Architecting Virtual Worlds: A Study of Game Engine Development with Non-Euclidean Ambitions

Jôzef Carruthers - 2274599 - B.Sc. Computer Science
Supervised by Martin Escardo
9667 words - 40 credit project

Contents

1 Abstract

This project details the development and implementation of a game engine, a platform to host an environment with rendering and physics simulations, used to build games. Initially proposed to incorporate non-Euclidean geometry, due to time constraints, the project evolved to focus on producing a high-quality game engine.

The project began with a literature review on graphics rendering, game engine architecture, and non-Euclidean geometry. The development process involved initially creating the core rendering components using OpenGL, then input handling and physics simulation. Emphasis was placed on building a modular architecture to facilitate future enhancements.

While the proposed goal of integrating non-Euclidean geometry was not met within the project's timeframe, significant progress was made in creating a foundation for future development. Challenges faced while developing included managing memory and processor efficiency, integrating complex mathematics surrounding physical interactions of objects and making the decision between continuing work on the base game engine and implementing non-Euclidean geometry.

Overall, despite the absence of non-Euclidean geometry, the game engine shows solid functionality and performance, capable of rendering and simulating complex scenes efficiently. Its modular design and adherence to software development best practices facilitate future iterations and usage as a game engine.

2 Introduction

The development of video games stands out as both a creative and computationally-advanced field; there are a huge range of genres, from story-focused single-player games to competitive first-person shooters. They provide not only entertainment but also many other benefits, including cognitive stimulation, social interaction, as well as learning and development [3].

Behind many games lie game engines – software frameworks responsible for rendering graphics, managing assets, and handling gameplay mechanics – which often support multiple platforms, allowing games to be developed for several devices, enabling cross-platform compatibility [22]. An example is Unity, which is compatible with most of the popular platforms, including Xbox, Windows OS, and Android [4].

The original aim of the project was to focus on producing a game engine which implements non-Euclidean elements, such as portals. The field of game engine development is constantly progressing, due to the ever-growing demands of modern game design. Challenges such as rendering of life-like scenes, and the growing complexity of world building means that we can't solely rely on the improvements of graphics hardware. One solution to this is improving the efficiency of game engines, which I have explored throughout the development of this project. Whilst non-Euclidean geometry is theoretically explored within this project, time constraints resulted in the project focus being shifted to designing an original and efficient game engine that could later be used to support non-Euclidean elements.

Traditionally, games have adhered to Euclidean geometry, wherein parallel lines remain equidis-

tant, and the angles of a triangle always sum to 180° [11]. However, the use of non-Euclidean geometry produces innovative environments with warped spatial properties. Non-Euclidean geometry has the potential to open up new avenues for artistic expression and gameplay, such as traversing corridors that impossibly loop back onto themselves, or navigating surfaces that curve and distort in ways that seem incomprehensible.

This project begins with a literature review providing a theoretical background of game design and non-Euclidean geometry, within the fields of Computer Science and Maths [21]. The next section details decisions regarding the use of OpenGL (a graphics library), rather than a pre-built game engine [4]. The development process is then outlined and described, and documentation provided in order to show how the aim of this project - building an efficient game engine - was achieved. Naming conventions and file structure are explained and justified, in order to provide further understanding and clarity surrounding the methods created for this study. Conclusions regarding the successes and limitations of the project are then detailed, with recommendations for further research outlined.

3 Literature Review

3.1 Introduction

The following section will start by exploring literature around early models of non-Euclidean geometry, including representation in physical form, rather than software. Following this, the section will look at recent papers in order to contextualise where the area of non-Euclidean game development currently stands, and how the field could be further developed. Whilst producing this literature review, I was working under the assumption that my final game engine would involve some form of non-Euclidean component but this was not possible within the time frame.

In order to find sources, the Google Scholar electronic database has been used, along with key words including the following:

- Non-Euclidean geometry
- History of game engines
- Visualisation of non-Euclidean spaces
- Non-Euclidean geometry in 3D games
- Euclidean geometry
- Euclid's elements

When analysing sources, I have used a scoping review to identify resources which relate to the fields of game design and non-Euclidean geometry. When choosing sources, I looked at the aims of the project, strengths and weaknesses, and how the learnings from these papers can inform my work. I will start by looking at the wider field of both non-Euclidean geometry and game engines, before looking in more depth at examples which relate more closely to my own work.

3.2 General literature

To look at non-Euclidean geometry, we must first explore its counterpart. Euclidean geometry was first conceptualised by early Greek mathematicians, but was grouped together into a series

of axioms by Euclid in his series of books, Euclid's Elements [11]. It considers the properties of points, lines, angles, and shapes across both two and three-dimensional space [11].

On the other hand, non-Euclidean geometry does not follow these axioms, instead the parallel postulate is not considered leading to distortions, of which the most notable being spherical and hyperbolic [20]. Early representations of non-Euclidean geometry include mediums such as engravings and crochet [23], and it was only in 1992 that Geomview (one of the first software based models) was developed [24] [18]. Non-Euclidean geometry is explored in further detail in section 4.1.1 of this document.

Game engines originated as a way to separate core functionality from the components which make up the content of the game itself, which gives the ability to reuse this core functionality for multiple games [17].

3.3 Central literature

Here, I explore three main papers which relate specifically to the aim I set out to achieve: creating a game engine which implements non-Euclidean geometry. I compare the papers, looking at strengths and weaknesses, including methods I could consider implementing within my own project.

3.3.1 Key findings and critiques Game engines were first developed in the mid-1990s [2] and, as such, there are various papers [20] [6] showing prior development. Particularly relevant is Osudin (2022)'s study which states its aim is to present "an implementation of a 2D non-Euclidean physics and graphics engine". This paper specifically looks at creating a game engine which allows for the space itself to be curved, and how this manipulates two-dimensional shapes within this space. Alternatively, Pascua et al. (2023)'s study discusses a 3D implementation, whereby portals are used to transport the user across a space. This paper focused on the creation of a singular game, rather than a game engine which would be more flexible, and allows for development down many avenues [17].

In another study [24], the intention was to allow users to experience the space using virtual reality in order to give "a better sense of being immersed." Although this isn't a game, several methods used by this study were interesting considerations. As a rendering method, this paper made use of ray tracing, a way to determine which components of a world are currently in view [19]. Although it is generally perceived as a more realistic method of rendering due to its ability to represent shadows and reflections, it is also computationally expensive and not as efficient as other methods (Navrátil, 2010). Efficiency is important within a game engine due to the many components which may make up a game, so it was important to look elsewhere. Alternative methods of real time rendering include splatting - "representing the volume as an array of overlapping base functions" - and shear-warp - clever in that it slices the data, rendering each part individually, and aligning with the current viewing direction to improve rendering speed [15]. Of the methods discussed in Huang et al (2000)'s paper, the fastest renderers are recognised to be shear-warp and 3D texture mapping, whilst raycasting came last.

Additionally, Navrátil (2010)'s paper does not take into consideration time, which is particularly important for a game engine, as different instances of the game engine will otherwise run at different rates, meaning there will be no consistency [12]. Looking at how the execution of this game

engine has been tested, I think it is important to similarly consider the efficiency of the engine in my own project. Since games often have intricate worlds which introduce many individual components, it is important to ensure that increasing complexity has minimal impact on the machine's hardware.

3.3.2 Focused analysis of key systems and experiments Having analysed these key papers, I have been able to identify a research gap regarding the representation of non-Euclidean components in a three-dimensional space within a game engine. The idea being that creation of non-Euclidean elements in a game engine makes their use more accessible, allowing for the development of more games in this field.

3.4 Conclusion

This review briefly explores early visualisations of non-Euclidean geometry, before diving deeper into modern software developments. Ultimately, I have decided that gaps in this field lie with regards to the production of a game engine for three-dimensional representations of non-Euclidean geometry.

4 Method / Approach / Specification

4.1 First steps

My first step in the project was to conduct some research into non-Euclidean geometry in order to deepen my understanding of it. The below section details what I found.

4.1.1 Non-Euclidean geometry Non-Euclidean geometry is any geometry that does not follow the rules of Euclidean geometry. It often refers to hyperbolic and elliptic/spherical geometry. Euclidean geometry has zero plane/space curvature, whereas hyperbolic has negative curvature and spherical/elliptic geometry has positive curvature, other geometries may have varying curvatures.

Imagine 3 lines, two parallel to each other and the third perpendicular to these two. In Euclidean geometry, the first two lines would stay at a constant perpendicular line distance from each other, however, in non-Euclidean geometry, this distance changes [7]. In hyperbolic geometry these lines will curve away from each other, and in elliptic geometry the line will curve towards each other [7].

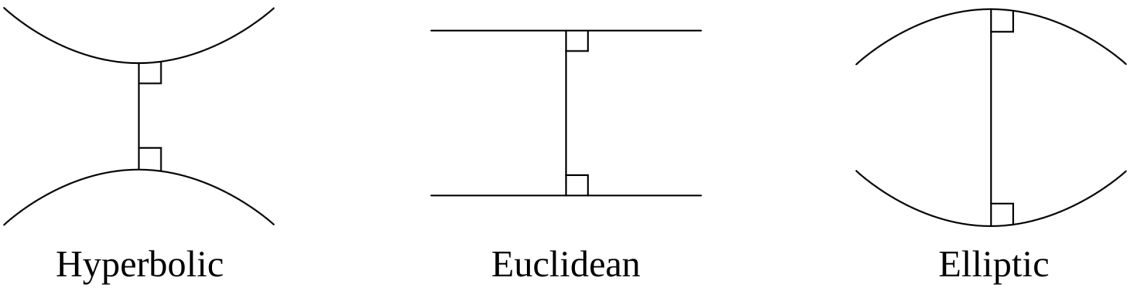

Figure 1: Curvature of different geometries (source: https://glumpy.readthedocs.io/en/latest/tutorial/cube-ugly.html)

After some research on non-Euclidean geometry, I began by first looking at different ways I could try to render it. My options were to either use an existing game engine that had an adaptable rendering system or write my own game engine and implement non-Euclidean geometry myself as part of its renderer.

After conducting research into game engines I could use, Unity seemed like the best option, as its rendering pipeline was fully transformable [8]. I started by writing some basic scripts for player movement and camera control and prototyped a basic scene to explore. Next, I started looking into Unity's render pipeline and at the same time continuing research into non-Euclidean geometry. Given the scope of non-Euclidean geometry and Unity's complex render pipeline, I gave myself the goal of rendering portals as they would be the simplest way to include non-Euclidean geometry.

After some time prototyping this, although it was possible to use Unity for my project, I ultimately decided that I would much prefer going down the route of writing my own game engine. The reason I had decided to do this was that I had full control over every aspect of the game engine, including rendering, and I only had to include exactly what I needed, whereas Unity had a fair amount of additional complications. Since Unity has far more functionality than would be required for my project, I decided starting from scratch would make it easier to improve efficiency. Also,

the end product would have just been a large modification of an existing rendering system which would have taken a while to learn given Unity's rendering complexity, so instead of doing this, the idea of creating a smaller scale game engine from scratch was much more enticing.

Making the decision to write my own game engine led me to further research. From this, I found OpenGL (a graphics library) to be most appropriate for my game engine; especially considering I have previously made a basic pseudo 3D ray casting renderer, similar to the ID Tech 1 game engine, using this library. Given that I wanted to make a further developed game engine than I had previously, I looked into learning OpenGL at a more advanced level, at which point I found the online resource learnopengl.com [9]. To begin with, I used this online resource to aid me in creating the base code for rendering with OpenGL, which I then went on to use in my game engine.

4.2 Overview

After laying the foundations, I then continued with other aspects of my project. My game engine revolves around a simulated '$World$', which encapsulates '$Objects$'. This $World$ is updated within a main while loop, each update clearing the screen, applying any changes to the $World$ and then displaying the state of the $World$, which can be referred to as a frame. Given that there are many frames displayed per second, this gives the effect of smooth movement.

The next few sections will describe the classes of my game engine and their fields and methods, and acts as documentation for use of the game engine but also explains the reasons behind some choices made.

4.3 *Object*

The following describes the fields and members of the *Object* class respectively. In this document, '*Object*' refers specifically to this *Object* class, I have done this as things can get quite confusing, often referring to objects of different contexts.

4.3.1 *objID* This integer field is employed for *Object* identification. When an *Object* is instantiated, the $objID$ field is set as the amount of *Object*s in the $World$ incremented by 1.
4.3.2 *objScale* This is a $vec3$ (a type used to represent 3 floating point numbers ($float$s) from the OpenGL Mathematics header file "glm.hpp") field storing the *Object*'s size in $World$ units (denoted as U_w), defaulted as $\{1, 1, 1\}$, where each $float$ is the scale along the x, y and z axes respectively.
4.3.3 *objPosition* The coordinates of the center of the *Object* in $World$ space are stored in this $vec3$ field.
4.3.4 *objRotation* Defining the rotation of the *Object*, this $vec3$ field uses Euler angles where each x, y and z component of the $vec3$ represents the orientation of an *Object* in three-dimensional space by specifying rotations around its principal axes.
4.3.5 *objVelocity* This is a $vec3$ field used for the velocity of an *Object*. The $objPosition$ field is incremented by this every second.
4.3.6 *objAngVelocity* Similar to $objVelocity$, this field is a $vec3$ used to increment $objRotation$ every second.

4.3.7 ***model*** This field is of the type *mat*4 (a 4 dimensional matrix), encapsulating the *Object*'s position, rotation and scale. This is used in addition to other matrices discussed later on in order to render the *Object*.
4.3.8 ***shader*** Serving as a pointer to a *Shader*, this field is used to specify how this *Object* should be rendered. *Shader*s will be explored subsequently. I have used pointers here as shaders hold a lot of data, and in game engines - including mine - often have shaders which are used for many *Object*s, so instead of holding a copy of the same shader on each *Object*, the shaders are initialised first, and then the *Object*s just hold a pointer to them.
4.3.9 ***texPath*** *texPath* is a *string* field used to store the file path (relative to the base folder) of a texture to be drawn onto the faces of the *Object*.
4.3.10 ***properties*** This field has the type *ObjectProperties*, it represents two *boolean* values which determine whether the *Object* can be collided with and whether the *Object* is affected by gravity. The reason a type was created just for this purpose instead of just using a bool array was that each property can be given its own field name, making further development easier as if at some point *Object*s are to have many properties, accessing each one doesn't require memorizing which property is which element of the array, as they each have their own name.

Shown below are the methods of the *Object* class.

4.3.11 ***SetPosition*** This method has one *vec*3 parameter, which is used to update the *Object*'s *objPosition* field and returns a void type (it returns nothing).
4.3.12 ***SetRotation*** This method of type *void* serves to set the *objRotation* field to the given *vec*3 parameter.
4.3.13 ***SetScale*** This takes a *vec*3 parameter used to update the *objScale* field and return nothing.
4.3.14 ***Translate*** This method of type *void* - using *SetPosition* - translates an *Object* by a given *vec*3 parameter (the sum the translation parameter and *Object*'s current position is set as the *Object*'s new position).
4.3.15 ***Rotate*** Serving to rotate the *Object*, this method adds its *vec*3 parameter to the *Object*'s *objRotation* field and then sets it using *SetRotation* and returns nothing.
4.3.16 ***Grow*** This method of type void sets the *objScale* field as the sum of its *vec*3 parameter and the current scale of the *Object* using the *SetScale* method. Given its name and purpose I was not sure whether to use addition or multiplication between the *Object*'s current scale and the method parameter, however given the other setting methods I chose to use addition to remain consistant.
4.3.17 ***GetVertices*** Used in collision detection, this method returns a *vector* (in this case the *vector* is a list from the standard c++ header 'vector.h') of *vec*3s which represent the coordinate of each vertex of the object in *World* space.

4.4 *World*

The following describe the fields of the *World* class.

4.4.1 ***objects*** This field holds a vector (list) of pointers to *Object*s, representing the *Object*s in the *World*, the reason pointers are used here is again for memory efficiency.
4.4.2 ***objCount*** Of type int, this field is used to keep track of how many *Object*s are in the *World*.

4.4.3 ***gravityStrength*** This field is a $float$ describing the strength of gravity (in $U_w s^{-2}$, s being seconds) in the $World$, used in the method *ApplyGravity* described below.

Below shows the methods of the $World$ class.

4.4.4 ***AddObject*** Taking a pointer to an *Object* as a parameter, this method of type void adds the *Object* pointer to the *objects* field.
4.4.5 ***ApplyGravity*** This method takes one $float$ parameter 'deltaTime' (which will be discussed later in the $Time$ class section) and returns void.
4.4.6 ***HandleCollisions*** Again taking a $float$ parameter $deltaTime$, this method considers the positions of all of the *Object* vertices in the $World$ and if any are colliding with another *Object*, handles that accordingly. This is explored in more detail later in the Collision detection section. Unusually, *Object*s velocities are applied here, this is because this is closely linked to whether they are colliding or not.
4.4.7 ***Update*** This void type method takes a $float$ parameter $deltaTime$ and uses it to call the methods *ApplyGravity* and $HandleCollisions$. It is hierarchically at the top of the $World$ class, it is the method that is called from outside the $World$ class, once per game loop.

4.5 *Time*

This class handles everything regarding time and rates of events in my game engine. Below show the fields of this class. They are used more as variables for the methods of the $Time$ class with elevated scope rather than fields to be accessed outside the class. Below shows the fields of the $Time$ class.

4.5.1 ***currentTime*** This is a *double* (double precision floating point number) acting as a timestamp for the $deltaTime$ calculation.
4.5.2 ***frameCount*** This field is used in the calculation of frame rate in the $GetAvgFrameRate$ method, it has the type *double*.
4.5.3 ***limitFrameRate*** Of type *bool*, this field is used as a flag to represent whether the frame rate should be limited or not. It is set in the $LimitFrameRate$ method and used in the $Update$ method.
4.5.4 ***targetFrameRate*** This *double* field is used as the target frame rate to be acheived when limiting the frame rate.
4.5.5 ***targetFrameTime*** This field of type *double* is the reciprocal of $targetFrameRate$ and is also used when limiting the frame rate, representing the target time for 1 frame to last.
4.5.6 ***deltaTime*** This field is common to many game engines. It helps ensure that the passage of time within the game engine remains constant no matter how fast the game engine runs. Without its usage, if a game engine is run on a faster system, things will naturally run faster and if more calculations are needed for some frames this would seem to make things look slower. To keep things consistant, $deltaTime$ is used. $deltaTime$ itself is the amount of time it took between the last frame and the current frame.
An example usage could be an *Object*'s movement, where if an *Object* has some velocity, this is added onto its position each frame, but the added amount is multiplied by $deltaTime$. This means if more time was taken between frames, the *Object* is moved more, resulting in a consistant smooth movement on the screen, which is why a lot of functions take $deltaTime$ as a parameter. In retrospect, I could have made time a global class or added a field to *Object* which points to the $Time$ object, so I didn't have to pass $deltaTime$ so often, but it does not cause any issues as is. In short,

deltaTime is a way of syncing up real world time to the inconsistant frame time of the game engine.

The following detail the methods of the *Time* class

4.5.7 ***Update*** This method of type void takes no parameters and is called at the start of every frame. First it gets the current time, and finds the difference between this and the last time it was fetched. Then, it increments $frameCount$ and, if $limitFrameRate$ is true, it calcultes the difference between the $targetFrameTime$ and $deltaTime$. If this value is greater than zero, the 'thread' standard library is used to pause for the difference in time. This leads to every frame taking at least $targetFrameTime$, which means $targetFrameTime$ is always acheivable if the execution rate is faster than the target rate. The only way this is not acheived is if a system is not able to run the game engine well, which can only be improved by making the engine more efficient.
4.5.8 ***GetDeltaTime*** This is a basic get method, returning $deltaTime$ and taking no parameters.
4.5.9 ***GetAvgFrameRate*** This is another get method, taking no parameters and returning frame rate by dividing frame count by the elapsed time of the game engine (using OpenGL's $glfwGetTime$ function).
4.5.10 ***EveryNSeconds*** This method takes a *double* parameter N where every N seconds, this function returns true, otherwise false. It is used to orchestrate events that occur on a regular basis.
4.5.11 ***GetTime*** This basic get method takes no parameters and returns $currentTime$.
4.5.12 ***LimitFrameRate*** This helper method aids in limiting the frame rate by setting up the frame rate limiter in $Update$. It first sets the $limitFrameRate$ to true. Then using its int parameter of the target frame rate it sets the field $targetFrameRate$ to this parameter and then $targetFrameTime$ to $1/targetFrameRate$. By default, this is called, setting the frame rate limit to be 60, which is very often what most screens have a limit of rendering. This ensures that the game engine isn't needlessly using as much hardware resources as possibe to get the highest frame rate possible.

4.6 Collision detection

Detecting whether two *Object*s are colliding (touching or inside) each other was a greatly time-consuming part of my project, second most so to rendering. At a more general scope, the way my game engine detects collisions is as follows; each frame, the function *Update* is called from the *World* object, which involves applying gravity to the *Object*, checking if the vertices of an *Object* are within the bounds of another, and then applying the velocities if not, otherwise setting the velocities of the two colliding *Object*s to $\{0, 0, 0\}$.

After some failed attempts at prototyping this, I came upon the separating axis theorem (SAT).

4.6.1 Separating Axis Theorum (SAT) This theorem detects collisions between two objects by projecting both objects onto planes made from each of the faces of the two objects [5]. If there is a projection that exists which has no overlap, there is a separating axis. This equates to trying to find a plane which has no intersections between the two objects. See the below figure.

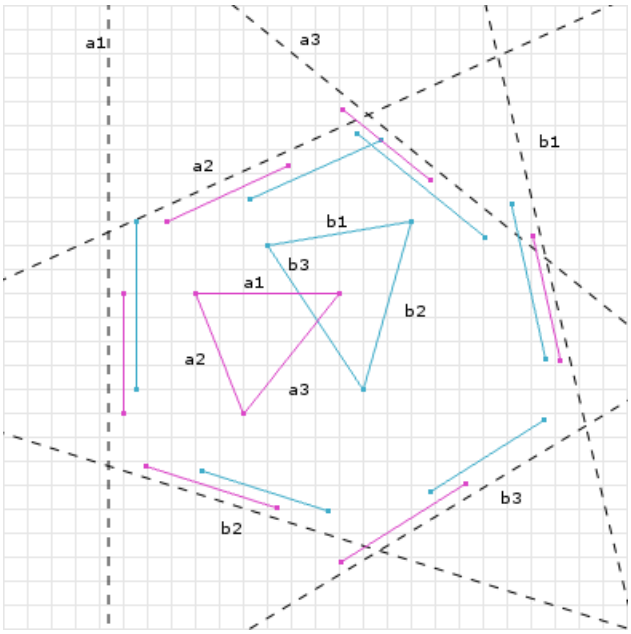

Figure 2: Using SAT to detect a collision between two triangles (source: https://dyn4j.org/2010/01/sat/)

4.7 Rendering

As mentioned in previous sections, my game engine uses OpenGL to render the world to the screen. This was the most complex and time-consuming part of my engine. To generalise the rendering process, my engine projects the *Object*s onto a plane in front of the camera (explained later in this section), and then this plane is rendered to the screen.

As the concept of rendering has been commonly explored in game engine development, in this section I will not go into as much detail into individual fields and methods of classes developed for these processes. Instead, I will explain these concepts at a higher level, providing examples of their

uses within my game engine.

4.7.1 Early rendering At the very start of the project, the first goal I had was to further my understanding of OpenGL, so I started with trying to draw a basic 2D object to the screen. Doing this was initially very simple, just requiring a few OpenGL functions. After this, things start getting more complex. My next goal was to implement shaders.

4.7.2 Shaders In graphics rendering, shaders are computer programs that are used to calculate the colour of pixels on the screen given the state of what is being rendered (for example the coordinates of an *Object*s vertices) [14]. In OpenGL, shaders are written in GLSL (OpenGL Shading Language) which is a programming language based on C creating specifically for writing shaders.

Shaders effectively just take inputs and return outputs. Given their flexibility, shaders can be written for many uses. In OpenGL, two kinds of commonly used shaders are 'vertex' and 'fragment' shaders [14]. Vertex shaders are used to transform the geometry of the scene into a projection, effectively laying out what shapes should be displayed to the screen. Fragment shaders define the colour of each pixel within these shapes.

Initially, while shaders were simple, the class in my code for shaders was also quite simple. Since shaders are given to the OpenGL context in the form of a string, it became quite time-consuming and made the code less readable to write the shaders in the same file as they are passed to OpenGL. Because of this - as is common for many OpenGL based game engines - I included a shader parser.

4.7.3 Shader parsing Reading a shader from a file, instead of writing it directly in a string in the code, involves using C++'s file reader 'ifstream' and string builder 'stringstream'. The parser first looks for the tag '#shader' in the shader file. If found, it will then check the next word on the line; if it is 'vertex' it tags the shader as a vertex shader, and similarly with 'fragment'. It then uses stringstream to add the next lines into a string, until it reaches another shader tag or the end of the file.

The next step I took was to develop rendering complexity, by going from using basic OpenGL functions to draw shapes to using arrays of vertices. To do this in OpenGL, a 'vertex buffer object' (VBO) is required.

4.7.4 Vertex buffer object (VBO) VBOs have the purpose of storing vertices in the GPU's memory. The buffer object is initialised, bound to the OpenGL context, and then filled with vertices to be drawn [16]. A single vertex can hold more data than just the position, for example, a vertex can also contain colour values (vertex colours are often used to be interpolated between on the faces of an object, to give a gradient effect). As a result, OpenGL needs more information about how the VBO is laid out. This information is provided by a 'vertex array object' (VAO).

4.7.5 Vertex array object (VAO) VAOs are bound and populated just like VBOs. They specify how data is laid out in the VBOs, defining the layout for each attribute (e.g. position, or colour), using offset, size, type, and stride [16]. Offset describes how many bytes there are from the start of the VBO to the first element of an attribute in the VBO. Size is used to describe how many elements an attribute has. Type represents the data type of the elements of an attribute. Stride is the amount of bytes between the starts of consecutive attributes in the VBO.

As an example, let's say there is a VBO defining a buffer of vertices, where the first three elements of each vertex define the $\{x, y, z\}$ position of the object, and the last four define the colour of each vertex in RGBA format (4 values describing how red, blue, green and transparent a colour is). To describe the layout of this vertex, for position and colour, A VAO can be created and two layouts can be added to it. The position layout describes positions of the VBO as having offset 0, as the first element of the VBO is a position value, size of 3 as there are 3 values describing position, type as $float$, and stride as $7 * sizeof(float)$ where $sizeof(type)$ is a C operator that returns the size in bytes of a given type. The same applies for colour, only difference being the size, which will be 4, and offset which will be $3 * sizeof(float)$, representing how many bytes until the first colour element. These two layouts can be added to the VAO.

4.7.6 Textures Sometimes developing a highly complex fragment shader to give objects some detail isn't the best option. Textures allow game engines to draw images to the faces of objects [10]. In order to draw textures to objects in OpenGL, information about how the texture is to be applied to the object needs to be provided. This is done by adding texture coordinates to the vertex data of an object. For each vertex, 2 values are given representing the x and y coordinates. These form a mapping of the texture for each face of the object.

Textures can be very high resolution images if objects need lots of detail, but if there are many objects in the scene, this can lead to a lot of VRAM (video RAM, which works just like normal RAM but is specific to the GPU). OpenGL does have a built-in solution for this, which is known as minimapping. Minimapping creates a collection of textures from a single texture, decreasing in resolution. These are used instead of the base texture when viewed from far away to increase memory efficiency.

After getting a 2D object successfully rendered using vertex shaders, fragment shaders and textures, the next step was to implement 3D rendering.

4.7.7 Camera To render in 3D, there must be a point from which *Object*s are observed. In my game engine, this observer is denoted as the 'camera' as it captures what is infront of it, in order to be displayed to the screen as an image. The camera has its own matrix, named the 'view' matrix, describing the position and orientation of the camera in world space. This matrix is used to transform the *Object*s in the world to be relative to the camera. This process of transforming coordinates to be relative to different spaces is explored further in the following section.

4.7.8 Coordinate systems In order to get from various points in the world to an image on the screen, the notion of differing coordinate systems must be considered. For example, the model matrix of an *Object* translates the coordinates of the vertices of a basic cube of side length 1, centered at $\{0, 0, 0\}$, to the coordinates of the vertices of the *Object* in the world.

The view space is next. This is the world transformed by the view matrix of the camera, changing the *Object*s from being relative to the world origin to being relative to the camera.

Finally, these points must be projected onto a 2D plane, as the screen itself is 2D. This is done with a projection matrix, effectively casting a frustrum out from the camera, capturing *Object*s in the world and squashing this down onto a plane. This is visualised in the figure below.

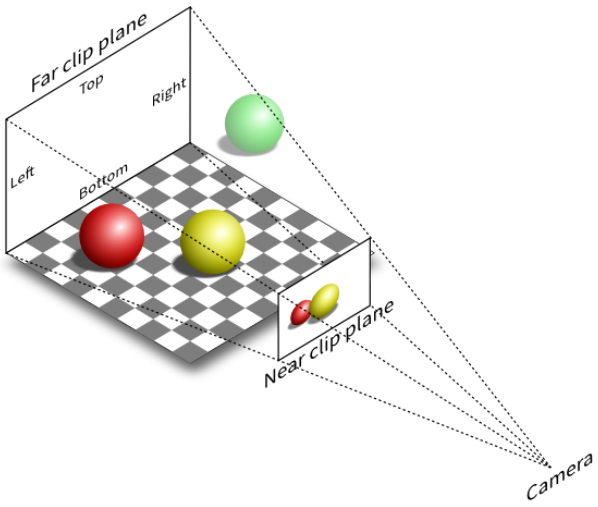

Figure 3: Projecting spheres to a camera (source: https://glumpy.readthedocs.io/en/latest/tutorial/cube-ugly.html)

4.8 Input handling

To be able to explore the world, the user must have some way of interacting with the system. In my game engine, I created a class in order to handle keyboard and mouse inputs and convert those inputs into interactions in my world. In general, the input system uses W, A, S, and D keys for forwards, left, backwards and right movement of the camera respectively, and the mouse controls the direction in which the camera is pointing. I have chosen these inputs as it is how most video games are controlled [13].

I had an issue with the inital input system not being able to handle more than one key at a time, meaning the system was not able to handle diagonal movement, as this required a forwards/backwards key as well as a left/right key. To solve this, I had to completely rewrite my input system.

The way it now works is by detecting key up and key down events. When a key is detected as being pressed down, it is added to a list, and if it is then released, it is removed from the list. I then added a function to check whether a given key was in that list, allowing for multiple key presses at once. This, however, led me to another issue. The way I had set this up was by applying movement for each pressed key, meaning for diagonal movements, the resulting movement of the camera is greater than moving in a single direction. I solved this by, instead of directly applying the movement in the checks for key presses, creating a movement vector which is added to within each check to result in a combined movement vector. If exactly two keys were pressed, this vector is then divided by the square root of two. This works since movements in each direction have the same magnitude, and are all at right angles to eachother, so if a diagonal movement occurs (when two keys are pressed), it can be described as a right-angled triangle. To reduce the hypotenuse (diagonal movement vector) to have the same side length as the two identical perpendicular sides, it just needs to be divided by the square root of two.

4.9 The result

This section discusses the current state of my game engine, exploring how I combined the concepts of the previous sections into an explorable 3D environment with physical *Object* interactions. First, there is a world, with a camera and some *Object*s. Below lists the steps of the main while loop:

1. Time class is updated

- **1.1.** Current time is fetched
- **1.2.** DeltaTime is calculated
- **1.3.** FrameCount is incremented
- **1.4.** Limits frame rate if appropriate

2. User input is processed

3. Screen is cleared

4. Modifications to the world can be made

4. World is updated

- **4.1.** Gravity is added to the *Object* velocities
- **4.2.** Check *Object*s for collisions
- **4.3.** Applies velocities to *Object*s that aren't colliding

5. World is drawn to the screen

- **5.1** Using camera position and orientation, view and projection matrices are calculated
- **5.2** Then, for every *Object* in the world:
  - **5.2.1** Their shader is bound to OpenGL
  - **5.2.2** This shader is given the view and projection matrices, as well as the model matrix of the *Object*
  - **5.2.3** The vertex shader then combines these matrices to calculate the vertex coordinates of the *Object* on the screen
  - **5.2.4** These points are passed into the fragment shader, which combines a given colour and texture to draw the *Object* to the screen.

6. This will result in all *Object*s being drawn to the screen.

The resulting game engine is shown in the below figures, where it first shows of an object above another, and then it shows the same object after it had fallen (the captures taken were just from different angles, the larger object was static throughout the engine run):

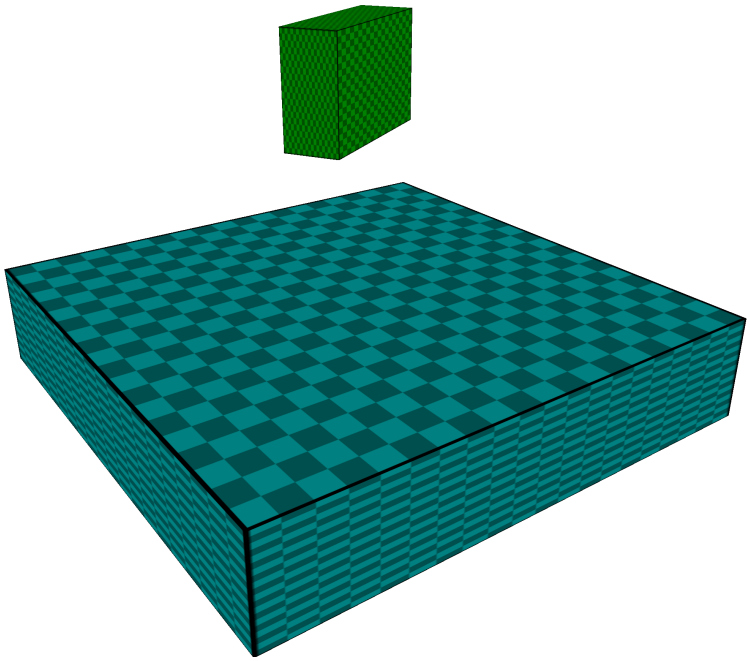

Figure 4: Object collision (before)

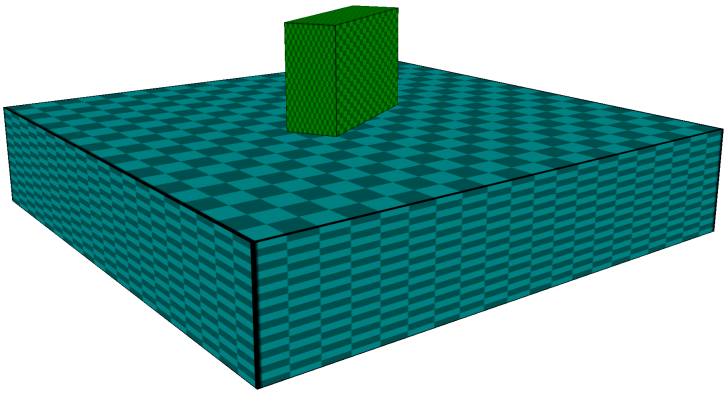

Figure 5: Object collision (after)

4.10 Code design

Throughout development of my game engine, I kept the goal in mind of having good code design. This involves having good file structure, method and field names, code structure and file names, and staying consistant with how I choose to approach each of these points. In this section, I will go over these code design attributes, discussing what I did to achieve them.

4.10.1 File structure To have a logical file structure, I adhered to the following rules. I created the subsequent folders for the purpose of separating out files for their own specific purposes. First was the 'src' folder,

- *Dependencies*: This folder holds all external dependencies that my code relies on: 'GLEW', 'GLFW', 'glm', 'stb_image'.
- *NonEuclidRenderer*: This folder holds the resources of the project that I have included.
  - *res*: This folder is specific for non-code resources.
    - *shaders*: This folder holds the GLSL shaders used in rendering.
    - *textures*: This folder holds PNG images used as textures in rendering.
  - *src*: This folder holds the code that makes up my game engine.
- *Notes*: This folder stored any notes made about the project along the way.

4.10.2 Field and method naming This was fairly simple to stay consistant with. I made sure to use camel case for field names, where the first letter of each word in a field name is capitalised, except the first word. An example of this would be 'exampleFieldName'. For methods and class names, I used Pascal case, which is similar to camel case but the only difference is that the first letter of the first word is capitalised. An example of this would be 'ExampleMethodName'. For the few constants and defined values, I would use a fully capitalised snake case. An example of this is 'DEFINED_VALUE'. Doing so helps with identifying code for myself and others, as it maintains a consistant and predictable naming convention throughout the codebase.

4.10.3 Code structure A big focus of code design was on code structure. This involved having 2 files for every large class, these two files being a header and source files ('.h' and '.cpp' respectively). I used header files to store imports and class definitions, listing the method and field names, and types, and then wrote the code for these functions in the source file. Any smaller classes went in the class files that they are most relevent to. This made the code much easier to read through. As an example, 'Time.h' will include any necessary header file imports, and the definition for 'Time' class - its field and method names, whilst 'Time.cpp' is used to write the code within the methods of the time class, effectively pointing to the laid out methods in 'Time.h' file and describing how each of the methods work. This was to keep large functions out of class definitions so, if a developer is looking at how a class is laid out, they won't be bombarded with big chunks of code they are not looking for.

4.10.4 File naming For my file names, I made sure to always use Pascal case, and ensured that the names of the file best matched their contents. Also, for a pair of files describing the same class (a '.h' and '.cpp' pair), the file names are exactly the same, with the only difference being the file extensions. This was to get across the point that they are directly linked to eachother, both being used for the same purpose, each as a part of describing a process in my game engine. For example, for the 'Time' class, all the code is contained within 'Time.h' and 'Time.cpp'.

4.10.5 Source Control As I was developing on my laptop, PC, and lab machine, I did not want to have to keep my work stored on an external drive. Another issue was that if I made any large mistakes in my code, I had no way of reverting to an earlier state unless I was to keep regular backups on one of the computers I was working on. To solve this issue, I looked into source control. The source control method I used was 'GitLab', which enabled me to store my code online, where any time I updated the code repository, 'GitLab' would keep track of changes made, storing older versions of my code, so older versions could be viewed or reverted to. Additionally, 'Git' - the source control framework upon which 'GitLab' is built - enabled me to create 'branches', serving as diverging paths of development. This allowed me to create a branch for a new feature, and then merge this back to the main branch, ensuring there were no code conflicts across the computers I was working on.

5 Evaluation

This section will discuss how the resulting game engine compares to what was proposed, what issues I came into during development, and how I tested various aspects of my game engine.

5.1 Comparison with proposed project

Looking back at the proposed project, I had the goal of creating a 3D explorable environment with some aspect of non-Euclidean geometry. As mentioned previously, due to the ambitous scope of my goals, I was not able to realise the non-Euclidean aspect of my project, however, I did acheive a 3D explorable environment.

5.2 Efficiency

As discussed earlier, throughout the development of my game engine I made sure to consider efficiency. This was in order to make my game engine both more accessible to computers with lightweight hardware, and at the same time improve the scalability. In doing this, I had two types of efficiency in mind. The next two sections will discuss what these are, what caused issues with them, and how I attempted to solve them.

5.2.1 Memory efficiency This was the easier of the two types of efficiency to improve in my game engine. It involves using as little RAM as possible - both standard RAM and VRAM, which will both be referred to as RAM in this section.

Games can often have thousands of objects in the scene, along with shaders, textures and many other necessary components which all require a huge amount of RAM.

The way I improved memory efficiency in my game engine was mostly by ensuring large objects like shaders, *Object*s or textures aren't copied. This would be an issue as lots of classes needed access to the data in these large classes, for example, *Object*s can be rendered with different shaders. This means that each *Object* needs to have access to a shader, and when there are thousands of *Object*s in the scene, this could cause a lot of memory to be used up. Due to the fact that each shader can be used for multiple *Object*s, I decided the best way to increase memory efficiency here was to use pointers. This meant that instead of creating a shader for every *Object*, shaders can be

initialised first, then *Object*s can just point to their required shader, so copies of shaders do not need to be created.

5.2.2 Processor efficiency Improving processor efficiency was much more complex. Processor efficiency refers to how effectively the CPU and GPU (both referred to as "processor" in this section) utilise their resources to handle various tasks. Generally, increasing processor efficiency involves reducing the time complexity of algorithms, which can be achieved with a deep understanding of the algorithms used.

An example of a processor heavy task in my game engine is the collision detection component. This is because every object must be checked for collisions, and this check involves a lot of calculations (see section 4.6.1 on SAT).

The main way that I ensured processor efficiency was by minimising redundancies as much as possible. A single redundant line of code may have only a small effect on the processor but, if there are many frames each second, each frame including many methods iterating over *Objects*, this leads to a great toll on the processor. Removing as many redundencies as possible can produce a substantial improvement to processor efficiency.

There are many ways of improving processor efficiency other than just writing less redundant code, but these are very complex in the scope of game engines, and I was not able to implement them given the deadline. However, I ensured that further development to my game engine was as easy as possible. One such method of improving efficiency is known as culling. This technique avoids including out of view objects in rendering calculations, improving efficiency of rendering. Another method, improving efficiency of collision detections, is spacial partitioning. This involves dividing world space into smaller, more managable regions. This means that instead of checking every pair of objects, only the objects in nearby partitions must be checked, reducing the number of collision checks therefore improving processing efficiency.

5.3 Testing

This section will cover what was tested, why it was tested, and how this testing was done.

5.3.1 Performance testing As discussed throughout this document, each iteration of the game engine is known as a 'frame', where the more frames that are displayed each second, the better the engine is performing. This means that frames per second (FPS) can be used as a measure of general performance. To measure this, I added the method $GetAvgFrameRate$ which divides the amount of frames the game engine has completed by the time that the engine has been running for. On my laptop - a Dell XPS 13, with an 11th Gen Intel Core i7-1165G7 processor, 16 GB of RAM, and Intel Iris Xe Graphics - I can acheive an average of 350 FPS with 100 objects being rendered. Given that most screens can only display up to 60FPS, my game engine has a good performance.

Another way I measured performance was by using Visual Studio 2022's diagnostic tools. This allowed me to see how much processor and memory resources my game engine was using during runtime, and how this varied. I used this in tandem with other general tests throughout my project to see how different scenarios affect hardware usage. For example, a method of testing I employed was stress testing. This involved pushing the game engine to its limits, which I did by adding more and more objects to the world until the FPS was so low that the effect of smooth movement was no

longer apparent, where each individual frame was visually distinguishable. In doing this, I gathered data in order to see how object count affects FPS, processor usage and memory usage. The below figure shows this collected data.

Object count	**FPS**	**CPU usage (%)**	**RAM usage (MB)**
0	1500	14	91
50	540	19	96
100	350	20	98
200	235	21	107
300	170	22	112
500	100	21	127
750	60	21	144
1000	45	21	162
1500	30	22	198
2000	24	21	232
3000	16	21	303
4000	12	21	374
5000	9	21	444

Figure 6: Performance data table

The next figure shows these points plotted on a graph, where the x axis represents amount of objects in the world, and the y axis represents FPS and RAM usage for the green and black lines respectively.

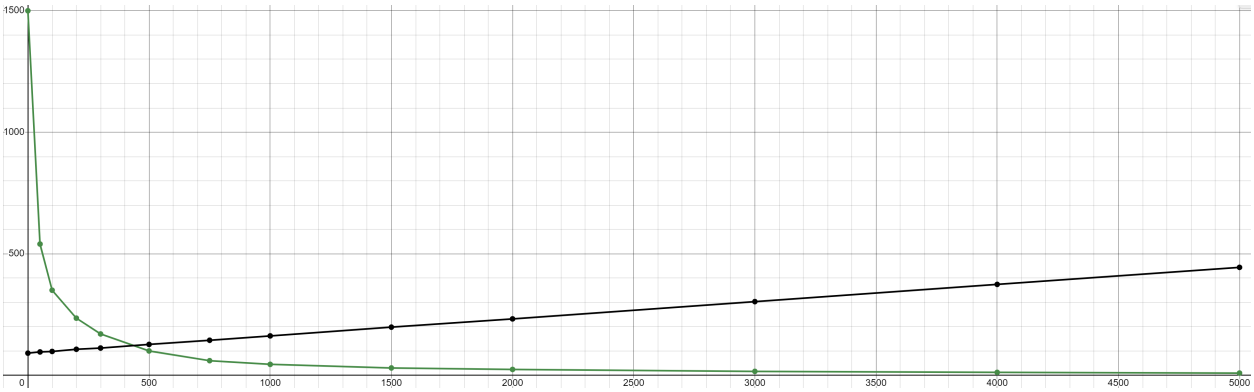

Figure 7: Performance data graph

In this graph, I have omitted the CPU usage data points becuse they show little correlation with object count. I believe that this is due to the efficiency of processor heavy tasks in the game engine, in addition to the fact that the game engine relies more on GPU and RAM resources.

The graph shows RAM usage as increasing linearly with respect to object count, which makes sense as all the game engine was adding during testing was objects, and objects should be represented by the same amount of information. From the data shown, objects take up approximately 0.07 MB of data, which equates to 7×10^4 bytes. Initially, with a small amount of objects, FPS

was shown to have a sharp decline as objects were added. However, as more objects were added, the rate at which FPS decreases begins to slow. In other words, each additional object contributes to less FPS decline compared to earlier stages. This diminishing rate of decrease suggests the game engine's performance degradation becomes less severe as the workload intensifies.

Due to the varying state of game engines, this data cannot be perfect, however I did adhere to a set of standards while collecting the data to ensure it is as accurate as possible. First, I made sure to note the data at exactly 3 minutes of game engine runtime. This was both to remain consistant, and to remove as much noise from FPS fluctuation by noting the average after this time.

In addition to this, to make sure FPS data was accurate from my *GetAvgFrameRate* method, and to avoid large fluctuations affecting this frame rate average, I used the tool 'FRAPS' [1]. This showed a more local average of frame rate, rather than frame rate across total time. A downside to this was that having this tool running could have added some additional strain to my laptop's hardware. However, this is negligible and, as I used this tool throughout all testing, it would not have affected the result of the data

I also created a file called 'Demo.h', which had the purpose of summoning objects (with random position, scale, and rotation) to the world, in order to aid in stress testing, and also act as a demonstation of the game engine's capabilities. This random object summoning is not hugely representative of a game engines proper usage, but in order to stay consistant and avoid having any bias towards certain game development styles, a random spread of objects was most appropriate.

As I was collecting data on my laptop, there were additional processes running in the backround unrelated to my game engine. This had the potential to add noise to the data, but only if these processes used enough resources that my game engine could not run at its maximum potential. No hardware on my laptop was being fully utilised, so these additional processes should only have had minimal effect on the quality of my data.

The below figures show what this testing looked like in my game engine for different numbers of objects.

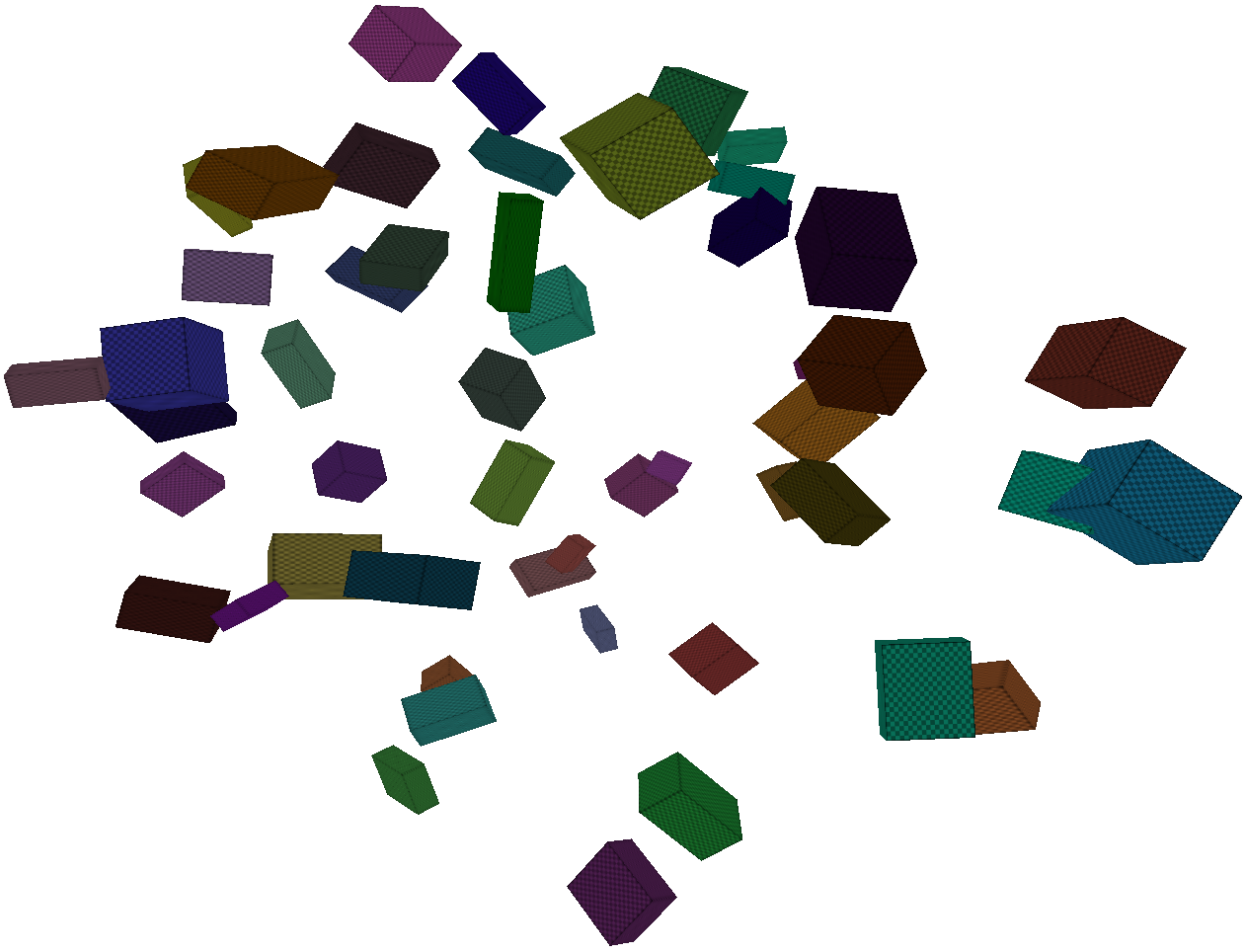

Figure 8: 50 objects

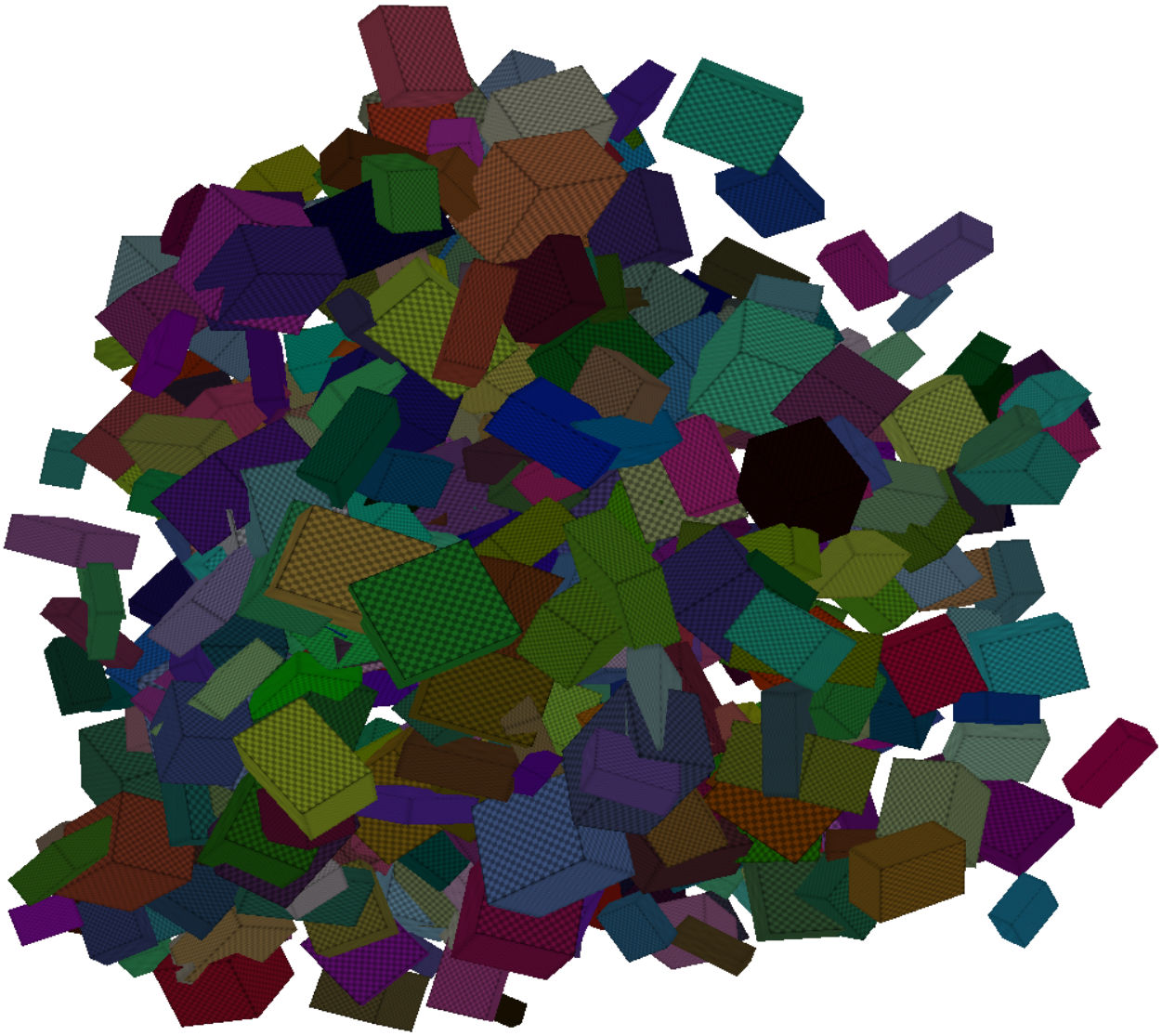

Figure 9: 500 objects

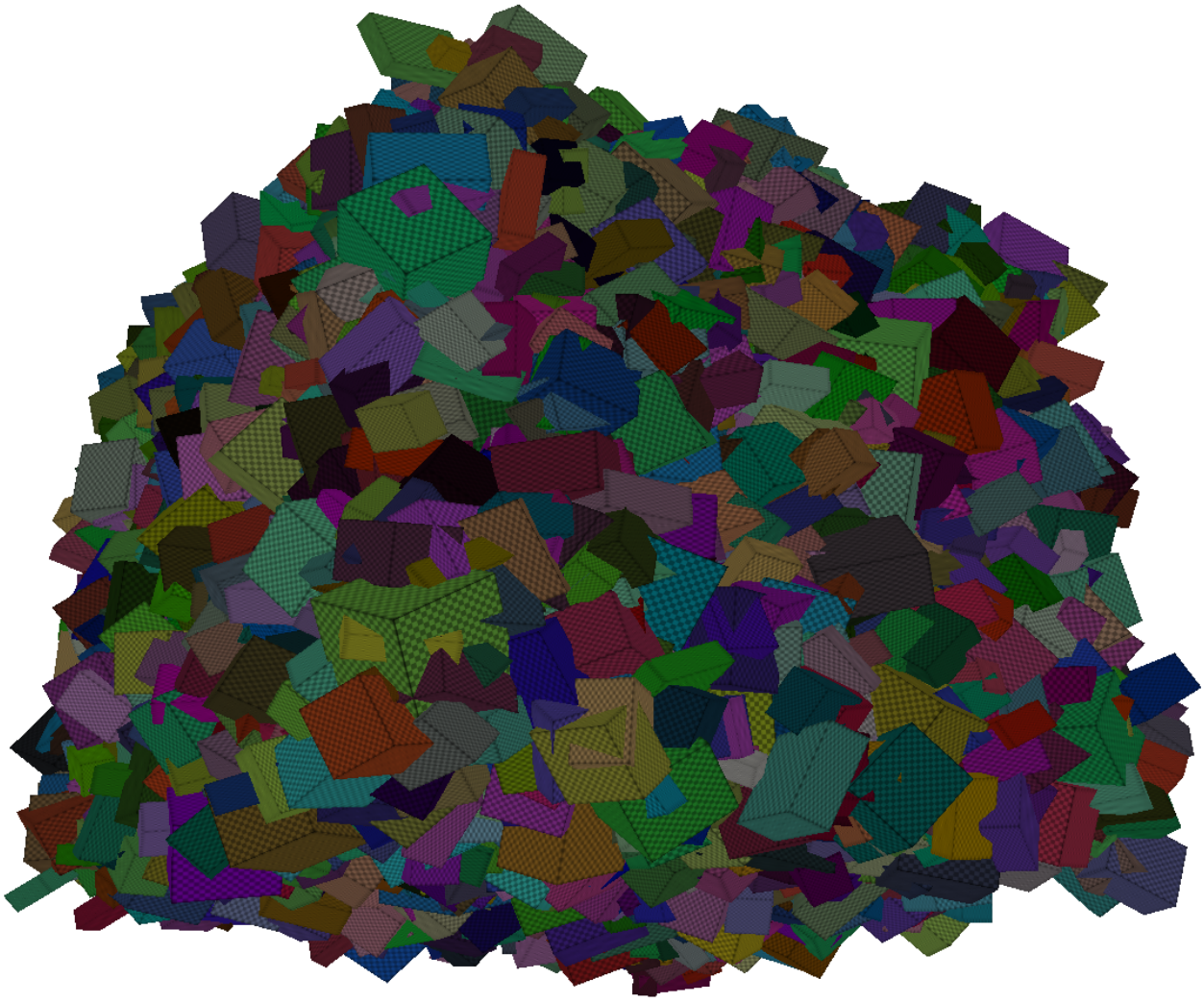

Figure 10: 5000 objects

5.3.2 Feature testing To test functionality of individual features, I first ensured the associated methods were well commented and clearly presented such that I could easily add a line of code to output their current states. An example of this was when I was having trouble with gravity, where objects falling didn't look anything like they would in real life. To test this, I added multiple lines of code to output the velocity of objects as they fell. This led me to discovering that rather than adding the gravitational strength to the downwards velocity of the object every frame, I was instead just setting it, which meant that objects fell at constant rates, instead of accelerating downwards. Another method used to test features was setting breakpoints in my code, which would stop execution at a given point, allowing me to view the state of the variables involved.

6 Potential further development

This section will outline possible changes and additions that could be made to the game engine, and how they can be acheived. It will start by discussing how the rendering of non-Euclidean geometry could be integrated, before describing how the physics and rendering systems could be improved.

6.1 Implementation of non-Euclidean geometry

There are a few methods of implementing non-Euclidean geometry that I considered. The first and most complex being what I refer to as 'full' non-Euclidean rendering. This is where the whole world is in a non-Euclidean space, and the whole rendering method is changed from rendering euclidean geometry to rendering some curved geometry.

6.1.1 'Full' non-Euclidean rendering As mentioned previously, 'full' non-Euclidean geometry is what I use to refer to a method where the whole world is in a non-Euclidean space, which requires the whole render pipeline to be modified. This would have been a timely process to implement so was not considered in the development of this project. However, the next section details another method which was explored in more detail.

6.1.2 Portals The other method of implementing non-Euclidean geometry involved the addition of portals. This is a smaller scale modification, not necessarily modifying the whole render pipeline, rather adding additional render methods specifically for portals. This is still an example of non-Euclidean geometry, as it allows two areas of space to be joined by a 'portal' - a plane in space whereby if an object travels through it, it will be transported to the portal's partner.

This could be implemented by adding a class for Portals, where a portal is a plane in space - therefore it must be described either by 4 vertices, or by scale and rotation. Portals must also have some link between a partner portal. Then, to be able to see through portals, first, the cameras translation relative to the first portal is used to set the translation of a second camera. Then, the second cameras view would be drawn on the first portal. This would give the effect of the main camera seeing through the portals into the other area of space. Additionally, objects moving through portals would need to be implemented.

Issues could arise when drawing a secondary cameras view to the face of a portal if an object is between the secondary camera and portal. This other object would obstruct the view of what should be rendered from past the secondary portal. Also, the secondary portal itself should not be

rendered. To render objects exclusively past the secondary portal, a clipping space can be used, which defines a specific region in space to be rendered.

The figure below shows a diagram of the possible implementation of portals.

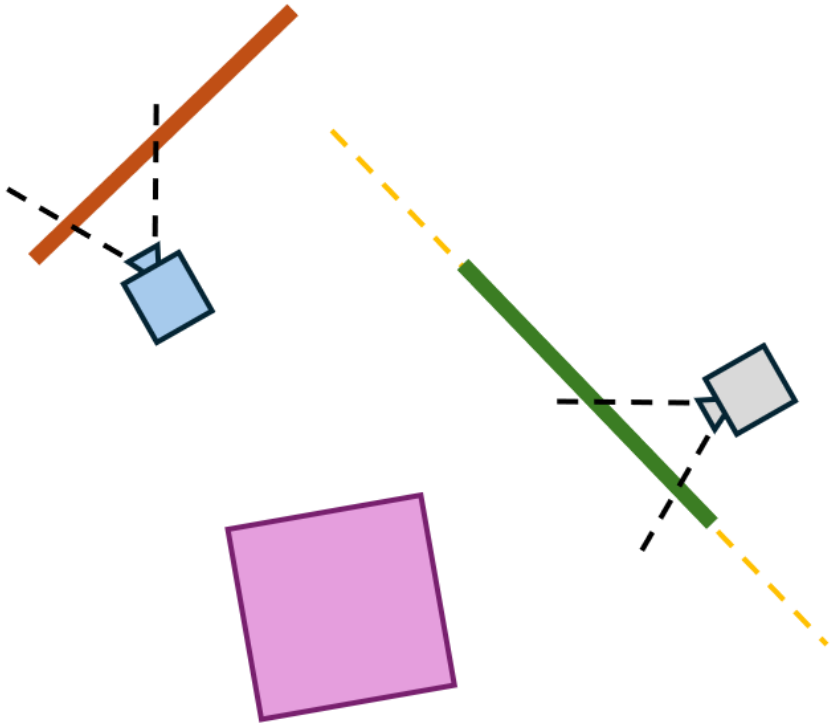

Figure 11: Portal diagram

In this figure, the light blue camera is the main rendering camera, and the light grey camera is the secondary camera used just to render the portal view, where the dotted lines show the cameras field of view. The thick straight red line is a portal, linked to its partner portal - the thick straight green line. The yellow dotted line represents the clipping plane; only objects past this, relative to the secondary camera, are to be viewed through the red portal. Finally, the pink square is the object to be rendered.

6.2 Better collisions

As explored earlier, when objects collide in my game engine, they just stop. This is however not what happens to objects in real life. A way I could improve the realism of my physical interactions would be a complete overhaul of the collision system, removing it from the world class to then create a whole class for physical interactions.

First I would add the functionality to detect where on object the collision occurs, to then add a reactionary force, where objects would 'bounce' off of eachother, carrying their momentum. This would also require some notion of mass, where potentially objects could have some fields related to mass and or density, which would be used in collisions.

6.3 Better rendering

A first point on better rendering would be to use a more efficient rendering algorithm. One example would be shear-warp rendering. Where a 3D object is converted into a stack of slices, this stack is then broken down into smaller sections, then each of these sections are rendered individually, applying a technique known as shear transformation to allign the sections to the viewing plane. This method is very efficient, however, it is highly complex and would be quite timely to implement into my game engine.

A more realistic way of improving my rendering system would first be to allow for objects of different shapes. As my engine currently stands, objects are only able to be represented by cuboids, which limits the types of shapes that can be drawn. Implementing objects of different shapes would be possible by representing objects as lists of vertices, which would then be passed to the vertex shader to be rendered, instead of using the objects model matrix to transform a basic cube.

6.4 Graphical user interface

Another point of improvement which I had considered implementing was a graphical user interface (GUI). This provides the user with other ways of interacting with the game engine, allowing the developer to add things like menus for game options, or, if non-Euclidean geometry is successfully implemented, perhaps some menu to select worlds of different geometries to display.

7 Conclusion

Starting the project, I set out to build a system that could render non-Euclidean space. This was to build on a curiosity of graphics rendering and doing so in a non-conventional way, displaying worlds with mind-bending spatial properties. This lead me to starting work on my own game engine, built from scratch, which was a great learning experience, being able to develop on my established experience on OpenGL to create an efficient, scalable and modular system, which gives room for many future improvements. After building the foundations of the game engine, making the choice to develop these foundations into a polished system instead of trying to rush to meet my requirement of non-Euclidean rendering was the most difficult but important choice. After finishing up with the additional aspects of my game engine, I set out to test its capabilities. In doing so, I found it was able to handle huge amounts of objects with little effect on performance. Overall, I believe that I have met my revised goal to build an efficient game engine.

References

[1] Fraps show fps, record video game movies, screen capture software. 02 2013.

[2] Faizi Noor Ahmad. An overview study of game engines. 09 2013.

[3] Alan Amory, Kevin Naicker, Jacky Vincent, and Claudia Adams. The use of computer games as an educational tool: identification of appropriate game types and game elements. *British Journal of Educational Technology*, 30:311–321, 10 1999.

[4] Andrzej Barczak and Hubert Woźniak. Comparative study on game engines. *Studia Informatica*, pages 5–24, 12 2020.

[5] William Bittle. Sat (separating axis theorem). 01 2010.

[6] Miguel Chover, Carlos Marín, Cristina Rebollo, and Inmaculada Remolar. A game engine designed to simplify 2d video game development. *Multimedia Tools and Applications*, 01 2020.

[7] H. S. M. Coxeter. *Non-Euclidean Geometry.* Cambridge University Press, 09 1998.

[8] Jeff Craighead, Jennifer Burke, and Robin Murphy. Using the unity game engine to develop sarge: A case study. 09 2008.

[9] Joey de Vries. Learn opengl, extensive tutorial resource for learning modern opengl. 2019.

[10] David S Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing and modeling A procedural approach.* Amsterdam [Etc.] Kaufmann Cop, 2003.

[11] Richard Fitzpatrick. Euclid's elements of geometry. 2007.

[12] Jonas Freiknecht, Christian Geiger, Daniel Drochtert, Wolfgang Effelsberg, and Ralf Dörner. Game engines. *Springer eBooks*, pages 127–159, 01 2016.

[13] Kostas Gkikas, Dimitris Nathanael, and Nicolas Marmaras. The evolution of fps games controllers: how use progressively shaped their present design. 2007.

[14] Kyle Halladay. *Practical Shader Development: Vertex and Fragment Shaders for Game Developers.* Apress, 04 2019.

[15] Jian Huang, Klaus Mueller, Roger Crawfis, Dirk Bartz, and Michael Meißner. A practical evaluation of popular volume rendering algorithms. *CiteSeer X (The Pennsylvania State University)*, 01 2000.

[16] John M Kessenich, Graham Sellers, Dave Shreiner, and Opengl Architecture Review Board. *OpenGL® programming guide : the official guide to learning OpenGL®, version 4.5 with SPIR-V.* Addison-Wesley, Cop, 2017.

[17] Henry Lowood. Game engines and game history. 2014.

[18] Tamara Munzner, Stuart Levy, and Mark Phillips. History - geomview manual.

[19] Paul Arthur Navratil. Memory-efficient, scalable ray tracing. *repositories.lib.utexas.edu*, 08 2010.

[20] D. Osudin. Real-time physics and graphics engine for non-euclidean geometry using spherical and hyperbolic trigonometry. 2022.

[21] Cristina Pascua, William Henry Datu, Jann Ceasar, and Benedict Benz Ocampo. Illusory: A non-euclidian concept adoption in a 3d puzzle game. *Illusory: A Non-Euclidean Concept Adoption in a 3D Puzzle Game*, 08 2023.

[22] Riku Skyttä. Cross-platform game development. 2018.

[23] Daina Taimina. *Crocheting Adventures with Hyperbolic Planes: Tactile Mathematics, Art and Craft for all to Explore, Second Edition.* CRC Press, 02 2018.

[24] Luiz Velho, Vinicius da Silva, and Tiago Novello. Immersive visualization of the classical non-euclidean spaces using real-time ray tracing in vr. 01 2020.