the aberration is compressing the space in the y axis, which makes the cube less tall.

This method allows to create the effect of compressing and expanding the space with a great visual fidelity and interest and allows for a lot of flexibility when creating and editing a scene, but this flexibility causes a mirroring problem.

TODO: Add image title



In this image we can observe that there is an aberration at the entrance of a white tunnel, this aberration has a deformation value big enough to make some of the rays that collide with it bounce out the aberration creating a mirror effect. Solving this effect involves setting a limit to the deformation of the rays severely limiting the possibilities of the tool, and since it is possible to hide this problem using the layout of the scene (limiting the angles from which the aberration can be seen, use the same colors for the room and the tunnel so that the mirror effect is not noticeable…) it was decided to not limit the deformation of the rays.

Inside-outside aberration deformation: When the ray exits the aberration it should be deformed in a way that reinforces the illusion of the deformed space.

Firstly, the ray-aberration intersection method was modified in order to check if the ray was entering or exiting the aberration, which led to 3 approaches:

When the ray exits the aberration, apply the same deformation again.

TODO: Add image of this one

This led to a lot of confusion on where were the objects located when exiting the aberration and made the user feel like he was teleported instead of feeling like he was in a unified space with different densities.

When the ray exits the aberration, apply the inverted deformation.

TODO: Add image of this one

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