Creation of an interactive environment through the development of a Videogame Engine

Jorge Tejado López

Final Degree Work
Bachelor's Degree in
Video Game Design and Development
Universitat Jaume I

June 15, 2022

Supervised by: José Vte. Martí Avilés

To my parents, for supporting me and encouraging me to be who I want to be.

Acknowledgments

First of all, I would like to thank my Final Degree Work supervisor, José Vte. Martí Avilés, for helping me and for always being willing and understanding.

I would also like to thank Miguel Chover Selles for his initial help and motivation at the beginning of the project. Although he does not know it, without Yan Chernikov's help this project would have been impossible. I would also like to thank the people who have accompanied me during the development of the project, Eva María Hernández, Carlos Sanchez, Cristina García, Ignacio Ory, Carlos Rámos, Antonio Marcos, Fernando Montoro, Guillermo Jara and Cosimo Leonardo, whose support and motivation has been essential and invaluable.

I also would like to thank Sergio Barrachina Mir and José Vte. Martí Avilés for their inspiring LaTeX template for writing the Final Degree Work report, which I have used as a starting point in writing this report.

Abstract

This work consists of the development of a videogame engine that facilitates the creation of games with 2D and 3D graphics, mainly shooters, RPGs in 2D or isometric perspective, and board or puzzle games with basic mechanics. This project is focused on developing an efficient and modular engine, which can be easily used through a graphical interface, which is possible thanks to the internal design of the engine or API for each part of it. This interface allows to create entities and add components to them, such as code to give them behavior, textures, or physics among others, also allows saving and loading scenes, facilitating the development of several complex projects.

Contents

CHAPTER 1

INTRODUCTION

Contents

This chapter tries to explain what motivated me to choose and realize this project, where the idea started, and how it has developed in its first steps.

1.1 Work Motivation

Game engines are becoming more and more powerful, becoming necessary to allow developers who do not want to immerse themselves in the technical complexity that often comes with programming to make their ideas possible. These engines mask their components and give us basic functionalities, it is neither necessary nor allowed to modify the use of their physics, their rendering engine, or the most basic mathematical operations, which usually results in a tedious development, with errors or complications in carrying out an idea due to all the possibilities offered by the engine.

This is what motivated me to make my videogame engine, to be able to develop my ideas through a tool that I have designed myself, that offers what I have decided to offer and, above all, how much you can learn about the operation of each of the

parts of which a videogame engine is composed when embarking on a project of these characteristics.

1.2 Objectives

- The engine must be stable, without errors.
- The engine must be efficient. Redundant loading of elements must be avoided.
- The engine must be simple and easy to extend, being able to add new functionalities from the existing ones.
- The engine must be multiplatform, so all the libraries it will use must also be multiplatform.
- The engine must have a design that is easy to understand and that allows its correct functioning, and it must also offer an abstraction layer or API so that developing videogames in it is a quick and easy task.
- The engine must support the main components that form the minimum unit that makes up a video game: 3D and 2D graphics, input device control, entity system, scripting system, physics and sounds.
- The engine has no end goal. As video games are developed, the design and the tools offered by the engine will be iterated, providing it with more and more and better functionalities, making it possible to develop more and more types of games.
- The main objective that has been kept in mind is to learn the inner workings of a videogame and to improve in the development of these.

1.3 Environment and Initial State

The idea for this work came from learning more about engines such as Unity or Unreal Engine, and thanks to some subjects of the degree such as Engines or Computer Graphics, which fuelled the author's interest in learning more about the functioning of the components that make up a video game to develop his own.

It is possible to create a simple engine that satisfies one's own needs or even the specific needs of other people, unlike the most popular engines, which are designed to meet almost any demand.

When defining the objectives of the engine, the engine demanded requirements on its design and operation. The author's lack of knowledge in many areas involved

and the difficulties of finding learning sources that teach the most basic operation of an engine and how to put it into practice, as well as the initial design possibilities and what these imply, have made the use of libraries, explanatory videos, and books on the subject fundamental. In addition, the lack of documentation in this aspect has made it necessary to search intensively for support material that could be useful for the development of this work.

The development environment in which the project was developed was Visual Studio 2019 and the programming language C++. The tools that Visual Studio offers to detect errors are very useful when developing a complex work with many parts involved in which sometimes something goes wrong and detecting where the error comes from quickly and efficiently saves a lot of time and problems. As it was a requirement for the engine to be efficient, the programming language chosen was C++, which is essential because it is powerful and allows working directly with memory thanks to the use of pointers.

As the engine is multiplatform, the library used for the development of 3D and 2D graphics was *OpenGL 4.6.* Widely supported and very powerful, it has recently received its latest 4.6 updates, and it also has great support from the community, which is fundamental for resolving doubts or learning about the development of a rendering engine.

Initially, the *SDL* (Simple DirectMedia Layer) library was used to make it possible to deal with the lower level areas of the engine, as well as input management, window creation, and its fallacy for rendering text or loading sounds. However, due to a bug together with the *ImGui* library to create the graphical interface, *SDL* was replaced by *GLFW*, which although not as easy as *SDL*, still allows the creation of a window and the management of inputs.

Chapter 2

Planning and resources evaluation

Contents

This chapter will discuss the planning that has been followed for the successful realization of the project and the necessary resources that have been used to complete it successfully.

2.1 Planning

The project will start on 21 February and end on 11 May. The total work of the project can be divided into phases corresponding to the implementation of each of the parts of the engine, with a previous phase of documentation and learning of the library to be added or programmed, sometimes shorter than would be ideal due to the limited time available. All the libraries that it has been decided to use are open source, multiplatform, and have an MIT license, which allows us to copy, modify or even distribute any application that has been created with them.

In the beginning there is a general phase of research and documentation on the correct way to configure solutions in Visual Studio and on the architecture

and design of videogame engines. A *Gantt* chart is also attached that explains graphically and in detail the planning that will be detailed below(see Figure 2.1)

- **Initial documentation (10h):** useful information of other reference engines and libraries that would be interesting to use in each component of the engine is collected, as well as the previous learning of each library that is going to be used.
- **Initial configuration of Visual Studio (5h):** to easily configure the programming environment and be able to add libraries, *premake5* is used, written in the *Lua* programming language, which will be in charge of generating all the necessary files and configuring the solution.
- **Window and Inputs (10h):** The *GLFW* library is the one that provides the window on which the game applications and their context will be executed. For the inputs, has made our implementation where the detection of the input is provided by the *GLFW* library itself.
- **Maths library (10h):** *GLM* has been chosen, a library provided by *OpenGL*, but which does not depend in any way on OpenGL. It is lightweight, easy to use, and, most importantly, uses the SIMD language, which guarantees a high speed of operations.
- **Render engine (110h):** *OpenGL 4.6* and *GLEW* to provide cross-platform *OpenGL* support. This would be used in conjunction with mathematical operations to create a visualization, transformations, shaders, and many other graphical possibilities.
- **Import of textures (5h):** the chosen library is *stb image*, for being light and easy to use, it will allow us to import a great amount of widely used formats such as .png, .jpg...
- **Import of models (20h):** *Assimp* is a library dedicated to supporting a lot of formats like .obj or .fbx that store information about geometry, normal maps, animations, and much other relevant information for the import of assets that have want to use in our games created in software like *Blender*, *3ds Max*, or *Maya*.
- **Engine interface (40h):** *ImGui* is a library that is widely used in the video game industry. It is cross-platform, has great support, and even allows docking and high customization of the interface.
- **Entity Systme (15h):** one of the most complicated parts of the engine due to the importance of entity management, sorting, and selection among many

other things. The entt library manages these entities in an easy-to-use and very optimal way, so much so that it is used in renowned videogames such as Minecraft.

- **Physics engine (20h):** as the aim of the engine is to support 2D and 3D videogames, it has been decided to use two separate libraries. For 2D graphics, it has been used *Box2D*, a well-known, lightweight library. For 3D graphics, it has been decided to use *Bullet*. In addition to the free use of these libraries, their choice has been prioritized over others because of their ease of use, allowing us to familiarise ourselves quickly and efficiently with them.
- **Sound (10h):** *Fmod* is a library that supports 3D audio and allows us to import events that have been created directly from the *Fmod Studio* program.
- **Serialisation (10h):** *Yaml* is a library that allows reading a file containing information about the geometry of the model and other information. The programmer must program a function to import, organize and display this information correctly.
- **Gizmos (10h):** The gizmos that appear in the engine editor are provided by the *ImGui* library itself. A modification called *ImGuizmo* has been used and is freely available on GitHub.
- **Documentation (25h):** elaboration of the Final Degree Project report and other important documents.

As it can be seen, the total duration of the project has been 300 hours, most of them dedicated to the graphics engine, due to its complexity and breadth, being still out of the scope of this work due to lack of time for many basic graphical features of other engines such as shadows or particle effects. Another part worth mentioning is the interface, to which a lot of time was spent to fix bugs and make it work well with the rest of the components that make up the engine. Finally, as mentioned at the beginning, each part includes hours of learning and documentation, which, if they were counted together in a split section, would certainly be half of the total 300 hours spent on the engine.

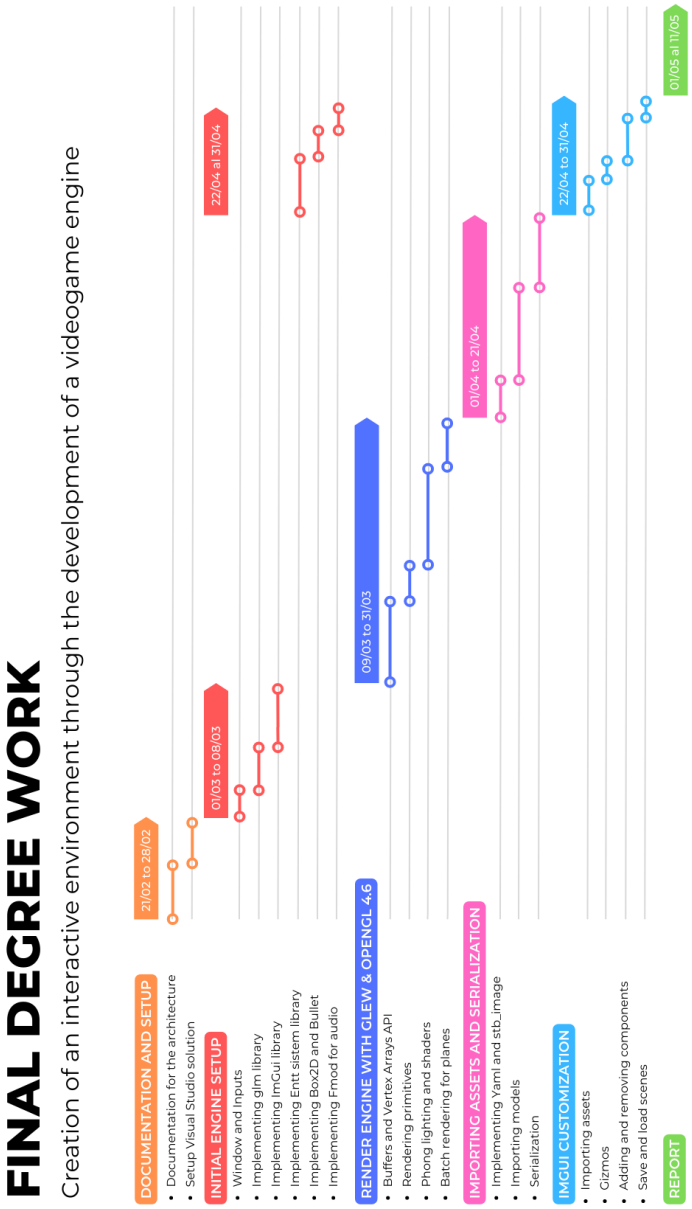

Figure 2.1: Gantt chart of the Final Degree Work (made with Canva)

2.2 Resource Evaluation

The resources used for this project are:

- An HP PC with 16GB RAM, 500GB SSD hard disk, 9th generation i7 processor, and Nvidia GTX 980 graphics card for a price of 750€.
- *Visual Studio 2019* for programming the project.
- *GitHub Desktop* as a tool for managing changes to files, logging changes, and progress, and as a backup system.
- *Blender* to test the import of some assets and different types of textures and maps to the engine.
- chrome://tracing for instrumentation and profiling of the project, which has helped to see runtimes easily and help to debug.
- *NSIS* to generate a .exe installer with the engine generated applications, assets, and .dll files needed.
- *Grammarly*, an application to ensure the correctness of English texts.
- *Overleaf*, a website specialized in editing LaTex documents.

	Partial Cost	Useful information
HP Pavilion 15-bc520ns	750€	
Visual Studio 2019	0€	
Github Desktop	0€	
Blender	0€	
chrome://tracing	0€	
NSIS	0€	
Overleaf	0€	
Deleaker	149€	The free trial version has been used, the partial cost corresponds to the free trial version.
Junior Salary for Game Engine Programmer	30€/h - 40€h 57.000€ - 75.00€/year	Websites glassdoor.es and ziprecruiter.com was consulted to obtain information on the average salary of a junior game engine developer.
Total Cost	9750€/12750€	

Figure 2.2: Table of partial costs and total costs

CHAPTER 3

System Analysis and Design

Contents

This chapter presents the requirements analysis, design and architecture of the proposed work, as well as, where appropriate, its interface design.

3.1 Requirement Analysis

The process of analyzing the entire engine consists of separating it into its parts to be able to observe each of them individually in detail.

The input system will be able to detect inputs coming from a controller, keyboard, or mouse, and the system will even handle events when several simultaneous events are detected.

The rendering engine is by far the largest part of the engine. The engine offers 2D and 3D graphics through *OpenGL 4.6* and *Glew*. A particular shader made for the engine unifies all the basic primitives supported by the engine and the models that can be imported, this ensures that the geometry is affected by the lighting you

have chosen to give it. There are available a directional light, any number of spot lights, and any number of point lights available. It is also possible to change the color of the lights or their ambient, diffuse and specular components. In addition to this, it is possible to color the primitive, add a texture thanks to the texture coordinates provided by the system, or even tint this texture.

Importing these models also allows us to add normal maps, occlusion maps, and many other useful textures for the correct visualization of the model that was used in the software where the model was created. As for the import of textures, the engine allows the import of an atlas of textures where the texture portion it has been interested in can be selected through indexes, as well as different visualization modes for it.

The engine has a native scripting system in C++ language as a component that can be added to the entities. This component implements the behavior it has been wanted to program

The engine is provided with a scene serialization system that allows to save or load scenes in a readable file that can be easily modified by anyone, this also facilitates the work with GitHub when making 'merge' of several branches.

The physics engine is another important component that allows defining the collider of a primitive or a geometry through a mesh collider. It also allows adding a rigid body component to an entity, which makes it possible to act as a static, dynamic, or kinematic body.

3.1.1 Functional Requirements

Thanks to the previous explanation you can easily see what the functional requirements are:

- **R1:** the engine can create a scene
- **R2:** the engine can save a scene
- **R3:** the engine can load a scene
- **R4:** the engine can create an entity
- **R5:** the engine can delete an entity
- **R6:** the engine can give components to an entity
- **R7:** the engine can remove components to an entity
- **R8:** the engine can import models
- **R9:** the engine can assign textures to a sprite renderer

- **R10:** the engine can map a script to a native scripting system
- **R11:** the engine can create a context in the environment that allows physics to work
- **R12:** the engine can resize and customise its interface at will
- **R13:** the engine can switch between edit and runtime mode

3.1.2 Non-functional Requirements

Non-functional requirements impose conditions on the design or implementation. In this project, the non-functional requirements are:

- **R14:** The engine will be simple to use, efficient, modular and easily extensible.
- **R15:** he engine will allow rendering 2D and 3D scenes.
- **R16:** the engine will provide the user with a graphical user interface to mask low-level functionalities.
- **R17:** the engine will be equipped with a scripting system, physics and sounds, basic components to make a videogame.

3.2 System Design

This section must present the (logical or operational) design of the system to be carried out. Below are the different use cases for functional requirements:

Requirement:	R1
Actor:	Engine
Description:	The engine can switch between edit mode, for the creation of applications, and runtime mode, to observe a simulation of the application.
Preconditions:	1. The engine must be in the opposite mode to the one you want to switch to 2. The user must press the stop button or the play button to switch to the opposite mode.
Normal sequence:	The engine will stop the simulation and change the camera to show the editor if it was in runtime mode. On the contrary, the editor camera will change to the one that has been chosen as primary (if there is a camera in the scene) and the engine will start the application
Alternative sequence:	None

Table 3.1: Case of use «Editing and runtime mode»

Requirement:	R2
Actor:	Engine
Description:	Engine can save a new scene
Preconditions:	1. The user must press the save scene button
Normal sequence:	The current scene, if any, is saved to a 'yaml' file, initiating a serialisation process of the current scene elements
Alternative sequence:	None

Table 3.2: Case of use «Save scene»

Requirement:	R3
Actor:	Engine
Description:	Engine can load a scene
Preconditions:	1. The user must press the create scene button
Normal sequence:	Current scene, if any, is deleted to make way for a deserialisation process of the selected scene to be displayed
Alternative sequence:	None

Table 3.3: Case of use «Load scene»

Requirement:	R4
Actor:	Engine
Description:	Engine can create a new entity to which to add components
Preconditions:	1. User must click on the create entity button
Normal sequence:	A new entity is created, which is added to the group of entities in the scene shown in the scene hierarchy
Alternative sequence:	None

Table 3.4: Case of use «Create entity»

Requirement:	R5
Actor:	Engine
Description:	Engine can remove an entity from the scene
Preconditions:	1. The user must press the button to delete an entity
Normal sequence:	The entity to be deleted is deleted from memory and disappears from the scene hierarchy window
Alternative sequence:	None

Table 3.5: Case of use «Remove Entity»

Requirement:	R6
Actor:	Engine
Description:	Engine can add components to an entity in the scene
Preconditions:	1. The user must click on the add component button and select which one to add
Normal sequence:	When adding a new component, it gives the entity new capabilities and functionalities that can be modified
Alternative sequence:	None

Table 3.6: Case of use «Add component to an entity»

Requirement:	Delete component of an entity
Actor:	R7
Description:	Engine
Preconditions:	Engine can remove the entity component from the scene
Normal sequence:	1. The user must press the delete component button of the component he wants to be deleted
Alternative sequence:	When a component is removed, it disappears from the entity inspector and the functionalities it provided to the entity disappear

Table 3.7: Case of use «R7»

Requirement:	R8
Actor:	Engine
Description:	Engine can import models into the scene
Preconditions:	1. The user must click on the import model window which will open a pop-up window to browse for the file
Normal sequence:	Geometry and texture mapping information contained in the file is loaded into the scene
Alternative sequence:	None

Table 3.8: Case of use «Import models»

Requirement:	R9
Actor:	Engine
Description:	Engine can assign a texture to a 'sprite renderer' component
Preconditions:	1. Entity must have a component of type 'sprite renderer'. 2. The user must drag an image of the valid format to that component
Normal sequence:	The texture is displayed on the geometry of the selected entity according to the characteristics determined by its sprite renderer component
Alternative sequence:	None

Table 3.9: Case of use «give texture»

Requirement:	R10
Actor:	Engine
Description:	Engine can assign a script to a native scripting system, which provides a behaviour to the entity
Preconditions:	1. The entity must have a component of type 'native script component'. 2. The user must assign a file of the correct format and without errors in the code provided
Normal sequence:	When running the application, switching to 'runtime' mode will observe the behavior given by the script to the entity
Alternative sequence:	None

Table 3.10: Case of use «Assign script»

Requirement:	R11
Actor:	Engine
Description:	Engine can create a context in the scene that allows the execution of the physics-enabled entities
Preconditions:	1. The user must add a component to the scene of type rigidbody. 2. The user must add a component to the scene of the collider type and of the desired geometry
Normal sequence:	The current scene, when switched to 'runtime' mode, will simulate the physics that the entities have assigned to them
Alternative sequence:	None

Table 3.11: Case of use «Create physical»

Requirement:	R12
Actor:	Engine
Description:	Engine can resize and customize its interface
Preconditions:	1. The user must drag the borders or the top bar of the windows to reposition or resize them
Normal sequence:	The graphical interface of the engine will be modified and adjusted according to the behavior programmed for the inputs that have been detected
Alternative sequence:	None

Table 3.12: Case of use «UI Customization»

Requirement:	R13
Actor:	Engine
Description:	Engine can create a new scene
Preconditions:	1. The user must press the create scene button
Normal sequence:	The current scene, if any, is unloaded and deleted to make way for the new scene that has not yet been saved
Alternative sequence:	None

Table 3.13: Case of use «Create scene»

None

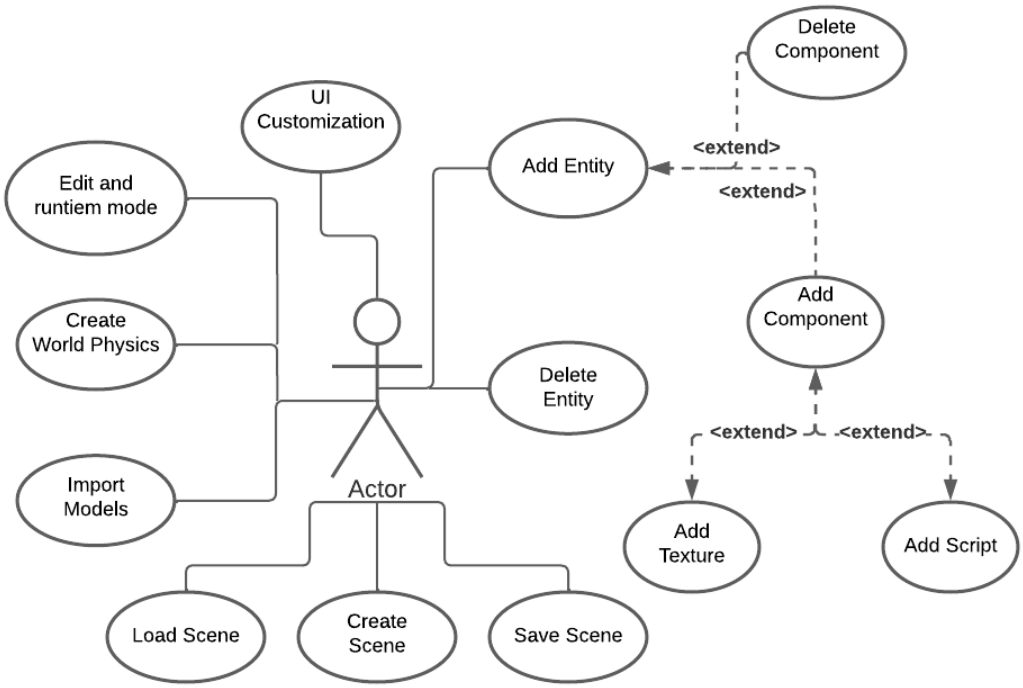

Figure 3.1: Case use diagram (made with *LucidChart*)

As can be seen in the use case diagram, the engine has several functionalities that work independently. Only in the part of the entities is when has need a previously created entity to add a component. In the same way, if it has been wanted to add an asset of a specific type such as textures or a script, the entity that is going to receive it must have previously added the appropriate component that is going to contain the asset.

It is also worthwhile to make a class diagram of the final game engine design. Many classes inherit from other classes and have distinct main parts. The engine design focuses on masking the libraries and their low-level functionalities so that they can be used with the engine's implementation, which is easy to understand and use so that any kind of functionality offered by these libraries can even be adapted in the engine's GUI.

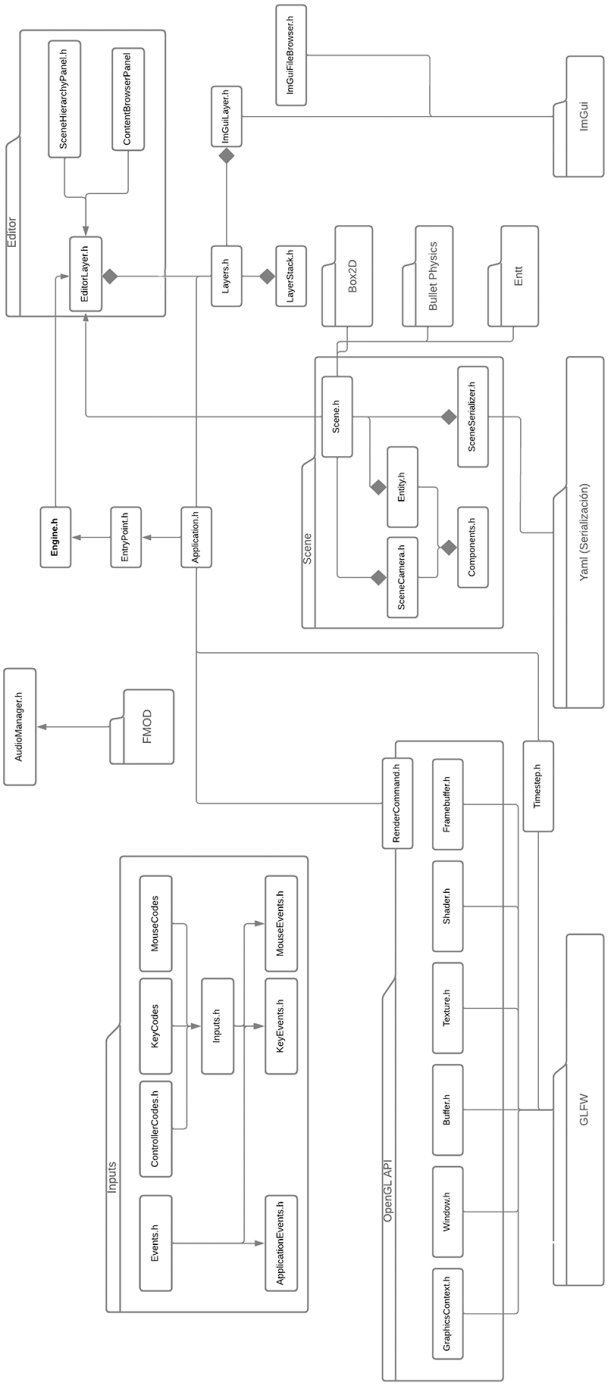

Figure 3.2: Class diagram of the engine (made with *LucidChart*)

3.3 System Architecture

The requirements to play the build of this project in a PC are:

- The operating system Windows 7 at least
- A CPU with x64 architecture
- A keyboard and mouse
- Support for *OpenGL 4.6*

3.4 Interface Design

The engine provides the user with a graphical interface to mask the main functionalities that allow the editing and creation of the scene, the windows that make up the editor and their usefulness are explained below.

- **Editor:**

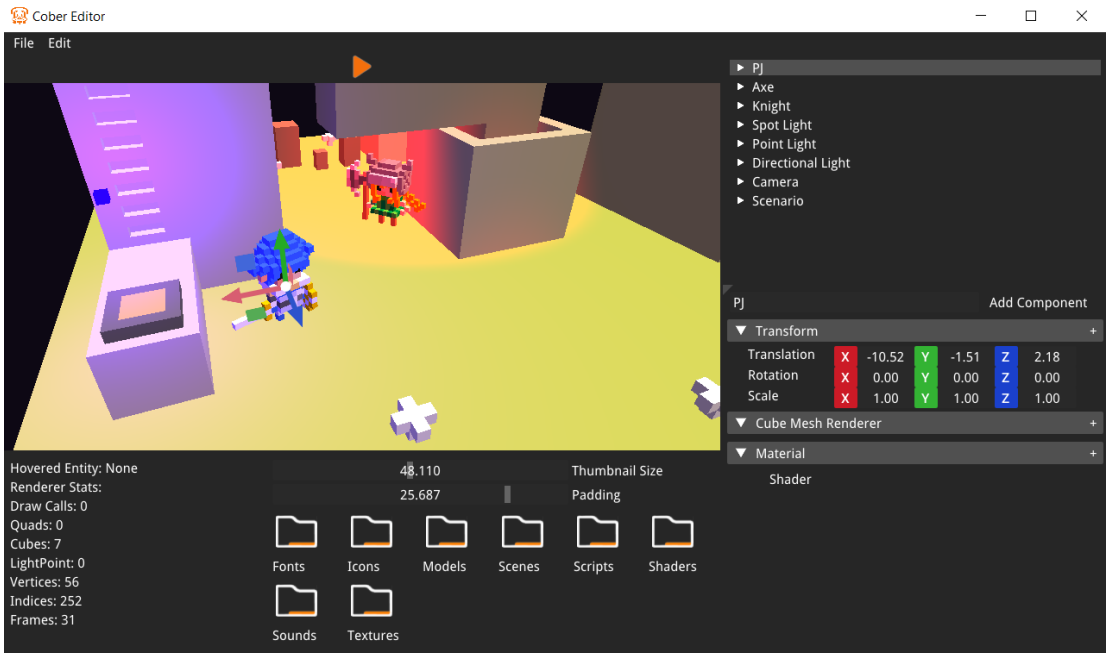

Figure 3.3: General editor interface (from *ImGui*)

- **File Menu:**

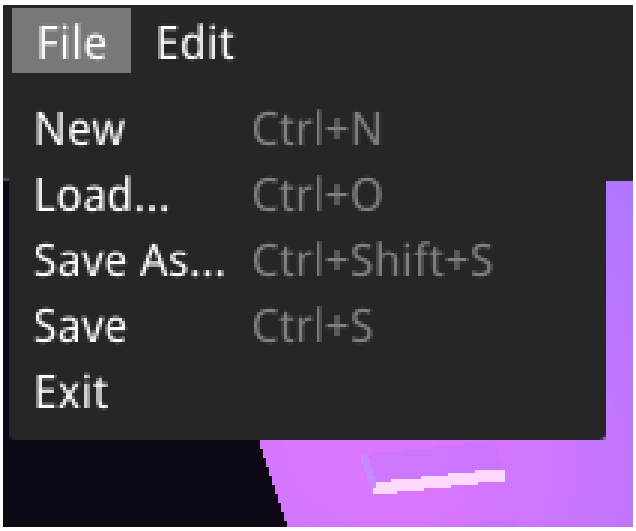

Figure 3.4: File menu window (from *ImGui*)

- **Inspector:**

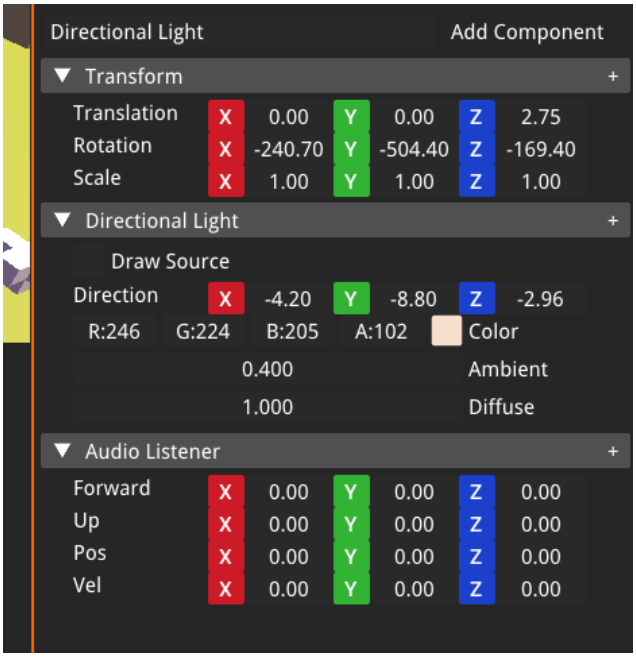

Figure 3.5: Properties of the entities detailed (from *ImGui*)

- **PlayButton:**

Figure 3.6: Play and stop button for change between editor and runtime mode (from *ImGui*)

- **Scene Hierarchy:**

Figure 3.7: Entities on the scene (from *ImGui*)

- **Settings:**

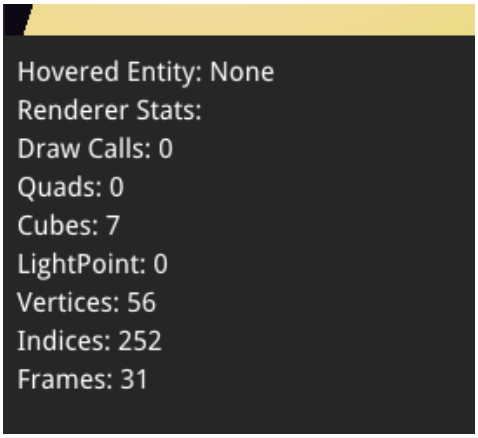

Figure 3.8: Important settings from the scene (from *ImGui*)

- **Runtime:**

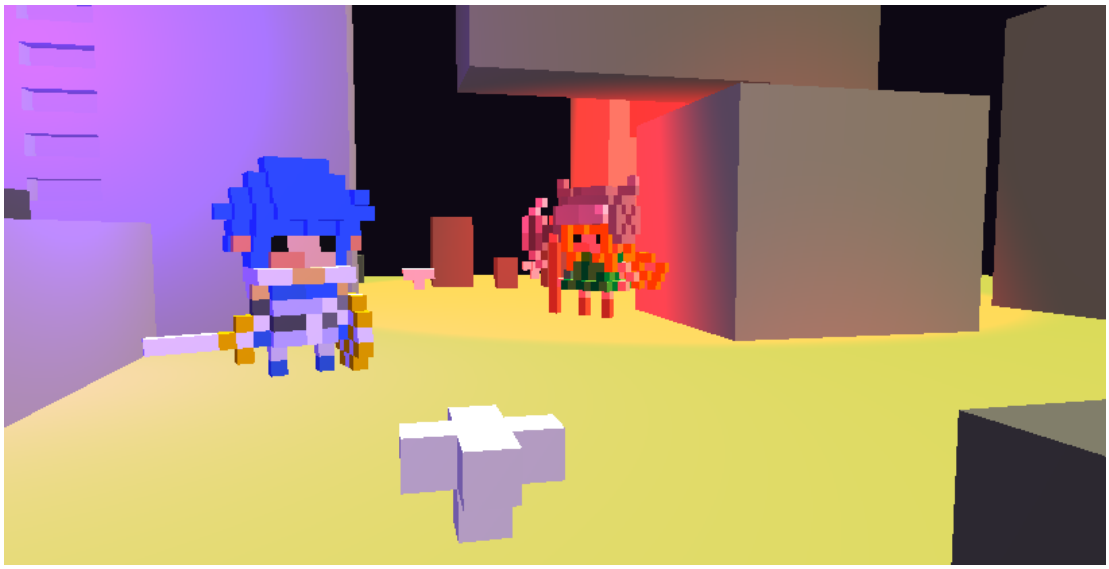

Figure 3.9: Application runtime (from *ImGui*)

Chapter 4

Work Development and Results

Contents

This chapter tries to show how the work has developed from the end of the planning, when the development started, to the completion of the project. It also includes a chapter on the results obtained and changes that have occurred concerning the initial planning due to difficulties in the implementation, changes due to decisions regarding the capacities and objectives of the project, or adjustments due to the time limits of the work and the characteristics of the work.

4.1 Work Development

The developed work will be explained in chronological order, as this is the best way to understand how all the developed parts work and how they intertwine with

each other. The chronological order is identical to the order of the parts that make up the project from the low level up to a higher level of abstraction, and then to the general interface that allows the user to handle everything easily.

4.1.1 Core

The first part of the project consisted of choosing the basic tools and systems that allow a project of these characteristics to be carried out: Visual Studio as the programming environment, *premake5* to easily generate and modify projects and [14]*SDL2* as the library that facilitates the most basic initial tasks such as generating a window or managing inputs and events at a low level. During the course of the project, [14]*SDL2* was changed for [2]*GLFW* due to an error in the compatibility with the engine's graphic interface library that made it impossible to continue working with [14]*SDL* if the objective of making the application work correctly and be customizable by the user was to be met. In this phase of the engine, it has been fundamental the reference of the engine that is made in the youtube channel of [6]*Cherno*, one of the few sources of reference to learn how to build an engine from scratch. The contribution of this channel to my project has been fundamentally the design of the architecture, which has helped to understand the functioning of how the parts that compose it are intertwined and why, as well as different design decisions.

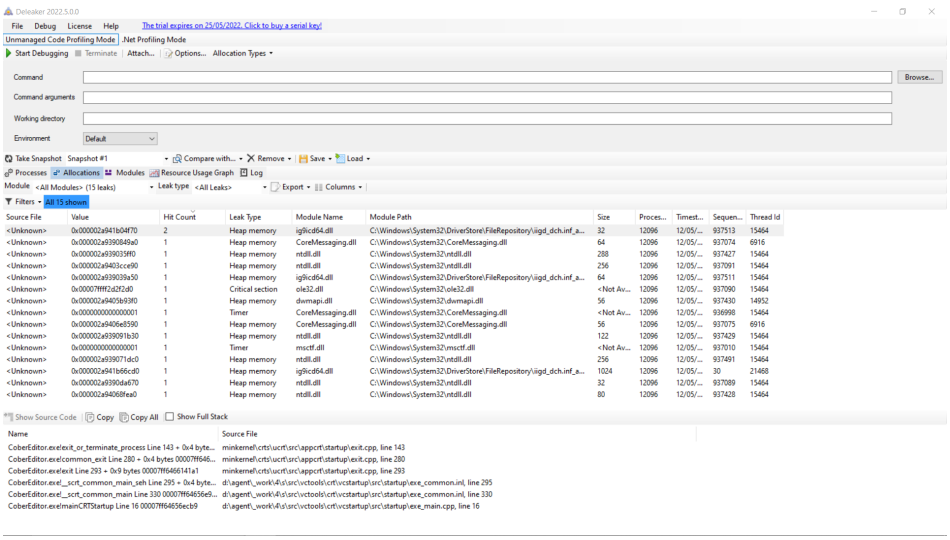

Figure 4.1: Profesional application Deleaker analize memory leaks on the code from the engine

The central components that make up the core of the engine are detailed below.

Application Class

The creation of the window is facilitated by the [2]*GLFW* library which is provided with commands for basic features such as generating the window context, dragging, minimizing and also detecting keyboard or mouse inputs. In this part, the initialisation of the engine and the game loop are performed.

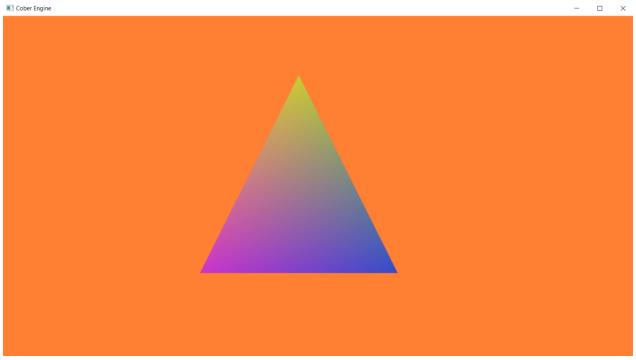

Figure 4.2: Early window of the engine

Timestep

The use of an internal clock is essential for the execution of methods such as *Update* that need a refresh rate for scene rendering or physics management, among others. This timer is again provided by the [2]*GLFW* library, which also serves to count the frames at which the application is running. It is also fundamental to allow the optimisation tools to work properly to help detect the execution time in each small part of the engine and thus be able to easily see if there is something badly optimised, memory leaks or errors.

Universal Unique Identifiers

The creation of identifiers is simple and fundamental for the correct identification of each entity that will later make up the scene. This identifier, apart from distinguishing one entity from all the others, has other utilities such as its use in an internal layer of the generated image that allows us to see the scene which stores in its pixels the identification data of each entity, this allows us to know which entity is being selected with the mouse when has been wanted to modify or handle any of them. As with the rest of the components that will be explained soon, the information stored by the scene must be treated with care and copied correctly so that it can be used when saving or loading a scene, creating a new one or copying

the current scene when switching from edit mode to run mode, operation that will be detailed later on.

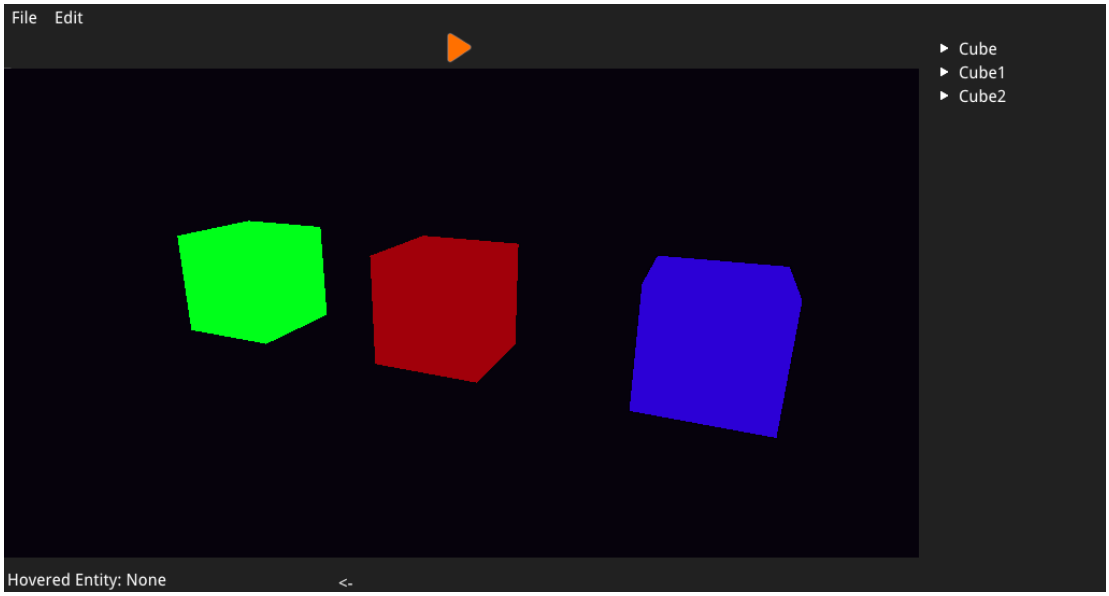

Figure 4.3: Visual representation of the UUID Framebuffer

Layers

Layers are a fairly simple tool that help to group parts of the engine at different times of execution. They consist only of a vector that stores the components of the project, and executes specific methods that all these parts share. So for example, when it has been wanted to execute the *Update* method, all the components that are inside a layer will execute its *Update* method. This design also helps to organise the project, since the editor is just a layer of the engine that can use all the engine's capabilities. Even more simply, the video game that is made in the editor is a layer with the sole task of executing the method that manages the real-time execution of what has been previously decided to create in the engine editor, which is exactly what can be seen when has been hit the play button.

4.1.2 Input

This short section describes the basic operation of the inputs and events available to the game window, keyboard and mouse, the possibility of using a controller is trivial, as each input corresponds to an identifying number. A common feature in videogames is to give the player the possibility to reassign the controls to their liking, that is something that could be programmed into this engine if the developer who is making their videogame wanted to.

Input

This class is in charge of managing all the inputs received with keyboard or mouse. The class is static, accessible from any part of the engine and easy to use as it is designed as an API or programming interface to abstract the most used commands by the [2]*GLFW* library.

Event Manager

The Event Manager manages all events through different categories that work with each other without excluding each other, and an event can belong to two or more categories if desired. To manage an event, it must be subscribed to the list of available events, and then it starts executing with the code has been defined for it. These events work through another API or programming interface, always with the goal in mind to be easily used, modified or extended.

4.1.3 ImGui

The *ImGui* library is a library for making graphical interfaces for applications. It is widely used in the industry and has the support of large companies such as *Ubisoft*, *Blizzard*, and *Supercell*. It is an MIT licensed and open source library as all the libraries that have decided to use, as it was one of the goals of the project. The documentation of [16]ImGui has been very useful, and is implemented entirely in C++ allows the modification of the variables that you want to send through pointers in an agile and efficient way.

ImGui

This library allows changing the style and creation of windows easily, as well as to dock them to each other to modify all the placement of the entire graphical interface to the user's liking, as it is frequently done in other engines and applications. Precisely the problem of the dock was what led to changing the [14]*SDL* library that did not achieve error-free results. For this purpose, a [12]project using [2]*GLFW* and a docker that worked efficiently was used. It could be seen when adapting it to [14]*SDL* that there were errors, so it was decided to remove everything that the engine of this project used from [14]*SDL* and change it to [2]*GLFW*. The creation, management, and customization of the interface windows have been carried out thanks to the features provided by the API or programming interface of the library, the design of the panels that make up the engine, and what is shown in each of them is the contribution to the project, explained in the following section.

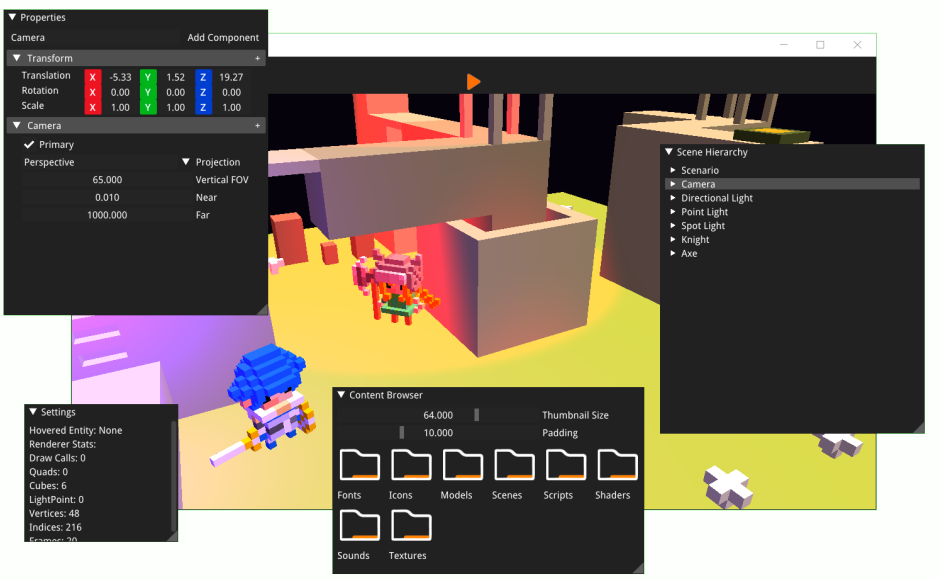

Figure 4.4: Floating ImGui windows

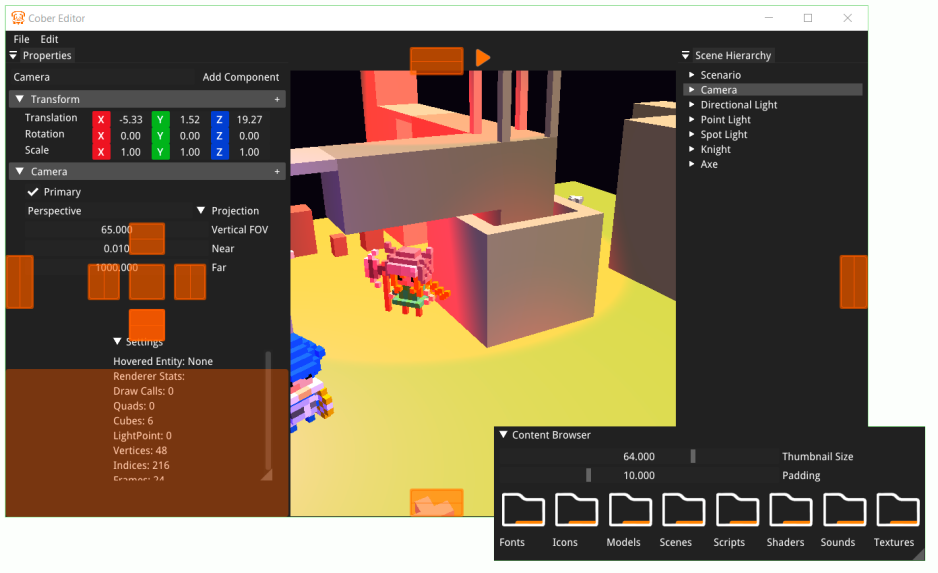

Figure 4.5: ImGui on docking

4.1.4 Panels

The two main panels of the engine, which are built and customized entirely with [16]*ImGui*, are a file browser, which facilitates the development of videogames by not having to leave the engine to assign files or create them, as well as easily visualizing the assets that the engine has, and the scene hierarchy panel, essential for creating and selecting all the types of entities that the engine provides us with. At the same time, when selecting any of these entities, its main components will appear and the possibility of adding any other desired components will be displayed, making it very easy to eliminate, add or modify any characteristic of any component, making this panel the most essential one.

Content Browser Panel

The file search panel has been implemented thanks to the search panel that is integrated into the engine of the [7]*Cherno* channel repository. This search engine has been customized to match the visual identity of the engine and has been completed with a file search engine that uses only the [16]*ImGui* library, so it is independent of any windowing system of the different operating systems. This pop-up file browser is thanks to the implementation in the [12]model loading repository mentioned above.

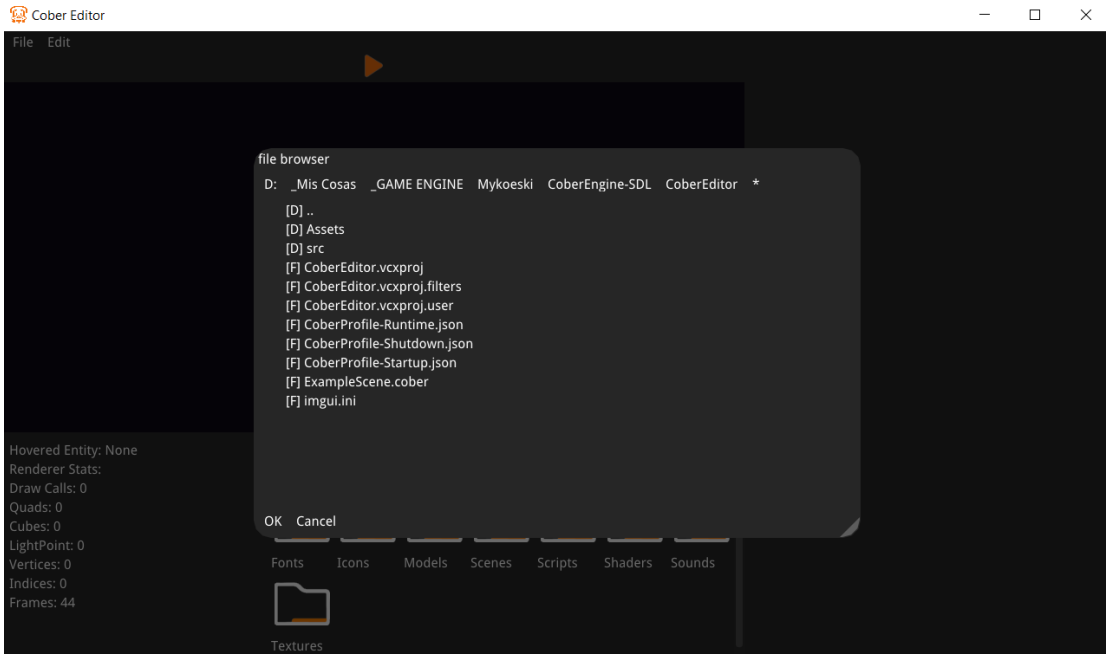

Figure 4.6: Floating file brower window

Scene Hierarchy Panel

The scene hierarchy panel is inspired by the properties panel of other famous engines. It consists of two main windows which are the scene hierarchy panel and the properties panel of the selected entity. The latter panel also allows to easily add, delete or modify components. Information is always correctly displayed when selecting or deselecting elements, when loading or deleting a scene, or when running and stopping the game. At the same time, the modifications in the attributes of the components respond effectively to the proposed design, such as not being able to change anything in the execution mode or the correct updating of values when loading a scene.

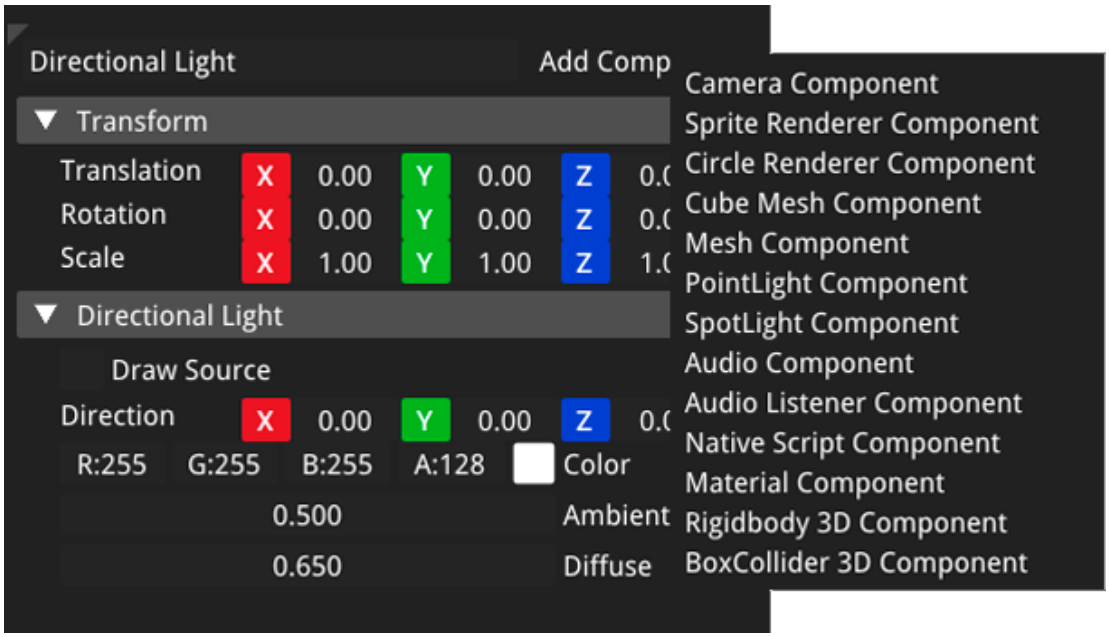

Figure 4.7: Properties panel

4.1.5 Renderer

This section will explain details of the rendering system implemented in the engine. Most of the graphic features belong to the field of [18]*OpenGL* and [11]*Glew*, the library that implements *OpenGL*, and the book [9]Learn OpenGL has been used as a reference and learning source. This section will not go into details concerning *OpenGL*, but will focus on important details concerning the engine and its development. All the elements of the rendering engine, as well as the textures, primitives or shaders are included in an interface oriented to be easily understood as components, as it will be explained below.

Renderer

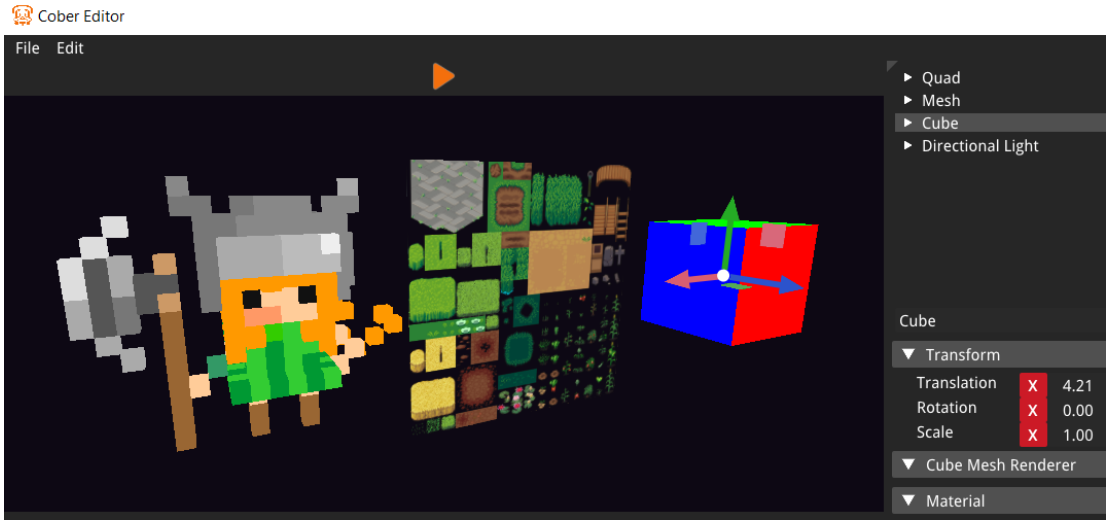

Figure 4.8: Primitive render examples

This is the API or main programming interface that will manage the methods that allow the engine to render primitives and meshes and manage the geometry through commands that encapsulate *OpenGL* commands and the mathematics library [10]glm, this library, light, free and easy to use has been fundamental in all the mathematical part of the engine: the creation of the camera, the use of vectors and matrices regarding geometric transformations, shaders, etc...

The methods allow us to draw sprites, draw models from the path has been assigned in the file browser panel and draw primitives such as cubes or planes. These methods are designed to continue working if they have not been provided with textures or shaders, since each type of primitive has basic default ones.

Shaders

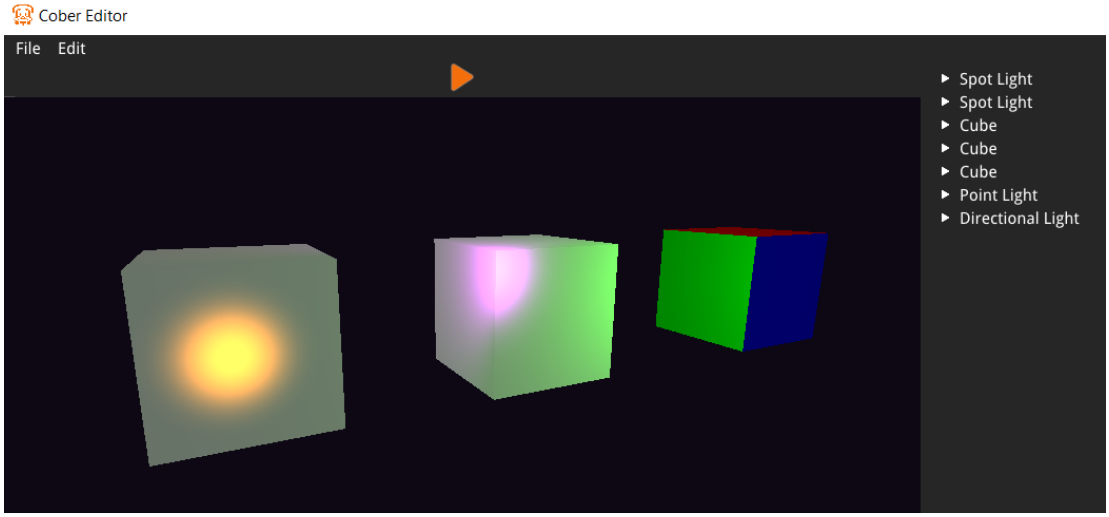

Figure 4.9: Three different shaders with the same lights

This is another class that implements methods to abstract [18]*OpenGL* commands and automates the creation and management of shaders. The API that is designed in the [7]*Cherno* repository makes it very easy to work with shaders and update data with different formats. The contribution that this project adds is the shader scripting focused on working flexibly for 2D rendering or 3D rendering with different features and lights that can be added or removed affecting the scene as the user prefers. A system of Phong fights is used for all the shaders that need lighting, so the lighting will affect one primitive or another in the same way.

Framebuffers

The implementation of the scene that will be detailed later on involves making different renders of the application by generating textures to which it can be given different utilities. The texture, generated in real-time, that represents the graphics of the application that is being executed can be introduced in the context of a [16]*ImGui* window, this is how it can manage the scene from the graphic interface. This texture is exclusive to the editor it can add visual elements that help in the development of an application such as [5]guizmos or icons. Another useful feature of the frame buffer is the generation of a texture that stores in each pixel information about the entity on which the mouse is held so that entities can be selected in the scene. Another important frame buffer is the texture that stores depth information, without which it would be impossible to render graphics with

three-dimensional or two-dimensional layered information. The implementation of framebuffers has been done with an extensible and easy-to-use design, thanks, again, to the design proposed in the [7]*Cherno* repository, the main source for learning game engine design and architecture.

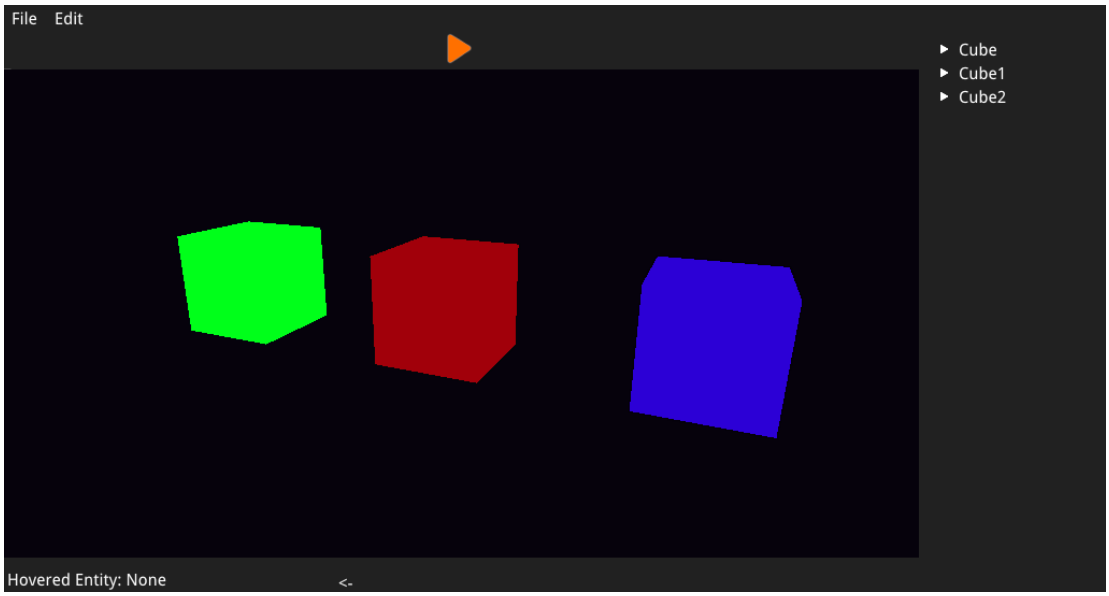

Figure 4.10: Framebuffer examples

Textures

The API of the textures is done thanks to the [1]stbiLoad library, which specialised in supporting different image formats correctly. The methods that make up the programming interface have been carefully designed to be able to modify information about their repeat mode, color, change a texture or have a default texture if there is none assigned. This API is also provided with methods to support texture atlases, a very common way to optimize texture loading in videogames, which consists of loading several elements in a single texture that will be trimmed and loaded into the scene thanks to indices. This is also an easy way to make 2D animations.

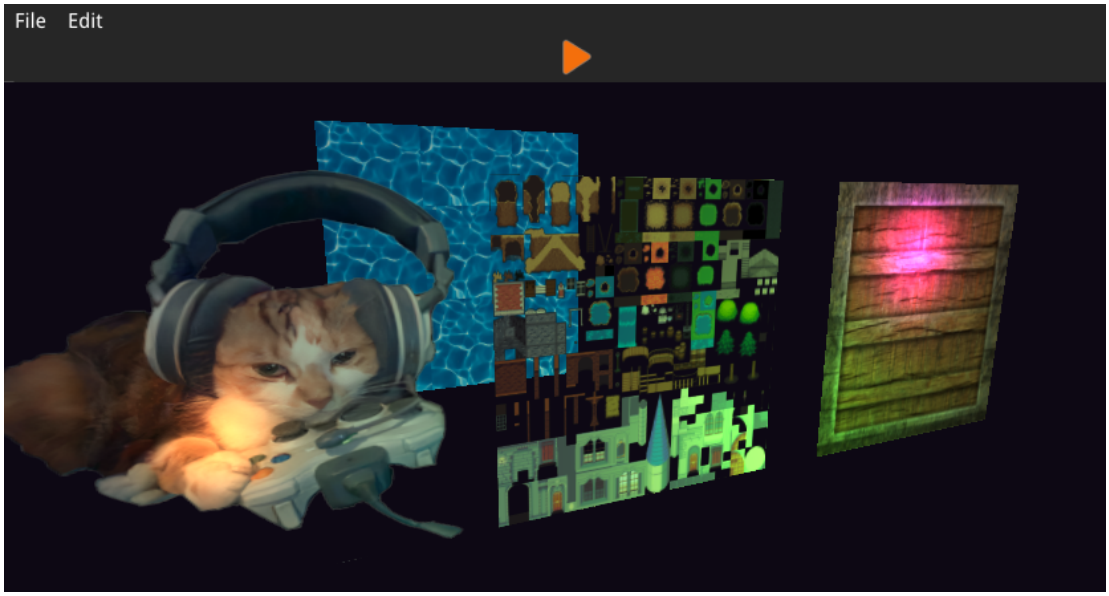

Figure 4.11: Textures performance

Renderer Commands

During the development of the engine, there are many operations that are frequently performed, such as recalculating matrices involved in the camera display, recalculating the scene view in the interface when resizing it, etc... This class becomes necessary to encapsulate those operations in static methods, easily invocable anywhere and abstracted to be easily understood.

Camera

This initial camera class is the minimum expression of the elements that make up a camera in a rendering engine. This class will inherit elements of the camera as the one that can be controlled in the scene or the one that can be programmed in our game. The specific details of each camera, such as controls or frustum features, are reserved for each specific camera class.

Editor Camera

The most important camera type is that it will have the editor of our graphical interface. This camera is provided with the basic functionalities that an editor needs in a three-dimensional graphic environment. With several inputs at the same time it will be able to move fluidly, zoom, vary the fov, or move along a specific axis.

Primitives

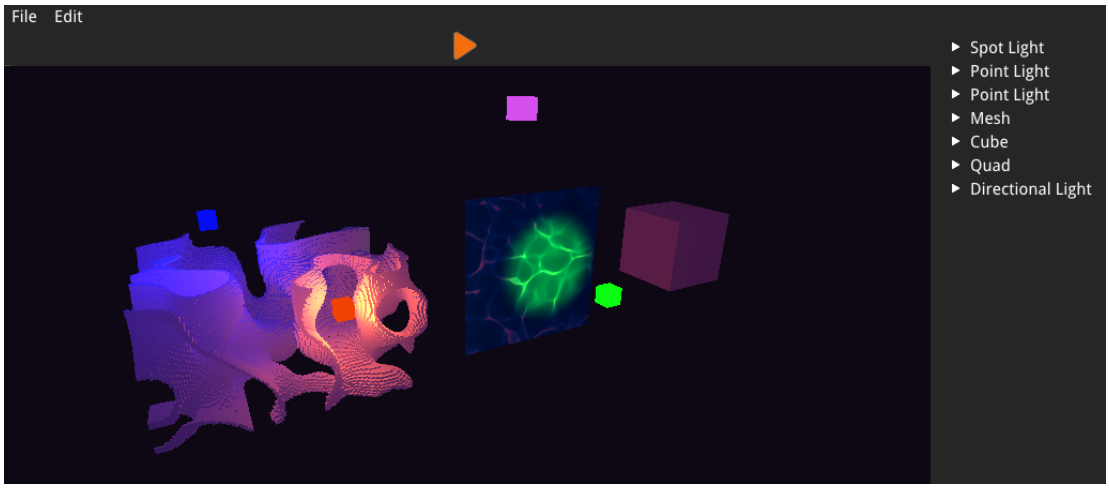

Figure 4.12: Some basic primitives

This class makes it easy to define basic features such as a cube, a light source, a plane or a mesh. The attributes and methods of each primitive have been programmed to work in a flexible way, being able to give it basic features or even custom shaders, although if it is not provided with its own shader, each primitive has certain characteristics. Thanks to the tutorials of [6]Cherno's channel, the rendering of the planes is done through a batch system that groups the geometry, accelerating enormously its processing. This geometry grouping system is worth to be taken into account to render the rest of primitives in the same way.

Mesh loader

In order to load models, it has been essential to use the [13]*Assimp* library, which supports a wide variety of formats and types of data that they contain. Models with a wide variety of textures can be imported into the scene to enrich its visualisation, and it is also possible to read any animation data that may exist, although it has not yet been possible to include a system of animations and a state machine.

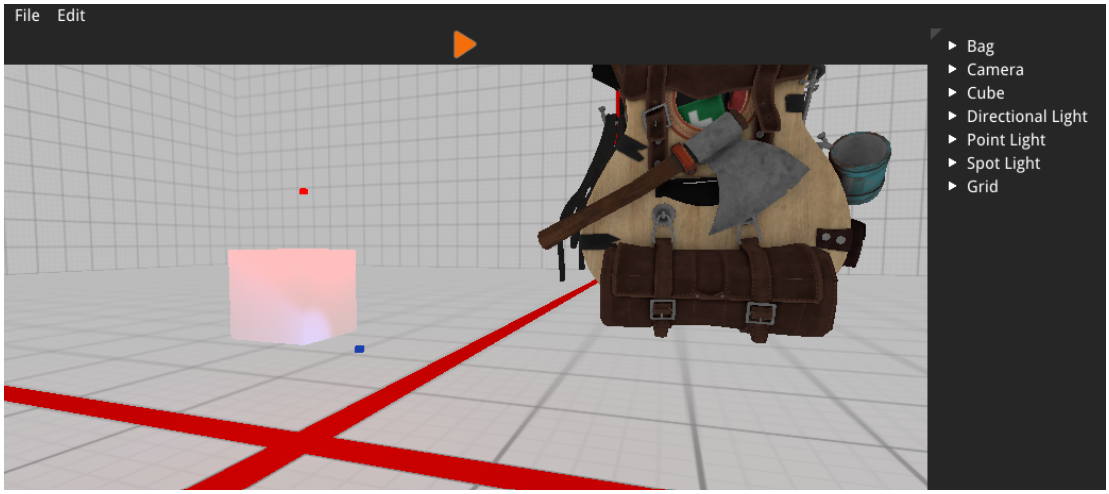

Figure 4.13: Mesh loader example

4.1.6 Physics

The physics of this project had to be able to flexibly handle 2D and 3D collisions. For this purpose, it was decided to use two specific libraries for each of these purposes, easy to use and free of charge.

Physics

For 2D physics management, the [4]*Box2D* library is used, which is light and very easy to use. For 3D physics it will be used [8]*Bullet*, another well-known, powerful and easy-to-use library written in C++. The APIs of these libraries allow them to be easily implemented in the scene, initialised and created for the physical world of the scene, then executed only in the execution mode of the engine. The implementation of these two libraries has been focused so that the different geometrical shapes that have been implemented for collision detection can function as components. These two physics systems are kept separate at all times by the possibility to choose to create a scene suitable for 3D or 2D physics.

4.1.7 Sound

For the sound, the implementation made by [17]*MotorCasaPaco* has been chosen. It is a static interface, so it is easy to use and understand.

Audio

The library used is *FMOD*, which is easy to use and allows the generation of 3D audio through a sound transmitter and a receiver, which greatly enriches the scene and gives it great realism.

4.1.8 Scene

This is the class that will contain and link the running application with all the entities in it and their components.

Components

The concept of a component in this engine is the public class of each of the things it has been wanted an entity to have. Some components could be lights, primitives and types of colliders and rigidbodies depending on whether you are in a 2D or 3D scene. All the entities are going to have two necessary tributes in an engine as they are its Tag or name and its transform component, a 4x4 matrix that will be decomposed to be able to modify the position, rotation and scale of the entity. Other fundamental components that it has been decided to keep separate are the material, which is in charge of applying the shader that has been chosen it to and the texture component, which although they work together internally, it has been decided to keep them separate since any type of geometry can have a texture. Another of the most important components is the Scripts component, essential for adding behaviors to any entity, thanks to the fact that each entity is identified with a unique ID, and through a search method for entities contained in this component, it can be communicated with any other entity in the scene and modify its attributes and components.

Entity

The system of entities is one of the most complex of the engine, making a system that stores the entities in an efficient and flexible way is a complicated task, that is why a specialised library called [3] has been used, which is the one used by *Minecraft*. For this project, an API of the main actions that it has been going to need when working with entities is made, for example, to see if an entity

has a certain component or to group all the entities that have the component or components that will be told it to have.

Scene Camera

As previously mentioned in the camera chapter, the SceneCamera class inherits from the general camera class and provides its characteristics. Unlike the editor camera that is programmed to move around the editor environment, this camera is the one that will be used in the game, so it is easily configurable as it is used through a component that can be added to any entity. Several cameras can be used in the scene, each with its attributes. The in-game behavior of the camera is reserved for the Scripts component, this class being solely for containing the graphical options and mathematical operations that pertain to the game camera.

Scene Serializer

Scene management is done through text files that are written and read with the [15] *YAML* tool, a library that allows us to serialize scene data easily. Usually, this task is carried out by a .json file, however, [15] *YAML* has been prioritized because it is very easy to use and understand. Anyone, reading the text file with scene data, can understand the attributes that compose the scene along with all the information of its components and modify them easily.

```
Scene: Untitled
Entities:
  - Entity: 9838763865116548908
    TagComponent:
      Tag: Axe
    TransformComponent:
      Translation: [-12.3299999, -5.80000019, 6.65790081]
      Rotation: [0, 0.35869208, 0]
      Scale: [3.00000024, 3, 3.00000024]
    MeshComponent:
      Mesh Path: Assets\Models\VoxelCharacters\chr_sword.obj
    MaterialComponent:
      Shader Path: Assets\Shaders\Primitive.glsl
  - Entity: 16409815784569791132
    TagComponent:
      Tag: Knight
    TransformComponent:
      Translation: [-6.67999983, -5.80000019, -4.9063673]
      Rotation: [0, -0.862541676, 0]
      Scale: [3, 3, 3]
    MeshComponent:
      Mesh Path: Assets\Models\VoxelCharacters\chr_knight.obj
    MaterialComponent:
      Shader Path: Assets\Shaders\Primitive.glsl
  - Entity: 17950340218348495429
    TagComponent:
      Tag: Spot Light
    TransformComponent:
```

Figure 4.14: YAML scene file example

Scene

This is the class that defines the scene and one of the most important classes of the engine. The management of the scene is done through a very simple system that is divided into the editor scene and the application scene. When you want to switch to the game mode by pressing the play button, the scene class takes care of reserving the editor scene so that the game scene can be loaded into the current scene. When will be want to go back to the editor, the game scene will be destroyed and the editor scene will be reloaded, protecting from changes that it can be made in the engine editor while it has been test the game. This class is also in charge of making it possible to load and save scenes, add components and create entities with specific characteristics to the scene. All these operations are carried out with care so that the management of entities and their components is efficient, so that the engine does not fail when copying and destroying components.

4.2 Results

The initial goal was to create an engine that would be optimal, easy to understand and use, allowing the development of some basic game types easily. Although some of these types of videogames that were initially proposed as shooters or board games cannot be developed due to some necessary parts that have not been implemented due to lack of time, the engine is fully functional, modular, easily extensible and allows the development of basic games on it. The following is a list of the initial objectives set out, those that have been achieved, those that have not, and the changes that have been decided and managed to implement even though they were not part of the initial plan.

Initial objectives:

- Input and Event Manager ☑
- Scene with 3D and 2D graphics ☑
  - Texture Support ☑
  - Lights ☑
  - Loading 3D models ☑
  - Animations ☐
  - Complex graphics such as shadows and post-processing ☐
- Physics system ☑
- 3D Audio System ☑
- HUD System ☐

Throughout the project, due to the author's learning and evolution in the subject of videogame engines, some changes had to be made to the initial objectives in order to fulfil the purpose of making a videogame in an easy and fast way that was unknown before. These changes dealt with the management of all elements through a graphical interface and the development of several APIs or programming interfaces that easily handle all the implemented parts.

- Use and customisation of a graphical interface ☑
- Entity system ☑
- Creating, loading and saving a scene ☑
- Framebuffers to select entities or view the scene in the interface ☑

- System of materials to modify shaders and properties easily ☑
- Creation and modification of shaders and properties ☑
- Creation and modification of scene entities from interface ☑

A link to the project is also attached at ***GitHub***:

Chapter 5

Conclusions and Future Work

Contents

In this chapter, the conclusions of the work, as well as its future extensions are shown.

5.1 Conclusions

This has been a project that I have been wanting to do for a long time and to which I have dedicated a lot of love and time because from the beginning my intention with the completion of this work was to delve into the guts of a video game and learn how it works, to know its components very well and understand them at a low level.

I think that throughout this final degree project I have managed to achieve a foundation that allows me to move quickly and understand much better the operation of a program as complex as a video game, break the initial hard barrier that involves understanding so many components and systems so complex and different from each other and facilitate the work when further developing this or other work.

Although I have done other videogames during my studies, this was my first contact with managing such a complex project, which has helped me to learn

how to handle the management and the correct way of programming to make the project sustainable. For future projects, I would like to continue exploring design patterns in the programming architecture that would have been very useful for me when I started developing the engine.

5.2 Future work

Although I am satisfied with the work done, I was aware from the beginning that the engine I wanted to achieve was not possible in only 300 hours and three months of work. This final degree work has been the first foundation in what I want to be my tool for the development of my videogames, so there are still many things that I would like to improve, optimize or implement.

- A Lua-based scripting system.
- That both basic primitives and geometry meshes are rendered together (batch rendering) as planes already do.
- 2D and 3D terrain generation component, as well as the procedural level generation with noise algorithms such as *Perlin Noise*, customizable from the engine interface.
- AI component customizable from the engine interface, allowing to quickly assign different types of behavior to the agents or to determine different algorithms to search for the most optimal path among other algorithms with other purposes.
- Support for sprite animations and bone geometry animation.
- Dialogue system to easily set up text conversations between characters in the game.
- HUD component, rescaling of this and possibility to edit it from the engine interface.
- Basic particle system
- Improve the import of assets to the engine, allowing to drag the files directly to the entity and automate the assignment of the asset to the object depending on its format, among other improvements.
- Advanced *OpenGL* features such as post-processing, shadows...
- Optimise the engine so that it doesn't have any memory leak.

- In general improves the general API until the process of developing a videogame in the engine is sufficiently comfortable and satisfactory, although achieving this is an iterative process that is achieved over time, and the development of many applications in it.

Bibliography

[1] S. Barret. stb image. https://github.com/nothings/stb/blob/master/stb_image.h, 2021.

[2] C. Bas, D. Shuralyov, J. Gray, P. Waller, R. Eklind, and S. Gutekanst. Glfw. https://github.com/glfw/glfw, 2022.

[3] M. Caini. entt. https://github.com/skypjack/entt, 2022.

[4] E. Catto. Project title. https://github.com/erincatto/box2d, 2020.

[5] Y. Chernikov. Imguizmo. https://github.com/TheCherno/ImGuizmo, 2020.

[6] Y. Chernikov. The cherno. https://www.youtube.com/c/TheChernoProject, 2022.

[7] Y. Chernikov. Hazel. https://github.com/TheCherno/Hazel, 2022.

[8] E. Coumans and Y. Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. http://pybullet.org, 2022.

[9] J. de Vries. Learn opengl. https://learnopengl.com/, 2014.

[10] glm. glm. https://github.com/g-truc/glm, 2020.

[11] M. Ikits, M. Magallon, A. Lefohn, Joe. Kniss, and C. Wayman. glew. https://github.com/nigels-com/glew, 2021.

[12] Jayanam. Jglmeshloader. https://github.com/jayanam/jgl_demos, 2021.

[13] K. Kulling. assimp. https://github.com/assimp/assimp, 2022.

[14] S.O. Lantinga. Sdl. https://github.com/libsdl-org/SDL, 2022.

[15] libyaml. libyaml. https://github.com/yaml/libyaml, 2020.

[16] Omar. Imgui. https://github.com/ocornut/imgui, 2022.

[17] I. Ory. Motorcasapaco. https://github.com/freesstylers/Untitled-Motor, 2020.

[18] M. Segal and K. Akeley. Opengl. https://www.opengl.org/, 2017.

APPENDIX A

Index of figures

Figures that appear throughout the report and quick link to where they are found.

List of Tables

List of Figures

APPENDIX B

Source Code

B.1 Core

Application Class

```
enum class GameState { PLAY, EDIT, EXIT };

class Application {
public:
    Application(const std::string& name = "");
        virtual ~Application();

        void Run();
        void Close();

        void OnEvent(Event& event)
        void PushLayer(Layer* layer);
        void PushOverlay(Layer* layer);
        void ProcessInputs();
        static Application& Get();
        Window& GetWindow();
        ImGuiLayer* GetImGuiLayer();
private:
    std::unique_ptr<Window> _window;
        GameState _gameState;
        ImGuiLayer* _imGuiLayer;
        LayerStack _layerStack;
        float _LastFrameTime = 0.0f;
        float _timeInSeconds = 0.0f;
```

```
25          int _frames = 0;
26  private:
27          static Application* Instance;
28  }
29  Application* CreateApplication();
```

Init Engine

```
1   Application::Application(const std::string& name) {
2
3           _Instance = this;
4           WindowProps windowProps = WindowProps("Cober_Engine", W_WIDTH, W_HEIGHT);
5           _window = Window::Create(WindowProps(name));
6
7           Renderer::Init();
8           AudioManager::SetupInstance();
9
10          _gameState = GameState::PLAY;
11
12          m_ImGuiLayer = new ImGuiLayer();
13          PushOverlay(m_ImGuiLayer);
14  };
```

Game Loop

```
void Application::Run() {

	while (_gameState != GameState::EXIT) {

		float time = (float)(glfwGetTime());
		Timestep timestep = time - m_LastFrameTime;
		m_LastFrameTime = time;
		timeInSeconds += timestep;
		frames++;

		if (!w_Minimized) {
			for (Layer* layer : m_LayerStack)
				layer->OnUpdate(timestep);

			m_ImGuiLayer->Begin();
				for (Layer* layer : m_LayerStack)
					layer->OnImGuiRender();
			m_ImGuiLayer->End();

			ProcessInputs();

			_window->OnUpdate();
		}
	}
}
```

Timestep

```

class Timestep {
public:
	Timestep(float time = 0.0f)
		: m_Time(time) { }
	operator float() const;
	float GetSeconds() const;
	float GetMilliseconds() const;
	float GetFrames();
	void SetFrames(float frames);
private:
	float _time;
	float _frames;
```

Universal Unique Identifiers

```
// Universal Unique IDentifier
#include <xhash>

class UUID {
public:
        UUID();
        UUID(uint64_t id);
        UUID(const UUID& other);

        operator uint64_t();
        operator const uint64_t() const;
        uint64_t _UUID;
};
```

Layer and Layer Stack

```
class Layer {
public:
        Layer(const std::string& name = "Layer");

        virtual void OnAttach() {}
        virtual void OnDetach() {}
        virtual void OnUpdate(Timestep ts) {}
        virtual void OnImGuiRender() {}
        virtual void OnEvent(Event& event) {}
        const std::string& GetName() const;
protected:
        std::string m_DebugName;
};
```

```
class LayerStack {
public:
        LayerStack();
        void PushLayer(Layer* layer);
        void PushOverlay(Layer* overlay);
        void PopLayer(Layer* layer);
        void PopOverlay(Layer* overlay);

        std::vector<Layer*>::iterator begin();
        std::vector<Layer*>::iterator end();
        std::vector<Layer*>::reverse_iterator rbegin();
        std::vector<Layer*>::reverse_iterator rend();
private:
        std::vector<Layer*> m_Layers;
        unsigned int m_LayerInsertIndex = 0;
```

B.2 Events

Input

```
class Input
{
public:
        static bool IsKeyPressed(KeyCode key);
        static bool IsKeyPressedOne(KeyCode key);
        static bool IsKeyReleased(KeyCode key);

        static bool IsMouseButtonPressed(MouseCode button);
        static std::pair<float, float> GetMousePosition();
        static float GetMouseX();
        static float GetMouseY();
};
```

Events Manager

```
enum class EventType {
        WindowClose, WindowResize, AppTick, KeyPressed, MouseButtonPressed //...
};
enum EventCategory {
        EventCategoryApplication = BIT(0), EventCategoryInput = BIT(1) //...
};

class EventDispatcher {
public:
        EventDispatcher(Event& event)
                : m_Event(event) { }

        template<typename T, typename F>
        bool Dispatch(const F& func)
private:
        Event& m_Event;
};

std::ostream& operator<<(std::ostream& os, const Event& e) {
        return os << e.ToString();
}
```

Application Events

```
class WindowResizeEvent : public Event {
public:
        WindowResizeEvent(unsigned int width, unsigned int height)
                : m_Width(width), m_Height(height) {}

        unsigned int GetWidth() const;
        unsigned int GetHeight() const;

        std::string ToString() const override;
private:
        unsigned int m_Width, m_Height;
};

class WindowCloseEvent : public Event {
public:
        WindowCloseEvent() = default;
    //...
};

class AppTickEvent : public Event {
public:
        AppTickEvent() = default;
    //...
};
//...
```

B.3 ImGui

ImGui

```
class ImGuiLayer : public Layer {
public:
        ImGuiLayer();
        ~ImGuiLayer();

        virtual void OnAttach() override;
        virtual void OnDetach() override;
        virtual void OnEvent(Event& event) override;

        void Begin();
        void End();

        void BlockEvents(bool block) { m_BlockEvents = block; }
        void SetDarkThemeColors();
        static void PlayModeColor(bool playMode);
private:
        bool m_BlockEvents = true;
        float m_Time = 0.0f;
        GLFWwindow* window;
};
```

B.4 Panels

Content Browser Panel

```
class ContentBrowserPanel {
public:

        ContentBrowserPanel();
        void OnImGuiRender();
private:
        std::filesystem::path m_CurrentDirectory;

        Ref<Texture2D> m_DirectoryIcon;
        Ref<Texture2D> m_FileIcon;
};
```

Scene Hierarchy Panel

```
class SceneHierarchyPanel {
public:
        SceneHierarchyPanel() = default;
        ~SceneHierarchyPanel();
        SceneHierarchyPanel(const Ref<Scene>& scene);

        void SetContext(const Ref<Scene>& scene);

        void OnImGuiRender();
        Entity GetSelectedEntity() const;
        void SetSelectedEntity(Entity entity);

        template<typename T, typename UIFunction>
        void DrawComponent(const std::string& name, Entity entity, UIFunction uiFunction);

        template<typename T>
        void AddIfHasComponent(std::string name);
private:
        void DrawEntityNode(Entity entity);
        void DrawComponents(Entity entity);
private:
        Ref<Scene> m_Context;
        Entity m_SelectionContext;
};
```

B.5 Physics

Physics

```
class Physics {
public:
    Physics();

    void Init3DWorld();     // With BulletPhysics
    void Init2DWorld();     // With Box2D
    void Delete3DWorld();   // With BulletPhysics
    void Delete2DWorld();   // With Box2D
    void AddRigidbody(void* bodyType);

    template<T>
    void AddCollider();
private:
    btDynamicsWorld* m_PhysicWorld;
        btCollisionConfiguration* m_PhysicConfig;
        btDispatcher* m_PhysicDispatcher;
        btBroadphaseInterface* m_PhysicBroadphase;
        btConstraintSolver* m_PhysicSolver;

        b2World* m_Physics2DWorld;

        int ITERATIONS_PER_SECOND;
        float STEP_TIME;
        float DEFAULT_LINEAR_SLEEPING_THRESHOLD, DEFAULT_ANGULAR_SLEEPING_THRESHOLD;
}
```

B.6 Sound

Audio

```
class AudioManager {
public:
        struct Emisor {
                glm::vec3 SoundPos;
                glm::vec3 SoundVel;
        };

        static AudioManager* GetInstance();
        static bool SetupInstance();
        static void Clean();

        ~AudioManager();

        void PlaySound(const char* path, int nChannel);
        void PlayMusic(const char* path, int nChannel, bool loop);

        void PauseChannel(int nChannel);
        void StopChannel(int nChannel);
        void SetVolume(float vol, int nChannel);

        bool IsPlaying();
        bool IsPlayingChannel(int nChannel);

        void Update();
        void UpdateListener(const glm::vec3& position, const glm::vec3& velocity,
                            const glm::vec3& forward, const glm::vec3& up);
        void UpdateSound(const glm::vec3& position, const glm::vec3&  velocity,
                         int nChannel, int numObj);

        int AddEmisor(const glm::vec3& position, const glm::vec3& velocity);
        void RemoveEmisor(int numObj);
private:
        FMOD::System* system;
        FMOD_RESULT result;

        Emisor emisores[32];
        glm::vec3 listenerVelocity, listenerUp, listenerForward, listenerPos;
        bool activo[32];
        FMOD::ChannelGroup* channelGroup;
        FMOD::Channel* channels[32];

        static AudioManager* instance;
}
```

B.7 Renderer

Renderer

```
class Renderer {
public:
        static void Init();
        static void OnWindowResize(uint32_t width, uint32_t height);

        static void BeginScene(const Camera& camera, const glm::mat4& transform);
        static void BeginScene(const EditorCamera& camera);

        static void EndScene();
        static void Shutdown();

        static void Flush();
        static void FlushAndReset();

        // Primitives
        static void DrawSprite(glm::mat4& transform, Sprite& src, Ref<Shader> shader, int ID);
        static void DrawCube(glm::mat4& transform, Ref<Shader> shader);
        static void DrawLightCube(glm::vec3 position, glm::vec3 size, glm::vec3 color);
        static void DrawMesh(Ref<Mesh> model, glm::mat4& transform, Ref<Shader> shader);
        // Lighting
        static void BindDirectionalLight(Ref<Shader> shader, DirectionalLight& light);
        static void BindPointLight(Ref<Shader> shader, PointLight& light, int i);
        static void BindSpotLight(Ref<Shader> shader, SpotLight& light, int i);
};
```

Renderer Commands

```
class RenderCommand {
public:
        static void Init() {
                glEnable(GL_BLEND);
                glEnable(GL_DEPTH_TEST);
                glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
        }

        static void SetViewport(uint32_t x, uint32_t y, uint32_t width, uint32_t height);
        static void SetClearColor(const glm::vec4& color);
        static void Clear();
        static void DrawIndexed(Ref<VertexArray>& vertexArray, uint32_t indexCount = 0);
};
```

Camera

```
class Camera {
public:
        Camera() = default;
        Camera(const glm::mat4& projection);

        virtual ~Camera() = default;
        virtual const glm::mat4& GetProjection()  const { return m_Projection; }
protected:
        glm::mat4 m_Projection = glm::mat4(1.0f);
};
```

Editor Camera

```
class EditorCamera : public Camera      {
public:
        EditorCamera() = default;
        EditorCamera(float fov, float aspectRatio, float nearClip, float farClip);

        void OnUpdate(Timestep ts);
        void OnEvent(Event& e);

        void SetViewportSize(float width, float height);
        // Getters and Setters
private:
        void UpdateProjection();
        void UpdateView();

        bool OnMouseScroll(MouseScrolledEvent& e);

        void MousePan(const glm::vec2& delta);
        void MouseRotate(const glm::vec2& delta);
        void MouseZoom(float delta);

        glm::vec3 CalculatePosition() const;

        std::pair<float, float> PanSpeed() const;
        float RotationSpeed() const;
        float ZoomSpeed() const;
private:
        float m_FOV, m_AspectRatio, m_NearClip, m_FarClip;

        glm::mat4 m_ViewMatrix, m_Position, m_FocalPoint;
        glm::vec2 m_InitialMousePosition;

        float m_Distance, m_Pitch, m_Yaw ;
        float m_ViewportWidth, m_ViewportHeight;
```

B.8 Scene

Entity

```
class Entity {
public:
        Entity() = default;
        Entity(const Entity& entity) = default;
        Entity::Entity(entt::entity handle, Scene* scene)
                : m_EntityHandle(handle), m_Scene(scene) { }

        template<typename T, typename... Args>
        T& AddComponent(Args&&... args);

        template<typename T, typename... Args>
        T& AddOrReplaceComponent(Args&&... args);

        template<typename T>
        T& GetComponent();

        template<typename T>
        bool HasComponent();

        template<typename T>
        void RemoveComponent();

        operator bool() const;
        operator entt::entity() const;
        operator uint32_t() const;

        UUID GetUUID();
        const std::string& GetName();
        Scene* GetScene();

        bool operator==(const Entity& other) const;
        bool operator!=(const Entity& other) const;
private:
        entt::entity m_EntityHandle{ entt::null };
        Scene* m_Scene = nullptr;
};
```

Components

```
struct IDComponent {
        UUID ID;
};

struct TransformComponent {
        glm::vec3 Translation = { 0.0f, 0.0f, 0.0f };
        glm::vec3 Rotation = { 0.0f, 0.0f, 0.0f };
        glm::vec3 Scale = { 1.0f, 1.0f, 1.0f };

        glm::mat4 GetTransform() const {
                glm::mat4 rotation = glm::toMat4(glm::quat(Rotation));
                return glm::translate(glm::mat4(1.0f), Translation)
                    * rotation * glm::scale(glm::mat4(1.0f), Scale);
        }
};

struct TagComponent {
        std::string Tag;
};

struct CameraComponent {
        SceneCamera Camera;
        bool Primary = true;
        bool FixedAspectRatio = false;
};

struct SpriteRendererComponent {
        glm::vec4 Color = glm::vec4(1.0f);
        Ref<Texture2D> Texture;
        float TilingFactor = 1.0f;
};

struct CubeMeshComponent {
        Ref<Cube> cube;
};

struct MeshComponent {
        Ref<Mesh> mesh;
        std::string meshRoute;
};

struct LightAttenuation {
        float Constant;
        float Linear;
        float Exp;
};

struct DirectionalLight {
```

```
49          glm::vec3 Direction, Color;
50          float AmbientIntensity, DiffuseIntensity
51          ;
52          bool ViewSource = false;
53          int index = 0;
54  };
55
56  struct PointLight {
57          glm::vec3 Position, Color;
58          float AmbientIntensity, DiffuseIntensity;
59
60          LightAttenuation Attenuation{1.0f, 10.0f, 20.0f};
61          bool ViewSource = false;
62          int index = 0;
63  };
64
65  struct SpotLight {
66          glm::vec3 Direction, Position, Color;
67          float CutOff, OuterCutOff, AmbientIntensity, DiffuseIntensity;
68
69          LightAttenuation Attenuation{ 1.0f, 10.0f, 20.0f };
70          bool ViewSource = false;
71          int index = 0;
72  };
73
74  struct MaterialComponent {
75          Ref<Shader> shader;
76          std::string shaderRoute;
77          int index = 0;
78  };
79
80  class ScriptableEntity;
81  struct NativeScriptComponent {
82          ScriptableEntity* Instance = nullptr;
83          ScriptableEntity*(*InstantiateScript)();
84          void(*DestroyScript)(NativeScriptComponent*);
85
86          template<typename T>
87          void Bind()
88  };
89
90  struct AudioComponent {
91          bool ReceiveEvent(Event& event);
92          int numObj;
93          glm::vec3 pos, vel;
94          std::string audioRoute;
95
96          void PlayMusic(std::string path, int channel);
97          void PlayMusic(std::string path);
98          void RemoveEmisor();
```

```
99  };
100
101
102 struct AudioListenerComponent {
103         glm::vec3 forward, up, pos, vel;
104         bool ReceiveEvent(Event& event) { return false; }
105 };
106
107 enum class BodyType { Static = 0, Kinematic, Dynamic };
108 struct Rigidbody3DComponent {
109         BodyType Type = BodyType::Static;
110         bool FixedRotation = false;
111         // Storage for runtime
112         btRigidBody* RuntimeBody;
113 };
114
115 struct Rigidbody2DComponent {
116         BodyType Type = BodyType::Static;
117         bool FixedRotation = false;
118         // Storage for runtime
119         void* RuntimeBody;
120 };
121
122 struct BoxCollider3DComponent {
123         glm::vec3 Offset, Size;
124         float Density, Friction, Restitution, RestitutionThreshold;
125         // Storage for runtime
126         void* RuntimeFixture = nullptr;
127         btCollisionShape* Shape = nullptr;
128 };
129
130 struct BoxCollider2DComponent {
131         glm::vec2 Offset, Size;
132         float Density, Friction, Restitution, RestitutionThreshold;
133         // Storage for runtime
134         void* RuntimeFixture = nullptr;
135 };
```

Scene Serializer

```
class SceneSerializer {
public:
        SceneSerializer(const Ref<Scene>& scene);

        void Serialize(const std::string& filepath);
        void SerializeRuntime(const std::string& filepath);

        bool Deserialize(const std::string& filepath);
        bool DeserializeRuntime(const std::string& filepath);
private:
        Ref<Scene> m_Scene;
};
```

Scene Camera

```
class SceneCamera : public Camera {
public:
        enum class CameraType { Perspective = 0, Orthographic = 1, FirstPerson = 2,
                                TopDown = 3, RPG = 4, ActionRPG = 5 };
public:
        SceneCamera();
        virtual ~SceneCamera() = default;

        void SetOrthographic(float size, float nearClip, float farClip);
        void SetPerspective(float verticalFOV, float nearClip, float farClip);
        void SetViewportSize(uint32_t width, uint32_t height);
public:
        // Gettes and Setters ...

        CameraType GetProjectionType() const;
        void SetProjectionType(CameraType type);
private:
        void RecalculateProjection();
private:
        // Perspective
        CameraType _projectionType = CameraType::Orthographic;
        float _perspectiveFOV = glm::radians(45.0f);
        float _perspectiveNear = 0.01f, _perspectiveFar = 1000.0f;
private:
        // Orthographic
        float _orthographicSize = 10.0f;
        float _orthographicNear = -1.0f, _orthographicFar = 1.0f;
        float _aspectRatio = 0.0f;
};
```

Scene

```
class Scene {
        public:
                Scene() = default;

                Entity CreateEmptyEntity(const std::string& name);
                Entity CreateEntityWithUUID(UUID uuid, const std::string& name);
                void DestroyEntity(Entity entity);
                void DuplicateEntity(Entity entity);
                void RenderSceneEntities();

                void OnRuntimeStart();
                void OnRuntimeStop();
                void OnUpdateRuntime(Timestep ts);
                void OnUpdateEditor(Timestep ts, EditorCamera& camera);
                void OnViewportResize(uint32_t width, uint32_t height);

                bool GetWorldType();
                void SetWorldType(bool worldType);
                std::list<Entity> GetEntitiesOnScene();
                Entity GetPrimaryCameraEntity();
                static Ref<Scene> Copy(Ref<Scene> scene);
        private:
                template<typename T>
                void OnComponentAdded(Entity entity, T& component);
        private:
                entt::registry m_Registry;
                std::list<Entity> enttOnScene;
                uint32_t m_ViewportWidth = 1280, m_ViewportHeight = 720;
                bool World3D = true;

                friend class Entity, SceneSerializer, SceneHierarchyPanel;
};
```