UNIVERSITY OF MACEDONIA
UNDERGRADUATE PROGRAM OF STUDIES
DEPARTMENT OF APPLIED INFORMATICS

A ROADMAP TARGETED TO NOVICES FOR 2D GAME ENGINE DEVELOPMENT UTILIZING OPEN-SOURCE LIBRARIES AND APIs

Thesis of

Konstantinidis Konstantinos
Kostidis Ioannis

Thessaloniki, June 2023

A ROADMAP TARGETED TO NOVICES FOR 2D GAME ENGINE DEVELOPMENT UTILIZING OPEN-SOURCE LIBRARIES AND APIs

Konstantinidis Konstantinos
Kostidis Ioannis

Thesis

submitted for the partial fulfillment of its requirements

Undergraduate Degree In Applied Informatics

Supervising Professor:
Xinogalos Stylianos

Approved by the three-member examination board on 23/06/2023

Xinogalos Stylianos	Chatzigeorgiou Alexandros	Ampatzoglou Apostolos
.....	.....	.....

Enter your full name here

Konstantinidis Konstantinos	Kostidis Ioannis
.....	.....

Abstract

In an age defined by rapid technological advancements, gaming has evolved remarkably with better consoles, immersive virtual reality (VR) experiences, enhanced graphics, captivating audio and advanced game mechanics. This change has encouraged companies worldwide to develop their own game engines for creating and releasing unique games. This paper proposes a roadmap on creating a functional 2D game engine, using programming languages like C++ and C#, technologies like OpenGL and YAML, and various open source libraries and APIs. This roadmap aims to answer questions like: What are the essential subsystems needed to make a game engine function? How can they be coded? What open source libraries can be used? Our results indicate that the development of a game engine requires significant dedication and effort as well as knowledge in various fields like programming, mathematics, physics and more. However, building a game engine is expedited by the use of excellent open source libraries for implementing core subsystems. The roadmap presented in this paper can support ambitious developers without prior experience in game engine development to start building their own engine, as well as instructors teaching courses on software engineering and game programming on designing relevant team projects that can both motivate students and support them in acquiring relevant knowledge and skills through a rewarding experience.

Keywords: Game engine development, 2D Game engine, Software, Game engine subsystems, C++, OpenGL, Game architecture, Video Games, Gaming

Contents

List of figures

List of tables

List of code snippets

1 Introduction

1.1 Problem - Importance of the issue

What is a Game Engine? A game engine is a software framework that provides the necessary tools and functionality to design, develop and publish video games [1–5]. It serves as the foundation for game developers to build their games on top of, offering resources and features to handle many of the complex technical aspects of game development, such as graphics rendering, physics simulation, and sound processing. By separating game-specific logic (such as level design and character behavior) from the generalizable logic and other technical details (such as the rendering process), game developers can focus on creating an engaging game experience, while leaving the technical details to the game engine [1–10].

It is worth noting that not all the game engines provide the same functionality for the end user. Powerful engines (e.g. Unity, Unreal Engine) offer such a huge variety of tools and assets, including marketplaces where developers can find 3D models, textures and plugins. They also provide visual scripting for creating game logic without coding and tools like shader graphs to create stunning visual effects. With these resources, developers can fully develop and publish games without writing a single line of code. In comparison, other smaller game engines or ones with a specific purpose could even be as simple as just a platform which will display the graphics. Some of the most commonly used game engines are presented in Figure 1.

Figure 1. Some of the most used game engines. Figure by: Unity vs Unreal

Why would anyone want to make their own game engine? Why would someone not prefer to create a game rather than an engine? Well, for novice game developers curiosity is the number one reason [4]. Creating a custom engine is a valuable learning experience. Building a game engine can be a good way to gain a deeper understanding of the technical aspects of game development, and it will definitely help any developer improve their programming skills. On the other hand, for experienced programmers working for large development companies, the most common reason would be that the already existing game engines provide limited tools, and the tools needed to create their games, may not be supported by the existing options [1,4]. The best thing to do in that case would probably be to create their own game engine, which will be specifically built in order to make the games they have in mind, by providing the basic layer of functionalities. Some game studios also choose to create their own engines to avoid the license costs of the existing game engines [1].

After finding the reasons for wanting to create a game engine either with a team or alone,

the first question that comes to mind is “How can someone develop a game engine?”. First and foremost, building an engine from scratch is a daunting task, which requires a lot of time, and a very good knowledge in programming, computer graphics, mathematics and other technical fields. As stated before, game engines often provide a variety of built-in features and tools that can be customized and extended to fit the specific needs of a game developer. Some of the subsystems a game engine should provide are the following [5,8]:

1. A Rendering Engine to communicate with the GPU, create graphics and visualize every element of a game.
2. A Math Engine to handle linear algebra, geometric and other complex operations.
3. An Entity Component System (ECS) to create entities in a game, give them components and keep track of them.
4. A Physics Engine to simulate real-world physics for the entities of a game.
5. An Audio System to support music and sound effects.
6. An Animation System to animate the entities of a game.
7. A Frame Timing Control System to control when the updates and rendering happens.
8. An Event System to efficiently dispatch and route events, ensuring accurate and effective handling.
9. An Input System to respond to mouse clicks, keyboard key presses etc.
10. A Scripting Engine to create game logic and behavior.
11. An Assets Management System to import and export models, textures and other game assets.
12. A Networking System to make it possible to create online multiplayer games.
13. Platform Support for various platforms and devices, such as PCs, consoles, mobile devices, and VR/AR devices.

There are actually a lot more subsystems a game engine could have to provide further tools and functionalities like: Serialization, Particle, Profiling, Text Rendering, Universally/Globally Unique Identifiers (UUID/GUID), User Interface (UI), Logging, Scene Management, Post Processing and many more.

But with what order and how should the aforementioned subsystems be implemented? How should someone approach this? These are questions that have not been adequately investigated and this paper tries to contribute to this field by providing a roadmap for a hands-on implementation of a 2D game engine. To start with, an engine does not need to have all these features, and it’s best to only create them as needed rather than upfront. Based on our experience, the best way to build a game engine is to create it while making the game. We decided to begin by getting the basics in like the rendering, the math and the physics engines and then start creating our first game. We chose this approach to make sure that the engine will provide to us or another developer only the features that our game will need. Additionally, through experimentation we found out that creating a second game right after finishing the first one makes it possible to figure out what was already implemented in the previous game, and then turn it into a general library or framework, instead of coding it again from scratch.

Before making any steps though, we advise everyone to spend some time researching and doing some reading sessions to find valuable information. A key consideration is deciding whether to code every single thing from scratch, or if it would be better to use existing libraries. We can assure anyone that there are some outstanding libraries out there making it unnecessary to code everything from the beginning.

Overall, game engines are a crucial tool in modern game development, providing developers with the necessary resources to bring their game ideas to life. However, the decision

to develop a custom game engine should not be taken lightly and developers should weigh the advantages and disadvantages very carefully before starting such a project. Before beginning, it is advisable to determine the specific games the engine will support, find out which engine subsystems to develop, and most importantly, never give up.

1.2 Purpose - Objectives

The purpose of this paper is first and foremost educational and then practical. The main objective is to provide people with a better understanding of what a game engine is, how it works, what are the necessary subsystems of it, how each of the subsystems work and how can someone implement them. We thought that the best way to solve these questions was to try and develop our own 2D game engine from scratch. By giving greater details over the technical aspects, we are also hoping to inspire more developers to start building their own game engine, as well as provide valuable material for instructors in terms of designing game development courses and/or team projects for game development and advanced software engineering courses.

1.3 Contribution

Developing a game engine is a valuable experience that can benefit both researchers and game developers. Here are some potential implications of this paper that can prove to be beneficial:

1. *Enhanced knowledge:* Both researchers and practitioners can find insights, methodologies and other technical details that may be valuable for understanding the structure of a game engine and the way it works. The technical details and design decisions discussed in this article could prove to be useful for different kinds of software, making the knowledge gained worth attaining.
2. *Inspiration:* This article can inspire researchers to explore new approaches related to game engine development. This is important since a lot of companies in the game industry wish to have their own custom game engine, as it is the most effective way of developing their games according to their needs.
3. *Deeper understanding:* This paper can serve as an educational resource to both a researcher/developer who already has prior knowledge on how to develop a game engine and an aspiring one.
4. *Increase interest:* Our work can suggest new directions to future research and hopefully increase the interest on the subject of game engine development.
5. *Educational aspect:* This paper could support instructors teaching software engineering and game programming courses, to create team projects that can both motivate students and assist them in acquiring the necessary knowledge and skills.
6. *Guidance and tools:* This paper can prove to be beneficial to practitioners who do not have basic knowledge about game engine implementation. It offers guidance and practical tips that could help them navigate through the first challenges of developing a new custom game engine.
7. *Skill enhancement:* Our paper can contribute to the professional growth of game developers by enhancing their knowledge of game engine architecture. By offering educational materials, multiple examples of building the components of a game engine and a variety of APIs and libraries that can be used to serve a specific purpose, practitioners can enhance their knowledge.

1.4 Basic Terminology

In Table 1, we present some key terms and their corresponding definitions, which will be used throughout this paper. If there is something you do not understand, you should refer back to this table for help.

Table 1: Terms and Terminology

Term	Terminology
2D	2 Dimensional
3D	3 Dimensional
UI	User Interface
ECS	Entity Component System
UUID - GUID	Universally - Globally Unique Identifiers
CPU	Central Processing Unit
GPU	Graphics Processing Unit
AR	Augmented Reality
VR	Virtual Reality
OpenGL	Open Graphics Library
GUI	Graphical User Interface
VAO	Vertex Array Object
VBO	Vertex Buffer Object
IBO	Index Buffer Object
FBO	Frame Buffer Object
UBO	Uniform Buffer Object
Vertex Array Object	An object that stores information about the index buffer object (IBO), and the vertex buffer objects (VBOs), including the buffer layout and data types, as well as any associated vertex attribute data.
Vertex Buffer Object	A buffer object which contains information about the vertices that make up a 2D/3D model e.g. color, position, texture, custom data used by a shader.

Index Buffer Object	A buffer object which contains information about which vertices will be drawn and in what order. It uses the index of each vertex in the VBO as a value.
Uniform Buffer Object	A buffer object which contains uniform data for a shader program. It is used to share uniforms between different shader programs, as well as quickly change between sets of uniforms for the same program object.
Frame Buffer Object	A buffer object which is used to manage rendering operations and facilitate off-screen rendering. It provides a mechanism for creating and managing an alternative rendering target such as a texture, instead of the default framebuffer (which is typically the main screen).
Render pipeline	The sequence of steps a graphics system needs to perform to render a 3D/2D scene to a 2D screen.
Shader	A program designed to run on some stage of a graphics processor. They provide the code for certain programmable stages of the rendering pipeline. They are used to render different pixels and detail shadows, lighting, texture gradients, and more.
Draw Call	A draw call is a command issued by a rendering engine or graphics API that tells the renderer what to render and how to render it.
Uniform	A uniform is a global shader variable which acts as a parameter that the user of a shader program can pass to that program. Uniforms are accessible at all the stages of the graphics pipeline.
Entity	An entity is a real time object that is different from others. It can be defined using its attributes. Every object in a game scene is considered an entity.
FOV	Field Of View
Field Of View	The field of view is the extent of the observable world that is seen at any given time. It also describes the angle through which one can see that observable world.

1.5 Structure of the study

The rest of the paper is organized as follows. The work of other developers and the relevant sources and literature are introduced in Section 2, while the methodology we followed during the development of our engine, is described as detailed as possible in Section 3. In Section 4 we delve deeper into our engine, by showcasing the subsystems we implemented and suggesting different APIs and libraries someone could consider, when implementing their own one. Section 5, presents a table showcasing all the classes used in order to develop the game engine. Section 6, serves as a guide, to show how we made one of our games using the engine. We also use this section to showcase some of our projects, talk about the testing and evaluation of our engine while also mentioning our future plans for improving the engine. In Section 7 we present the limitations and constraints we encountered during the development of our game engine. Finally Section 8 offers a summary of our thesis, including our final thoughts on the project.

2 Related Work

2.1 Literature and Research Papers

Even though more and more games are brought to life each day by both development companies (Electronic Arts, Ubisoft, Activision, etc.) and independent game developers, the literature focused on the game engine development field remains limited. So, after thorough research we discovered some noteworthy books and articles.

Gregory [11] authored a book entitled “Game Engine Architecture” which presents both the theoretical and practical concepts of a game engine. By also explaining in great detail most of the low-level technical subsystems within game engines and how the gameplay works, it has become a definitive guide to professional game development.

Thorn [12] has also written a book specifying how to design and develop a game engine. Offering a step-by-step journey from selecting the development environment to implementing all the core engine subsystems, he has managed to create a book for both aspiring and professional developers.

Bishop et al. [13] back in 1998 introduced a 3D game engine called “NetImmerse”, arguing that a high-level programming interface does not necessarily compromise performance. In their paper they describe the components of the engine providing insights into the design and implementation of a game engine. Through a detailed examination and experiments, they learned some lessons such as the distinction between game content and the game engine, the importance of creating an engine for a particular content style and the critical role the graphics engine plays.

Guana et al. [14] presented an innovative approach in their paper focusing on the design and implementation of PhyDSL2, a 2D physics-based game engine. By using modern model-driven engineering techniques, they made a specialized domain-specific language (DSL) hoping to benefit game designers in translating gameplay models into implementation. After successfully building and experimenting with the engine they validated that model-driven technologies can be effectively used in the construction of game engines.

2.2 Open Source Game Engine Projects

As technology advances, fortunately, developers are increasingly contributing to open-source projects more than ever. There are actually thousands of free and open-source game engines available on GitHub where anyone can easily browse around the code and see how the engine got implemented and how it works. Below we provide a list with some game engines that started as a small project and have ended up with thousands of users actively using and supporting them. The selection criteria for these engines include the number of stars on GitHub, the quantity of commits, and the engine's ongoing activity:

- Godot Engine: A 2D and 3D cross-platform game engine [15].
- Bevy Engine: A data-driven game engine built in Rust [16].
- OpenRA Engine: A strategy game engine for early Westwood games [17].
- Pyxel Engine: A retro game engine for Python [18].
- Minetest Engine: A voxel game engine [19].
- Ebitengine Engine: A 2D game engine written for Go [20].
- PlayCanvas Engine: A WebGL based engine to run games on browsers [21].
- Flame Engine: A 2D game engine based on Flutter [22].
- GDevelop Engine: A no-code 2D and 3D game engine [23].
- Stride Engine: A 2D, 3D and VR game engine written in C# [24].

2.3 Educational Video Tutorials

There is not as much relevant literature available in this field compared to other tech areas. What caught our attention though, was some instructive YouTube educational video series, where the creators took it upon themselves to share their journey of developing game engines in an educational style while also offering their insights and expertise.

Khatami [25], runs a YouTube channel by the name of “Game Engine Series”. He has created a video series which started in 2020 and consists of more than 57 hours of content (as of August 12, 2023). In this series, Khatami showcases the process of building a game engine using both the C++ and C# programming languages. In each video, he writes, shows and explains the code, to ensure that everyone can both implement and understand everything.

Similarly, Chernikov [26], a former game engine developer of Electronic Arts (EA), thought that it would be a great idea to build his own 3D game engine for his game studio, while also develop a 2D game engine and showcase this journey with an educational aspect on YouTube, in order to help everyone interested to build their own engine. The series started in 2018 and consists of more than 105 hours of available content (as of August 12, 2023). Chernikov's tutorials carefully explain each line of code, while he also teaches what each of the game engine subsystems does, how it works and how can someone both implement and connect it with the rest to make a functional end product.

On the other hand, Ambrosio [27] chose to create a tutorial series to help developers create their own 2D game engine using the Java programming language. His tutorial series launched in 2020 and concluded in 2021. The main objective of the series was to create a fully functional game engine and afterwards use it to create and distribute the classic Nintendo Entertainment System (NES) Super Mario Bros game, while also having an educational aspect hoping to inspire and help other developers to do the same.

Pouhela [28], in 2020 uploaded a small video series where most of the videos are under 20-30 minutes, showcasing how someone can build a 2D game engine from scratch using the C++ programming language and the Simple DirectMedia Layer (SDL) library. Even though the videos do not last long, they offer an effective resource to help any developer understand and build a basic functional engine.

In 2021, Prog [29] decided to record and upload his personal game engine journey. He made a small educational series by the name of “Let's make an engine!” using the C++ programming language. Prog managed to make a fully functional cross-platform engine while also explaining each step needed to achieve this result.

2.4 Building Individual Subsystems

What’s really fascinating about a game engine is the fact that each of its subsystems constitutes a whole project by itself. Large tech companies dedicate lots of resources to small teams, with each team focusing on developing a specific subsystem rather than the entire engine. This approach of breaking things down into smaller parts eventually leads to smoothly combining multiple top-quality subsystems to create a great game engine.

So even if there are not many related projects available for the public, there are lots of literature, guides and tutorials available for creating each of the necessary subsystems independently. Eberly [30], has authored an instructive book talking about computer graphics and rendering. In his book he explains in depth how a rendering engine works, while also offering explanations on scene graphs, shader-based effects, animation, physics simulation, collision between objects and even memory management.

Bauchinger [31], wrote a master's thesis explaining what a rendering engine is, how it works and how someone can design and implement a modern one. Meanwhile, Ambrosio [32], chose to create and upload a YouTube video series about coding a fully functional 2D physics engine from scratch using the Java programming language. Within this series Ambrosio addresses 2D implementation, collision handling, rotations, raycasting, gravity and other force interactions, constraints, impulses and more.

Other interesting examples are the work of Gutekanst [33] and Colson [34], where both have written some articles where they showcase the construction of an entity component system (ECS). By giving coding snippets and practical examples, their work serves as a stepping stone, helping developers create their own relatively solid ECS. By following these guides and getting more and more knowledge about each of the subsystems, while also creating them independently, at the end by trying to connect them all together, the final product known as the game engine will be built.

3 Methods and Materials

3.1 Data Collection

In order to develop a game engine, we needed to understand how they work. So, to start with, the research methodology we followed to develop the engine, began with a literature review to gain a deeper understanding of what a game engine is and how it works. This involved reading various books, articles and scholarly publications to find the fundamental concepts and principles involved in the game engine development. Additionally, we extensively searched through forums, blogs and online videos to find out how people created their game engine, how difficult it was, how much time it took them and also to gain guidance and learn some tips and tricks.

Once we had established a solid foundation of knowledge, we proceeded with the practical aspects of game engine development. This involved studying guides, following tutorials, watching lots of educational videos, using popular game engines such as “Unity” and “Godot”, and studying the code of existing open source engines we found available on “GitHub”. By using these resources, we were able to develop a clear understanding of the various technical and design considerations involved in building a game engine, and we managed to find out the strategies used by other developers in the field. This approach helped us ensure the validity of our research, and it was enough to lay the groundwork for the development of our own game engine.

3.2 Method

3.2.1 Programming Language Selection

In order to develop a game engine, the choice of which programming language or languages someone might want to use must be made. After reading some articles, blogs and forums we found out that “C++” is a common choice when it comes to choosing a language for coding game engines. Many popular game engines like Unity, Unreal Engine, Godot and CryEngine have been written in C++ [152-156]. C++ is an object-oriented language and it also supports manual memory allocation and deallocation through features like pointers. Such a feature is essential to a game engine and it was the main reason we decided to use C++ as our coding language in this project.

To gain as much proficiency as we could in C++, we decided to follow both beginner and advanced tutorials. Fortunately, there are a lot of high-quality tutorials and educational series available for free online. Based on our own experience, we highly recommend the following resources:

- LearnCpp.com [35]: This free website has been teaching C++ programming since 2007. With three main authors and an active community of readers providing constant feedback and support, it offers reliable instructions.
- Learn-cpp.org [36]: A free interactive tutorials website that has tutorials for more than 10 different programming languages. There is also an accompanying “GitHub” repository which has over 3000 stars and 2000 forks, indicating its popularity.
- freeCodeCamp.org YouTube Channel [37]: This channel offers a 31-hour video course on modern C++ programming. This free course covers both beginner and advanced concepts, ensuring an extensive learning experience.
- Code Beauty YouTube Channel [38]: This channel offers a 10-hour video which is designed to take anyone from a beginner to an advanced C++ developer.

- Bro Code YouTube Channel [39]: This channel provides a 6-hour video that serves as an effective introduction to the C++ programming language. BroCode channel is dedicated to offering free education.

Overall, our study of C++ was a very important part of our research methodology as it helped us develop the technical skills required to build a game engine.

3.2.2 Game Engines

Understanding the significance of practical skills and the need to gain more experience before starting a complex project like this one, we made the decision to use a game engine and create additional applications and games. This approach helped us gain a deeper understanding of the inner workings of game engines. To gain more practical experience, we followed a series of tutorials focused on developing primarily 2D games using the “Unity” game engine. By creating those games, we managed to improve our understanding of how a game engine works and how difficult it is to develop a quality game, while we also improved our skills in writing code using the C# language. We also got more inspiration on how the user interface of an engine should look like, and what tools and features should be provided to the end user in order to help them create their games.

3.2.3 OpenGL

To display graphics on the screen, it is necessary to have a system that helps the Central Processing Unit (CPU) and Graphics Processing Unit (GPU) communicate with each other. We should note that considerable time was invested in selecting the appropriate Application Programming Interface (API) for this task. After careful consideration, we chose the “OpenGL API”, which is a widely used technology in this field and also the easiest to learn to use and integrate in a project. To discover more about the “OpenGL API” we followed a series of carefully selected tutorials. The aim of the tutorials was to provide us with an understanding of its essential features and functionalities.

3.2.4 Technologies

The environments used to develop the project were “Visual Studio 2019 Community Edition” and “Visual Studio Code”. We also used “GitHub” to create a repository and make it possible to track and save our changes. Finally, we used “GitHub Desktop” as the main way to communicate with our repository.

4 Game Engine Design and Development

4.1 Game Engine Subsystems

Game engines typically provide a range of built-in features and functionalities to help as much as possible the game developers. There are lots of subsystems an engine could have, and of course not all of them must be created to call the engine a complete and functional one. The basic subsystems a game engine should have to provide developers the necessary tools to create a simple game include:

- A rendering system, to create graphics and visualize every element of a game.
- A math system, to handle all mathematical operations.
- A physics system, to simulate real-world physics between scene entities.
- A frame timing control system, to control when the updates and rendering happens.
- An event system, to handle events effectively.
- An input system, to respond to mouse clicks, keyboard key presses etc.

Assuming that all these subsystems are implemented correctly and provide enough features, the game developers who are going to use the engine should be able to create a simple game and publish it with ease. However, in today's era, aspiring to develop a "AAA Game" typically needs a high budget, and an engine with far more subsystems than just the ones already mentioned [41]. Other subsystems a game engine should provide to the end user are:

- An entity component system, to create entities in a scene, give them components and keep track of them.
- An animation system, to animate the scene entities.
- A physics system, to simulate real-world physics between scene entities.
- A scripting system, to be able to create game logic and entity behavior through scripting.
- An assets management system, to import and export game assets.
- A networking system, to develop multiplayer games.
- A platform support system, to publish the final game for different platforms.
- A serialization system, to convert the game data into a format that can be easily read, stored and reconstructed.
- A particle system, to simulate visually appealing effects like fire, smoke etc.
- A profiling system, to analyze the performance of both the game engine and the game.
- A text rendering system, to handle the rendering of text within a game.
- A universally/globally unique identifiers system, to generate and manage unique identifiers for game assets, objects, or entities.
- A user interface system, to display and manage the graphical elements of the game engine.
- A logging system, to record events, errors, and debug information.
- A scene management system, to organize and manage different game scenes.
- A post processing system, to improve the game’s visual quality by having effects like depth of field, motion blur, color grading etc.
- A multithreading system, to take advantage of multiple processor cores to perform tasks simultaneously.
- A build system to ensure that the game engine is ready to be used. This system is used for tasks like integrating external libraries, generating project files, simplifying the process of packaging the engine and all of its associated assets, and also building the final project in order to make it possible to publish it.

Most of these subsystems are complementary to the engine, so there is no need to implement them. It should also be mentioned that there are pre-existing free and open-source APIs and libraries which anyone can read and integrate to their code, instead of implementing each of the subsystems on their own.

4.2 Game Engine Overview

4.2.1 Introduction - General Engine Overview

Our aim was to learn how an engine works, how the subsystems of it work and are tied together, as well as how to code them or at least some of them. Our intention was not to create yet another 2D game engine for commercial purposes. First and foremost, we thought that we should start by creating the main game loop. This loop is designed to continuously repeat a defined set of instructions e.g. rendering and updating, until the user chooses to exit the application. Code 1, is used to show how a main loop could look like. However, before entering the main loop, certain initialization processes need to be carried out.

```
void Application::Run()
{
    // Game Loop
    while (isRunning)
    {
        float time = Time::GetTime();
        timestep = time - lastFrame;
        lastFrame = time;

        Update(timestep);
        UIRender();
    }
}
```

Code 1. A simple game loop

To begin our setup process, we first set a fixed directory for the assets, which is used to display all the project files and folders. Subsequently, we initialized the libraries needed to be able to use “OpenGL” methods and create the main user window. We continued with setting the event callbacks our engine supports, in order to be able to handle events like user input, window resizing, mouse movement etc. Moving forward, our attention turned on initializing the renderer, so we could use the “OpenGL API” to its fullest. We then set up a system to establish a definite order for the updates, events and rendering. Lastly, we focused on setting up the “DearImGui” library, which plays a vital role in rendering and providing a fully functional user interface.

With everything in place, we proceeded to create a new blank scene for the user to begin their project. Necessary variables and fields are initialized and configured, including the setup of the camera which will be used while editing a scene. Having completed all the necessary configurations and preparations, our game engine loop begins running, marking the start of the engine’s execution.

The game loop is used to keep track of each frame, by updating the scripts and physics of the scene entities in order to change their behavior, rendering the User Interface (UI) panels and processing all pending events. Using the help of an Entity Component System (ECS), we efficiently update and render each frame of the game scene. Also, we integrated an existing physics library which is used for all the physics calculations like gravity, mass, friction, and

collision detection between entities. We then used the “Mono” project to connect the core game engine with the scripting engine, providing the option of creating entities and updating their behavior through C# scripts. Ultimately, the game loop ends by rendering the scene and the UI each frame while also handling all pending events.

For all the math calculations inside the game engine a math engine is used. More specifically we integrated the “OpenGL Mathematics (GLM)” library. Furthermore, to render the entities on the screen, we have both implemented and used existing shaders. Our engine also provides users the ability to open, save, and create scenes. To ensure that everything executes smoothly we used “YAML Ain't Markup Language (YAML)”, a data serialization language which is also used by Unity [40]. Finally, to tie everything together and make it possible to integrate all those libraries in our project, generate the project files and build the project we relied on “Premake” even though a good choice would have also been “CMake”.

Once all the necessary subsystems were implemented and everything was running smoothly, we started making some games. Through this game development process, we were able to identify and resolve issues, add more functionality and support more features. This approach ensured that everything worked properly in order to be able to create a 2D game.

4.2.2 Game Engine Subsystems

4.2.2.1 Rendering Engine

4.2.2.1.1 Introduction

The rendering engine is arguably the most critical subsystem of a game engine. If it lacks quality, flexibility, and speed, the game’s performance and overall experience can be negatively impacted. In a game engine, the renderer serves as a bridge between the CPU and the GPU, making it possible to display visual elements on the screen. Without a renderer, even a single pixel cannot be drawn on the screen. To be able to communicate between the two processing units, an existing API should probably be used. There are numerous APIs available, each with its own strengths and weaknesses. In Table 2, we present the most used graphics APIs, along with their respective developers and some of the platforms on which they are used.

Table 2: Graphics APIs, Developers and Platforms

API	Developers	Platform
DirectX	Microsoft	Windows[42-46] , Xbox[42-46]
Metal	Apple	iOS [42][47-49], macOS [42][47-49], iPadOS [42][47-49], tvOS [42,47,49]
Vulkan	Khronos Group	Windows [42][50-53], Linux [42][50-53], Android [50-53], iOS [50,52], macOS [42,50,52], Nintendo Switch [50,51][54-56], Raspberry Pi [50]
OpenGL	Khronos Group	Windows [42][57-59], Linux [42,57,58], Android [61,62], macOS [42,57,58,60], Nintendo Switch [54-56]
WebGL	Khronos Group	Google Chrome [63,65], Mozilla Firefox [63-65], Opera [65], Safari [63,65], Microsoft Edge [63,65]

4.2.2.1.2 OpenGL API

After spending a considerable amount of time before making the final decision, we chose to use "OpenGL". As novice game engine developers, we thought that using OpenGL as our graphics API was the best fit due to its relatively user-friendly nature, as it is easier to learn than the other APIs, primarily because it has a simpler and more intuitive API [66-68]. It also offers cross-platform support, has extensive documentation and is flexible enough to create a variety of graphical effects [66-68]. Even though it has its limitations such as its lack of built-in support for multithreading, limited debugging tools, and relatively limited support for advanced features like ray tracing, motion blur and anti-aliasing, we found it to be a great starting for gaining insight into building a rendering engine for our game engine [66-68]. So, after spending quite some time reading and watching tutorials created by Joey de Vries [69], Alexander Overvoorde [70], Yan Chernikov [71], Mike Shah [72] and reading the documentation provided by Jorge Rodríguez [73], we started building our own renderer.

To ensure that OpenGL was properly configured in our project, we attempted to draw a triangle. First of all, we generated and bound a "Vertex Array" and then we did the same for the "Vertex Buffer". Next, we created some vertices for the 3 corners of the triangle and used the "glBufferData()" function to send them from the CPU to the GPU. Afterwards we generated and bound the "Index Buffer". Then, we created some indices and sent them to the GPU. Finally, we were ready to draw the triangle. We cleared the buffers using the "glClearColor()" and "glClear()" methods, and by invoking the "glDrawElements()" function we successfully drew our first triangle. Figure 2, depicts a diagram with all the steps needed to draw a simple triangle using OpenGL.

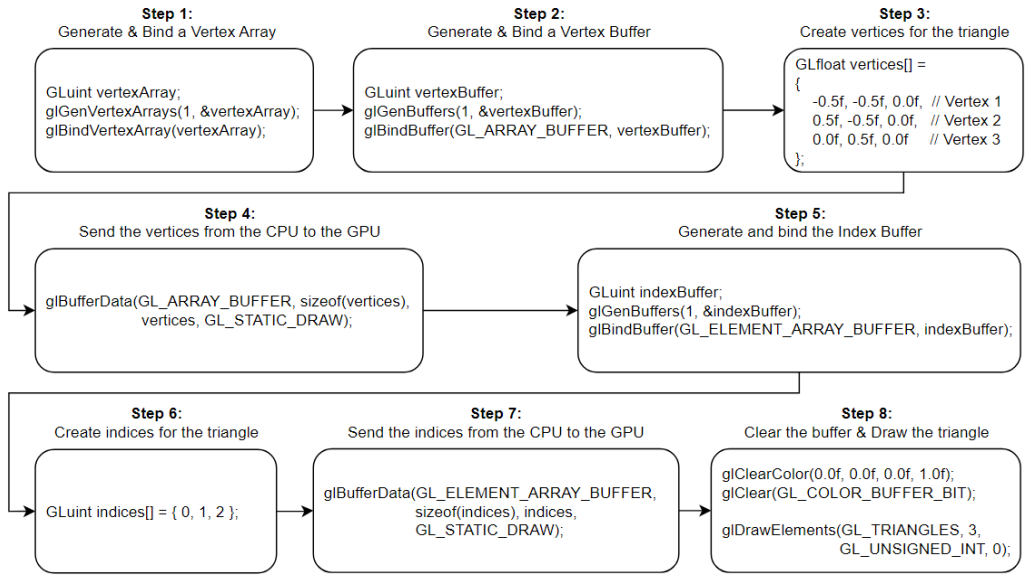

Figure 2. A diagram showing how to draw a triangle using OpenGL

After ensuring that everything was properly set up, we proceeded to expand the renderer. Throughout the process, we discovered that the renderer optimization is an ongoing endeavor. There is always room for improvement or new additions. Creating a simple and functional renderer is not overly challenging. The real difficulty lies in making it fast and flexible. When developing a renderer, several key considerations come into play. It becomes crucial to consider how things such as triangles, lines, and textures will be rendered. Also, creating the illusion of a

camera, determining the rendering order for each entity, minimizing draw calls, and numerous other factors all demand attention.

4.2.2.1.3 Renderer Initialization

So, to start with, upon creating our application, we proceed to initialize our renderer. This involves creating and initializing variables and invoking all the necessary functions. First of all, we call all the required OpenGL methods as Code 2 shows:

```
void OpenGLRenderer::Init()
{
    // Enable blending
    glEnable(GL_BLEND);

    // Set up blend function
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

    // Enable depth
    glEnable(GL_DEPTH_TEST);

    // Enable line smoothing
    glEnable(GL_LINE_SMOOTH);
}
```

Code 2. OpenGL API methods

Then we continue by creating all the "Vertex Array Objects" (VAO) needed to store the corresponding "Vertex" and "Index Buffer Objects" (VBO/IBO). After that, we initialize the different VBOs. We initialize the first one to render lines, the second one to render quads and the last one to render circles. As both circles and quads are made up of triangles, we need to create only 1 IBO. Also, a line is made up of only 2 vertices, so there is no need to create an IBO for it. Having created and set the layout of each VBO, we link the quad and circle VBOs with the IBO.

As we have completed the setup of the buffer and array objects, we continue with the creation of our shaders. Shaders play a vital role in the rendering process, offering a range of functionalities to manipulate the appearance and behavior of objects. While there are various types of shaders available, two of the most used ones are the "Vertex Shader" and the "Pixel (Fragment) Shader".

The "Vertex Shader" primarily handles the processing of individual vertices. It runs for as many vertices as we have. For example, if we want to draw a triangle, the shader will run 3 times, once for each vertex. Its purpose is to receive a single vertex from the vertex stream and generate a single vertex to the output vertex stream. Its main responsibility lies in transforming and manipulating vertex attributes, such as position, color, and texture coordinates [74].

On the other hand, the "Pixel Shader" operates at the pixel level during rendering. It runs for each pixel being rendered on the screen. The pixel shader receives specific information associated with each pixel such as its position, color, texture coordinates, and other relevant data. One of the primary uses of the pixel shader is to calculate the final color for each pixel. It often performs other tasks like image processing and lighting calculations [75-77]. Figure 3, depicts the graphics pipeline stages.

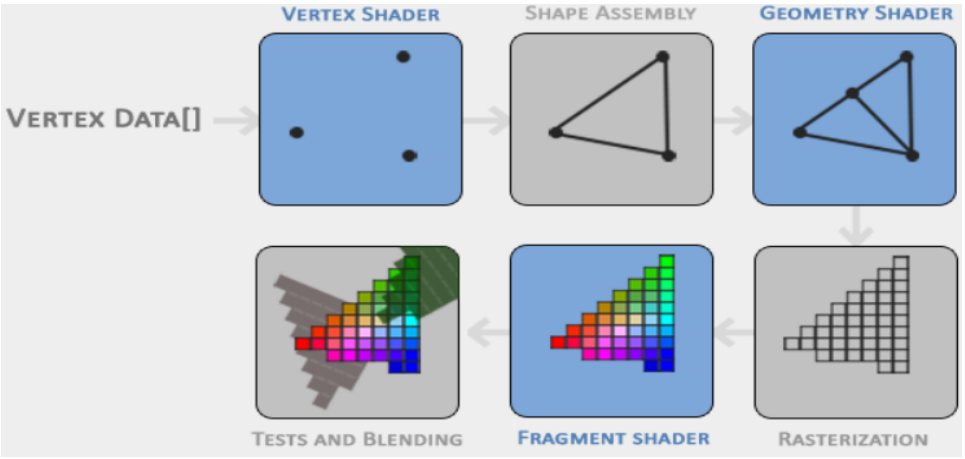

Figure 3. Graphics pipeline stages. Figure by: LearnOpenGL

Using "GLSL" (OpenGL Shading Language), we wrote the shaders needed for our engine and initialized them. To create a shader program for each shader, we followed the example provided by "Khronos Group" [78] with slight modifications. Our shaders support the following functionalities:

1. Line rendering, with the options to pass the position of the line vertices, the color of the line and the line ID.
2. Quad rendering, with the options to pass the position of the quad vertices, the color of the quad, a texture and the quad ID.
3. Circle rendering, with the options to pass the position of the circle vertices, the color of the circle, the thickness of the circle and the circle ID.

After creating all the shaders, the next step involves setting up and initializing a "Frame Buffer Object" (FBO). We configure the FBO with specific dimensions, such as a chosen width and height of 1920x1080, which corresponds to a 16:9 aspect ratio. Additionally, necessary attachments, including texture color format and texture depth format, are specified. Subsequently, we create OpenGL textures for the FBO, we bind the textures and based on the specified format of the pixel data, we attach to them their corresponding color and depth attachments [79].

Once the FBO setup is complete, the focus shifts to creating and rendering a camera, which serves as the editor camera. We set the "field of view", the "aspect ratio", the "near clip" and the "far clip" of the camera. Then, we calculate the orientation and translation of the camera and render it by giving the visual illusion of a dynamic viewpoint. Finally, we create a default texture and we initialize a "Uniform Buffer Object" (UBO) to store the projection view of the cameras that will be created later for the game scenes. This UBO acts as a container to hold the necessary data related to projection and view matrices of the cameras for efficient rendering. With these final steps done, the initialization process of our renderer is complete.

4.2.2.1.4 Scene Rendering

As depicted in Code 1, the main game loop is an endless loop that repeats a set of instructions e.g. rendering and updating, and will stop only when the user quits the application. Our game engine also follows a sequence of steps to update and render each frame. Using an "Update()" method, we update and render the new state of the scene, and then by calling the "UIRender()" method each panel of the User Interface (UI) gets rendered. To make something like that possible, first and foremost we clear the buffers. Subsequently, we bind the FBO we initialized before and we clear the texture which will be used to render the scene on top of.

Depending on the state of the scene we call the appropriate methods. Code 3 shows how we choose which method to call:

```
switch (state)
{
    case SceneState::Edit:
        scene->UpdateEditor(camera);
        break;

    case SceneState::Play:
        scene->UpdatePlay(ts);
        break;
}
```

Code 3. Scene state rendering

If we are in the "Edit" state, our first step is to update the editor camera. By rendering the camera each frame, we give the optical illusion of zooming, moving around and rotating the scene. Once we have updated the camera, we proceed to render the scene itself. To begin, we get the camera view projection and store it in the UBO mentioned earlier. Then, we initialize the necessary parameters to prepare our renderer. We used the concept of "Batch Rendering" to improve the performance. Rather than rendering each scene object individually, which can be inefficient and slow, batch rendering groups similar objects together and renders them as a single batch [80]. After everything is set up, we start by rendering all the entities with a quad shape. These entities have a sprite associated with them which is used to give them a color, a texture or both. If an entity lacks a texture, it just has a color.

For quads without a texture, we generate 4 vertices, and for each vertex, we store the position, the color, and the entity ID. These values are later used in the VBO when rendering the batch. Additionally, we have an indices counter which is used to divide the batches. Each quad needs 6 indices to be drawn (3 for each triangle it is made of). After incrementing the counter, we move on to the next entity.

On the other hand, for quads with a texture, we still follow a similar process. We generate 4 vertices and for each one of them, we store the position, color, texture coordinates, texture ID, texture tile amount and entity ID. Again, we increment the indices counter by 6 and proceed to the next entity.

In both cases, before setting all those values, we check the indices counter. If the amount of indices has exceeded a predetermined maximum number, we render that batch and begin a new one. Otherwise, we continue adding indices to the one currently being used. If the entity has a texture, we also verify if the texture has already been added to an array used for storing texture slots. If it has not been added before, we check if the maximum texture slots count has been reached. If so, we render that batch and start a new one, otherwise, we submit the texture to the array.

After rendering all the quads, we proceed to render all the entities with a circle shape. These entities do not need to have a sprite. For the circle entities, we still generate 4 vertices, as a circle is essentially a quad with disabled fragments, meaning certain fragments within the quad do not contribute to the final color of the corresponding pixels. For each vertex, we store the position, the color, the circle thickness and the entity ID. Similarly, we increment the indices counter by 6 and move on to the next entity. Prior to configuring everything, we once again check the index amount.

Lastly, there may be lines needed to be rendered on the scene. To render the lines, we calculate the position of 2 vertices each time and for each one of them we store the position, color and entity ID. Unlike quads or circles, lines do not need an IBO. Instead, we increment a vertex counter to keep track of the batch. Figure 4, is used to show all the steps described above.

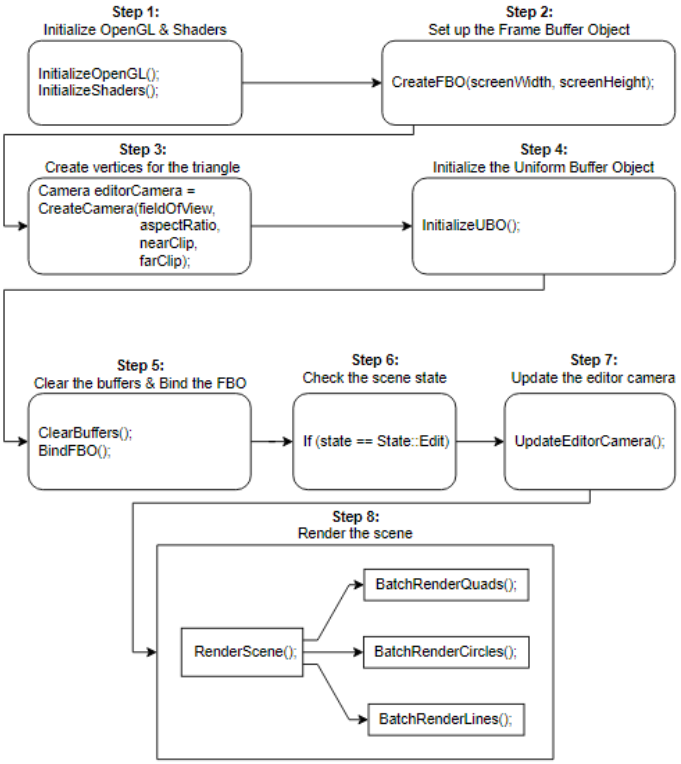

Figure 4. A simplified version of all the steps needed in our engine to render the scene

What does rendering each batch mean? First, we check if there are quads, circles and lines that require rendering. If there are, we retrieve the data we previously set for each vertex. We then initialize and bind a VBO with that data and we also bind the appropriate shader. If there are textures to be rendered, we bind each texture slot stored in the array. Once all the necessary configurations are in place, we bind the VAO and proceed to draw the elements by calling the appropriate OpenGL methods.

The “Play” scene state performs similar rendering tasks, with the key distinction lying in an extra step. We search through the scene to find the entity which is designated as the game camera in order to use it as the main camera. In the “Play” state, we also need to update the scripts attached to each entity and calculate their physics behavior.

4.2.2.1.5 Graphical User Interface Rendering

As we have finished rendering our scene, we move on to render the UI. For the UI, we use “Dear ImGui” [81]. Dear ImGui is an outstanding bloat-free graphical user interface for C++ [81]. This remarkable tool sponsored by technology giants, such as “Ubisoft”, “Blizzard Entertainment”, “Google”, “Nvidia” and more [81], stands out as a preferred choice for UI implementation. After setting the flags and the styles we want for the UI using Dear ImGui, we start rendering each of our panels.

First, we start a new “Gui” frame and we continue by rendering the “Assets” panel. The assets panel is the window which can be used by the end user in order to iterate through the directories and see the files and folders of the project. As we want all of the project files and

folders to be together, using the "std::filesystem" library [82], we made a predetermined "assets" directory. Then, depending on if there is a file or a folder inside the directory, we created and rendered the corresponding texture. Figure 5 is used to show how an assets panel could look like.

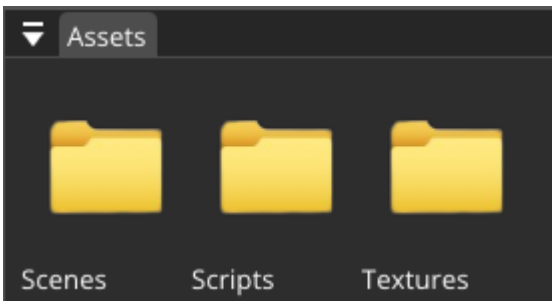

Figure 5. Assets Panel

After making sure the assets panel is fully functional, we continue on rendering the "Hierarchy" and "Properties" panels. The hierarchy panel is used to render all the entities that are currently in the scene. The user can create, delete, select and see the entities of the scene. On the other hand, the properties panel is used to render all the components of a selected entity and their corresponding values. The user can create, delete, select and see the components of a selected entity. It is also used to change the values of some of the parameters of each component. Figures 6 and 7 depict the hierarchy and the properties panels.

Figure 6. Hierarchy Panel

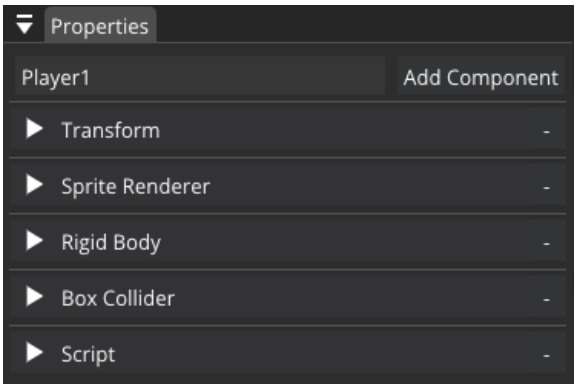

Figure 7. Properties Panel

Then, we render the "Scene" panel which is responsible for displaying our scene. To achieve this, we retrieve the color format that was previously stored in the FBO. This information is essential in order to create a "Gui Image", which serves as the underlying canvas for rendering on top of our scene. Finally, we proceed to render the buttons responsible for changing the state of the scene. These buttons provide functionality for switching between the two scene states. We also utilize "ImGuizmo" by Cedric Guillemet [83] to render gizmos on the scene entities in the "Edit" state. The "ImGuizmo" library is based on the "DearImGui" library and is a powerful tool that enables interactive manipulation and transformation of objects within the scene, offering enhanced editing capabilities [83,84].

We should note that the UI of our game engine has been heavily influenced by other engines such as "Unity", "Unreal Engine", "Hazel", "Cocoa", "Cryengine" and "Erhe". In Figure 8 we depict some of the buttons used for the UI, while in Figure 9 we depict how a gizmo is rendered around a scene entity in order to make it possible to rotate the entity.

Figure 8. UI Buttons

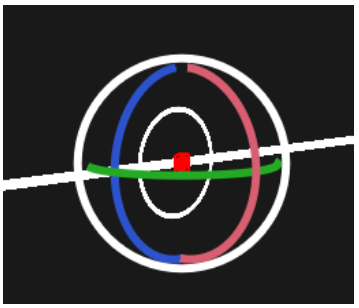

Figure 9. Gizmos rendered around a gameobject of the scene

As we have finished rendering all the main panels, we thought that we should also provide a panel for some core statistics like the frames per second (FPS), the amount of draw calls, the cursor position inside the scene, if VSync is on or not etc. Figure 10 is used to show the Core panel.

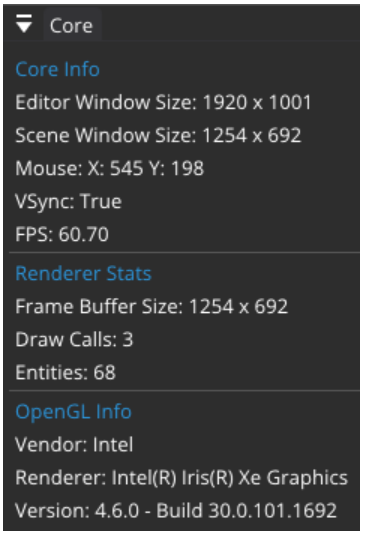

Figure 10. Core panel, used to show some game engine statistics

4.2.2.2 Physics Engine

4.2.2.2.1 Introduction

The physics engine is the subsystem responsible for calculating and handling the physical interactions and behaviors of entities within a game world. It is used to simulate things such as gravity, rigid body dynamics, soft body dynamics, fluid dynamics, collision between entities, realistic movements and more [85]. The physics engine typically has two main components: a collision detection system and a dynamics simulation component responsible for calculating the forces affecting the simulated objects [85-87].

In a physics engine, various algorithms and mathematical models are used to calculate and approximate the behavior of objects based on principles, such as Newton's laws of motion. These calculations determine how objects move, respond to forces, collide with each other, and interact in general with the virtual environment. Overall, a physics engine adds realism and

immersion to games by simulating the physical behavior of objects, allowing for more interactive and dynamic gameplay experiences.

4.2.2.2.2 Box2D

Writing a physics engine for a game engine can offer several benefits such as the ability to fully customize the physics, increased flexibility, easier integration with game logic and most importantly, a deep understanding of the underlying principles. However, it is important to note that developing a physics engine can be a complex and time-consuming task. It requires a strong understanding of physics, mathematics, and programming, as well as expertise in performance optimization. However, there is always a possibility to use a pre-existing library for the physics engine. There are several good choices out there such as “Jolt Physics” [88] which have been used by well-known companies and has also been the main physics engine used by “Guerrilla Games” to create the impressive “Horizon Forbidden West” game. Other great choices are “Bullet Physics SDK” [89], “LiquidFun” [90] which is an open source physics engine for 2D games developed by “Google”, and “Box2D” [91].

After careful consideration, we chose to use Box2D, a widely adopted physics engine used by renowned game engines, frameworks and libraries such as “Unity”, “GameMaker”, “LÖVE”, “libGDX” and many others [91]. The team behind Box2D, provides an exceptional documentation resource [92] that covers everything someone might need in order to both integrate the library in their project and also utilize it correctly. Their documentation not only guides someone through the process of integrating Box2D but also offers a tutorial-style “Hello Box2D” project, serving as a guide to help beginners get started with using Box2D.

4.2.2.2.3 Update Physics

Both of our scene states invoke a method in order to update scripts, physics and render the scene. It should be noted that in the “Edit” state no physics calculations and script updating is done. On the other hand, in the “Play” scene state we calculate the physics and create collision callbacks for the scene entities while we also update the scripts. To change a scene state, the user must press the play button. After the button gets pressed, a function by the name of “PhysicsStart()” gets invoked, which has the role of creating and initializing every single variable needed by both the game engine and Box2D physics engine to start doing physics calculations.

To be more specific, this method starts by initializing a Box2D physics world and specifying the world gravity value which is typically around 9.8 [93]. Then, for all the entities in the scene with a “RigidBody Component” it creates a “Box2D body definition” to store everything needed in order to create a Box2D rigid body, like the body type, entity position, entity ID etc. Using this body definition, it initializes and stores a new Box2D rigid body.

Following the creation of the Box2D rigid body, we continue by checking if the same entity has a collider component. Depending on the collider shape, such as a polygon or a circle, we create the corresponding “b2PolygonShape” or a “b2CircleShape” that will be used as the entity’s collider in the Box2D world. To enable collision detection, we construct a “Box2D Fixture Definition” which contains relevant information for collision handling, such as collider shape, entity’s mass, entity’s friction etc. Finally, we use this fixture definition in order to create a fixture for the Box2D rigid body we made before.

As the Box2D initialization has finished, the “UpdateState()” method mentioned earlier is called during each frame to calculate and update the physics based on the current scene state. As previously explained, the physics calculations occur only when the engine is in the “Play”

state. This scene state invokes a function called “UpdatePhysics()”. First and foremost, inside that function we use the “Step()” method provided by Box2D. This method does all the necessary calculations to handle collision detection, integration and constraint solutions. Next, for each entity with a rigid body component, we first retrieve the newly calculated position and angle of the Box2D rigid body from the “Step()” method. Then, we apply these values to the entity translation and rotation, effectively completing the physics calculations for every frame. Figure 11, depicts the collision between 5 different entities in a game scene.

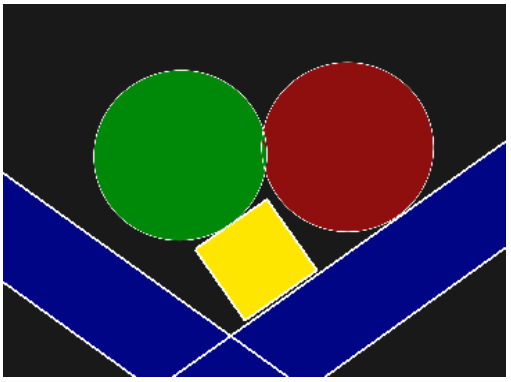

Figure 11. Collision between entities

We should also mention that in the “Play” state, as we also need to update the scripts, we must somehow send the collisions that happened in the scene from the physics engine to the scripting engine. To accomplish this, we utilize the “SetContactListener()” method provided by Box2D. To use this method, we need to create and use a contact listener. Hence, we made a new class named “MyContactListener”, that implements the Box2D “b2ContactListener” class, in order to be able to obtain contact information. Within the “MyContactListener” class, we overridden a function called “BeginContact()” which gets called when two Box2D fixtures begin to touch. The role of this method is to find out which two fixtures were involved in the collision. It then retrieves the entities’ IDs from their rigid bodies and sends them to the scripting engine for further processing.

4.2.2.3 MathEngine

A mathematical engine is an essential subsystem of a game engine. It performs math calculations required for game development. It is responsible for handling complex mathematical operations such as linear algebra, applied mathematics, calculus, numerical methods, trigonometry, and discrete mathematics. The purpose of a math engine in a game engine is to provide the necessary mathematical tools to create game mechanics, physics simulations and graphics rendering [94,95].

Mathematics is an essential part of game development, and game developers need to have a good understanding of mathematical concepts in order to create engaging and immersive games. A math engine provides developers with the necessary tools to perform complex mathematical operations without having to write the code from scratch. Some of the areas where a math engine is used in a game engine include:

- Entity position, orientation, scale and transformation
- Collision detection
- Physics calculations
- Lighting calculations
- Procedural generation

- Animation
- Game engine statistics

Overall, a math engine is an essential component of a game engine that enables game developers to create complex game mechanics, physics simulations, and graphics rendering, contributing to the overall functionality and realism of the game. Even though we plan on creating our own 2D math engine for our game engine in the future, building one from scratch is a time-consuming task. It requires a lot of mathematical knowledge. For example we would need to implement vertex operations, matrix operations, quaternions, functions for rotation, translation, scaling, normalizing vectors, (spherical) linear interpolation, projection, length, clamp and a lot more [7, 95].

As we were not familiar with everything, to free our hands, we decided to use "OpenGL Mathematics" (GLM) [96]. GLM is a C++ mathematics library based on GLSL which we have already used to create our shaders. It provides classes and functions designed and implemented with the same naming conventions and functionality as GLSL and it pretty much includes everything someone might need for their game engine [96]. Another noteworthy choice is MathFu [97]. MathFu is also a C++ open source math library developed by Google primarily for gaming [97].

4.2.2.4 Entity Component System

4.2.2.4.1 Introduction

An Entity Component System (ECS), is a software architectural pattern mostly used in game development for the representation of game world objects [98,99]. The ECS is used for organizing and managing game entities, their behavior, and their data. ECS follows the composition over inheritance principle, which offers better flexibility and helps identify entities, where all objects in a game scene are considered an entity [98].

In the ECS architecture, a game entity is typically represented as an empty container or identifier called an "entity". The entity itself does not contain any behavior or data, instead it serves as a unique identifier for grouping components together. In a game engine, components represent a piece of functionality that can be added to an entity to give it specific behavior or attributes. For example, a game character entity may have components such as transform, velocity, sprite and health. Each component holds specific data related to its property or behavior [98-103].

Components are usually simple and contain only the necessary data and logic for their specific role. They can be used for a wide range of functionality, such as rendering, physics, animation and sound. They are designed to be reusable and modular, allowing game developers to create complex game mechanics by combining different components [98-100][103]. The ECS is responsible for the behavior and functionality of the game. It allows game developers to create shorter and less complicated code and offers a clean design using decoupling, encapsulation, modularization, and reusability methods. Overall, ECS is a powerful tool for game developers that offers a flexible and efficient approach to game development by organizing entities and their components leading to improved code. It provides a clean design and better flexibility when defining objects.

4.2.2.4.2 Entt

Entity Component Systems are crucial for achieving high-performance in game development. They require constant maintenance and the ability to produce reusable code,

making them very challenging to write. Fortunately, there are some free and open source libraries that anyone can use as their ECS. Among these libraries, “entt” [104] stands out as a header-only, lightweight, and user-friendly option written in modern C++. Notably, “entt” has been successfully used in renowned projects like “Minecraft” by Mojang Studios and “Ragdoll Dynamics” [105]. Another notable library is “flecks” [106], which provides a fast and lightweight ECS framework capable of handling millions of entities. It has found practical use in projects such as “Equilibrium Engine”, “The Forge”, and “Territory Control” [106].

For our project, we chose to go with “entt”. Entt uses “registries” which serve as containers for entity IDs and associated data. Also, by utilizing “views” and “groups”, it gives developers the option of searching in their scene for entities with a specific component or multiple components. After creating a new scene, the user has the freedom to create an entity. When an entity gets created, a UUID gets added to it, to differentiate it from other entities. Also, we assign a name to the entity and a “Transform Component” to manipulate the entity’s position, rotation and scale. For each entity, we can check if it exists in the scene, if it has a specific component or a combination of components and we can also add and remove a component. Here is a list of the components currently supported by our engine:

- UUID Component: Used for identifying an entity.
- Name Component: Used for giving an entity a name.
- Transform Component: Used to store the position, rotation and scale of an entity.
- Camera Component: Used to create the illusion of a camera.
- Sprite Renderer Component: Used to render an entity with a sprite, either quad or circle. If the entity does not have a texture, it just has a color.
- Circle Renderer Component: Used to render a circle entity without a texture.
- Rigid Body Component: Used to add a Box2D physics body in an entity. This way we can check the entity body type (static, dynamic, kinematic), restrict rotation, freeze position and enable/disable gravity.
- Box Collider Component: Used to enable collision for quad entities.
- Circle Collider Component: Used to enable collision for circle entities.
- Script Component: Used to add a C# script to an entity.

We should note that the components used in the core of the game engine are not the same ones as the ones used in the core of the scripting engine. Within the scripting engine’s core, we create pseudo components that are used in order to set or get the actual entity component values. Through the use of internal calls, we pass values between the corresponding entity components. Code 4 is an example of how the code of a component looks like.

```cpp
struct RigidBodyComponent
{
    enum class BodyType
    {
        Static = 0,
        Dynamic,
        Kinematic
    };
    BodyType Type = BodyType::Static;

    bool FixedRotation = false;
    bool FixedX = false;
    bool FixedY = false;

    bool UseGravity = false;

    RigidBodyComponent() = default;
};
```

Code 4. Example of a component

We should note that the components used in the core of the game engine are not the same ones as the ones used in the core of the scripting engine. Within the scripting engine's core, we create pseudo components that are used in order to set or get the actual entity component values. Through the use of internal calls, we pass values between the corresponding entity components. Figure 12 shows an example of how the properties panel renders the components of an entity.

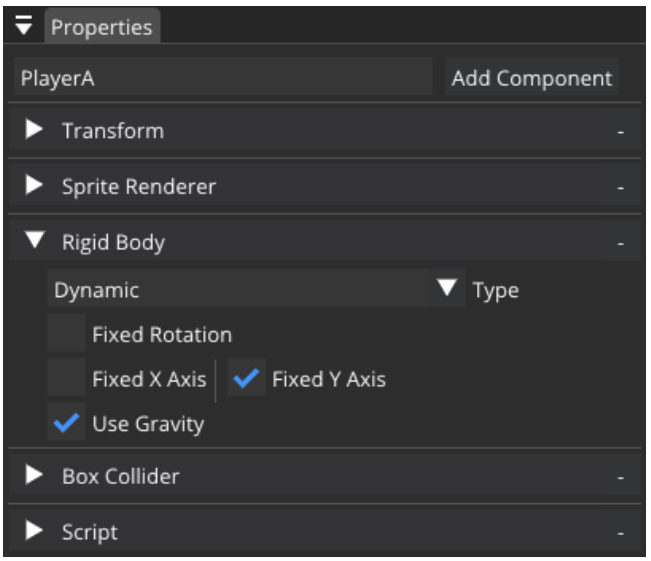

Figure 12. Visualization of components

4.2.2.5 Frame Timing Control

Another important subsystem is the frame timing control subsystem. It is used to establish a definite order for events, physics and script updates and rendering each frame. This way we have a coherent and predictable sequence of events within the engine. This subsystem provides a structured framework for organizing and managing the updates order and rendering process by separating different areas of the game engine into layers. Whenever a new layer gets created, it gets pushed at the back of an array which is used to store them. Following that, a method is invoked to initialize that layer. This method plays a crucial role in initializing and

configuring the layer with the necessary parameters to ensure its functionality. On the other hand, there is another method that gets called either when the layer destructor is invoked or when we intend to remove a layer from the array.

The “Update()” and “UIRender()” methods, used in the main game loop, get called for all the game engine layers, one by one. In the “Update()” method, the engine updates the physics, scripts, camera and other entities in general. This includes performing necessary calculations, applying changes, and preparing the scene for rendering. Once the updates are done, it proceeds to render the scene. On the other hand, the “UIRender()” method is responsible for rendering all the UI panels. Regarding the event handling, the layers are processed in reverse order, starting from the highest priority layer, moving towards the lowest priority one. This approach allows for proper event propagation and ensures that higher priority layers have precedence in responding to events. Additionally, we have implemented a simple overlay system that complements the layer system. The layer system consists of a contiguous list of layers, whereas the overlays are always positioned at the end of this list. Overlays must be rendered last to ensure they are displayed on top of other graphical elements and get prioritized in event handling.

4.2.2.6 Event Subsystem

An event subsystem in a game engine is crucial. It enables communication and interaction between all the other engine subsystems. It helps to decouple multiple subsystems and makes it possible to compile them independently. The event subsystem operates on the concept of events, which are messages or signals representing specific occurrences or actions within the engine. These events can include user input, window size change, game state changes and more. When an event occurs, it is dispatched by the sender object and propagated through the event subsystem [107-110].

The event subsystem acts as a dispatcher, responsible for routing events to the appropriate event handlers. It processes events that are triggered and places them into a queue. The events are only called when polled. Subsystems interested in specific events register themselves as listeners or subscribers to those events. When an event is dispatched, the event subsystem notifies all registered listeners, allowing them to respond accordingly [107-109]. Additionally, the event subsystem can support event prioritization, allowing certain events to take precedence over others or certain subsystems to handle events before others. Overall, the event subsystem is an efficient way for engine subsystems to communicate with each other. It helps to decouple many subsystems providing flexibility and communication and interaction between the subsystems [108,109].

Our game engine uses an event subsystem to check for window, keyboard, and mouse events. We set “GLFW” event callbacks for each of the events our engine supports and we chose to separate the events into types e.g. WindowResize, MouseMoved, KeyPressed etc. Each frame the engine polls the events using the “glfwPollEvents()” method given by “GLFW”. This way all pending events get processed. Using an event dispatcher for every layer and overlay in the engine, comparisons between event types happen and if a match is found, the dispatcher directs the event to the appropriate method for handling.

If someone prefers not to use “GLFW” and wants to avoid writing their own dispatcher, event types etc., there is of course the option of using an existing library. A good choice would be “libevent” [111]. Libevent is an event notification library with approximately 10,000 stars on GitHub and it can be compiled on Windows, Mac and Linux. Notably, Libevent is used by “Tor” [112] and “Chromium” [113] among many others [111]. Code 5 provides a visual representation of the code written to create an event when the mouse moves.

```
MouseMoved(const float x, const float y)
    : m_MouseX(x), m_MouseY(y)
{
}

inline float GetX() const { return m_MouseX; }
inline float GetY() const { return m_MouseY; }

EVENT_TYPE(MouseMoved)
```

Code 5. Mouse Moved Event

4.2.2.7 Input Subsystem

The Input subsystem is responsible for handling and processing user input. It allows the engine to receive input from various sources, such as the keyboard, the mouse, a controller, a touchscreen, or other input devices, and translates those input data into actions. It translates low-level input into high-level logical events that the game logic can understand [114-116]. The input subsystem typically works by monitoring the state of input devices and notifying the game engine or relevant game objects about any changes or events. For example, when a user presses a key on the keyboard, the input subsystem detects the key press event and sends that information to the game engine. The game engine can then interpret the input and trigger appropriate actions, such as creating a new scene, saving the current scene, interacting with objects in the scene game or other events like moving a player in a game or shooting a gun.

In our game engine, we have implemented an input subsystem that utilizes a polling mechanism alongside event notifications. Instead of waiting to be notified when an input event occurs, this subsystem allows us to actively inquire about the current state of specific inputs. For example, we can check whether a particular key, such as the "K" key, is currently pressed, or if a mouse button is being held down. This polling-based approach provides flexibility and allows us to incorporate complex input behaviors. For instance, we may want to move the camera around by moving the mouse while also holding the "Alt" key down or press both the "Left Control" and "S" keys together to save the current state of the scene. By regularly polling the state of the keys and mouse buttons, we can determine if both are held down during a mouse movement event or a key pressed event and perform the desired action. For our Input subsystem, we chose to use the exact same IDs for the keys and mouse buttons as the ones used by the "GLFW" library, for compatibility with OpenGL inputs [117]. Code 6 shows an example of a key pressed event.

```
bool Input::IsKeyPressed(const Key key)
{
    // Get app window
    GLFWwindow* window = static_cast<GLFWwindow*>(application.GetWindow());

    // Get key state through GLFW
    auto state = glfwGetKey(window, key);

    return state == GLFW_PRESS;
}
```

Code 6. Example of a key event

4.2.2.8 Scripting Engine

4.2.2.8.1 Introduction

Game developers use the scripting engine to control game aspects via written code. The scripting engine provides the ability to modify the game logic and behavior without the need of modifying or recompiling the core game engine code [118,121,122]. It is integrated into game engines for languages used for game development like C#, Lua, GameMonkey, Python etc. These

languages give developers the freedom to create script game systems and interact with game assets and game entities [118,119,121,122]. The scripting engine of each game engine is different, as it depends on the game needs and developer preferences. These differences include the choice of programming language for game scripting (e.g. Lua, C# etc), the range of features it may support (such as Built-In methods and keywords) and even the type of scripting methods available, such as traditional coding or visual scripting.

Through the implementation of scripts, developers can easily define game object behavior, like character movement, enemy interactions, world creation, item utilization, animations, collisions and a lot more. Additionally, scripts are also used to handle in-game events, including keyboard, controller and mouse inputs, timers, collisions, and system events. Furthermore, they can be used to control the game interface allowing developers to define UI elements, update UI states, and manage UI animations. Moreover, scripts have also found their place in numerous other areas such as level design, particle effect creation, sound management, support for custom assets and many more [118-122].

In recent years, game engines have introduced a revolutionary feature known as Visual Scripting, which makes the implementation of game logic and entity behavior a lot simpler. Using a visual interface, developers can now create scripts for game entities and environments without the need of writing code in programming languages. Instead of writing lines of code, users can drag and connect nodes and blocks that represent game elements and actions. These nodes typically represent functions, conditions, variables, and events that can be combined to define the desired behavior of the game [119][123-126].

4.2.2.8.2 Mono - C#

The programming scripting language we chose to support for our game engine was C#. C# is a general-purpose high-level language which supports static typing, object orientation and offers advanced capabilities and fast performance [127]. The C# programming language provides features that align with our scripting engine needs for desired functionality and flexibility, as it offers tools and resources for both further developing the scripting engine and integrating it smoothly with the game engine.

To include C# in our game engine and make it possible to communicate between C++ and C#, we had two choices. Either use the ".Net Core" version provided by Microsoft or use the "Mono" project sponsored by Microsoft. The ".Net" is a free, cross-platform, open-source developer platform for building many kinds of applications and it is built on a high-performance runtime that is used in production by many high-scale apps [128,129]. The GitHub repository has around 20.000 stars and is actively maintained [130]. On the other hand, the "Mono" project is an open source implementation of Microsoft's ".NET Framework" as part of the ".NET Foundation" and based on the ECMA standards for C# and the Common Language Runtime [131]. Its aim is to become the leading choice for development of cross platform applications [131,132]. The GitHub repository has more than 10.000 stars and more than 122.000 commits and of course it is actively maintained [133].

So, for our scripting engine and to basically make it possible to integrate C# in the engine, we chose to go with Mono. The main reasons for our choice were that Unity uses Mono, it is also designed to be cross-platform and the most important part is that it supports assembly reloading. Assembly reloading, also known as hot reloading, is the process of updating and reloading code during runtime without the need of restarting the application or game. It makes it possible for developers to see the effects of the changes they made to the scripts immediately, without having to go through the entire compilation and deployment process [134,135]. To

download and integrate Mono efficiently in our engine we followed the guide provided by Nilsson [136] and the ones provided by the Mono team in their documentation [137,138].

4.2.2.8.3 Scripting Engine Implementation

Whenever we start the application, one of the steps we do is initialize the scripting engine. First and foremost, we must create and set some paths and variables for Mono. After we have finished creating the Mono root and app domains, we continue with loading the C# assembly. As we have loaded the assembly, we can now read it and find all the C# classes the user may have created and all the parameters of each one of them. Then, for each C# class we create a Mono class and store it with its corresponding fields. We have now successfully read the C# assembly and the classes inside it. Our next step involves finding and getting the "Core" class from the C# assembly, which will be used as the base class from which all the game scripts will be inheriting the scripting engine methods. It is the same thing as the "MonoBehaviour" class used by Unity or the "AActor" class used by Unreal Engine.

Finally, we have to set up all the functions and C++ components which will be used internally to allow communication between C++ and C#. To create and utilize functions by both the scripting engine and the core game engine, we had to create "Internal Calls", as it is requested by mono for each one of them. Internal calls act as a bridge that allows C# scripts to access and invoke particular methods present in the C++ code. Code 7 shows an example of a method which is processed in the C++ part of the scripting engine, even though it got called from a C# script, while Code 8 shows how to create an internal call in C#.

```
static bool RigidBodyComponent_IsAwake(UUID entityID)
{
    Scene* scene = ScriptingEngine::GetScene();
    if (scene != nullptr)
    {
        Entity entity = scene->GetEntityByID(entityID);

        if (entity.GetID() != -1)
        {
            b2Body* body = (b2Body*)entity.GetComponent<RigidBodyComponent>().B2Body;
            return body->IsAwake();
        }
        return 0;
    }
    return 0;
}
```

Code 7. C# & C++ method

```
[MethodImplAttribute(MethodImplOptions.InternalCall)]
extern static string Sample ();
```

Code 8. C# internal call. Code by: Embedding Mono

Then, for each entity of the scene with a script component, we check in the core scripting engine which other components it has and store them. This makes it possible to inquire about the components an entity may have, in the C# scripts. In more details, we retrieve the component's name from the script, we convert the C# component into a mono type component, and through a map we compare the C# component with the ones in C++. We should note that the C# components do not contain the real entity fields like the C++ components do. They only contain internal functions and fields which are mapped to the correct C++ functions and fields and they are used to retrieve or send entity data. Code 9 shows an example of how we get a C++ component in C#, and how we use that component to invoke a C++ method:

```
m_RigidBody = GetComponent<RigidBodyComponent>();
bool isEntityAwake = m_RigidBody.IsAwake();
```

Code 9. C# & C++ interaction

As we have finished with the initialization of the scripting engine, we will now proceed to see how it works in general. First and foremost, whenever the scene starts playing, for each entity with a script component we need to call the C# constructor to initialize it and then search through the class to find and invoke the corresponding "OnCreate()" method (if there is one). Code 10 shows a script which utilizes the "OnCreate()" function:

```
public class FootballPlayer : Core
{
    public float Speed;
    public float JumpForce;
    public float Shoot;

    private RigidBodyComponent rigidBody;
    private bool isOnFloor = false;

    private Core floor;
    private Core ball;

    // Gets called before updating the first frame
    void OnCreate()
    {
        rigidBody = GetComponent<RigidBodyComponent>();

        floor = FindObjectByName("Floor");
        ball  = FindObjectByName("Ball");
    }
```

Code 10. C# OnCreate() method

Then, every frame we need to update the scripts in order to change the behavior of the scene entities. This means that for each entity with a script component we need to search through the class to find and invoke the corresponding "OnUpdate()" method (if there is one). This method is invoked every frame and its primary function is to update the fields and, more broadly, the data associated with an entity. Code 11 shows a script which utilizes the "OnUpdate()" method:

```
// Gets called every frame
void OnUpdate(float ts)
{
    Vector3 velocity = Vector3.Zero;

    if (Input.IsKeyDown(KeyCode.A))
        velocity.X = -5;
    else if (Input.IsKeyDown(KeyCode.D))
        velocity.X =  5;

    velocity *= Speed * ts;
    rigidBody.ApplyLinearImpulse(velocity.XY, true);

    if (Input.IsKeyDown(KeyCode.W) && isOnFloor)
    {
        rigidBody.ApplyForce(new Vector2(0, JumpForce));
        isOnFloor = false;
    }
}
```

Code 11. C# OnUpdate() method

Finally, using the "MyContactListener" class, we retrieve from the physics engine the collisions that happened and we process them in the scripting engine. For each entity with a script component, we again search through the class to find and invoke the corresponding "OnCollisionBegin()" or "OnCollisionEnd()" method (if there is one). These methods get called

when a collision starts or ends and get as input the two entities that collided. Code 12 shows a script which utilizes the “OnCollisionBegin()” method:

```
// Gets called when two entities start colliding
void OnCollisionBegin(ulong ent1ID, ulong ent2ID)
{
    if (ID == ent1ID && ball.ID == ent2ID || ID == ent2ID && ball.ID == ent1ID)
    {
        Vector2 kickDirection = Vector2.Zero;

        if (Input.IsKeyDown(KeyCode.A))
            kickDirection += Vector2.Left;
        if (Input.IsKeyDown(KeyCode.D))
            kickDirection += Vector2.Right;

        if (Input.IsKeyDown(KeyCode.Space))
            ball.GetComponent<RigidBodyComponent>().ApplyForce(kickDirection * Shoot);
    }

    if (ID == ent1ID && floor.ID == ent2ID || ID == ent2ID && floor.ID == ent1ID)
        isOnFloor = true;
}
```

Code 12. C# OnCollisionBegin() method

In conclusion, a scripting engine within a game engine is a valuable and helpful subsystem for developers. Even though it is not really necessary, it makes the implementation of game logic easier and it also makes it possible to customize gameplay mechanics, and dynamically create content. It promotes flexibility and with the hot reloading feature it makes the overall user experience a lot better.

4.2.2.9 Assets Management System

To improve the management of our files, scenes, and assets, we recognized the necessity to develop an “Assets” panel. The primary objective of this window panel is to organize all the project assets more effectively. With this addition, we aim to optimize the user experience and make it possible to navigate and access project folders and files within the engine. To achieve this feature, we used the “std::filesystem” library [82] in order to create a predetermined “assets” directory. Then, depending on the asset, we show the corresponding texture.

With the help of the “Dear ImGui” library, we managed to implement a drag and drop functionality. This makes it possible for the user to open scenes by dropping them in the scene panel or add textures and scripts to entities by dropping them on top of the scene game object. Currently our engine supports textures with the following extensions: “.png”, “.jpg”, “.jpeg”, “.raw”, “.webp”, “.svg”. Another feature the assets management subsystem provides is creating, saving and opening scenes through the traditional way, which is by navigating through folders and files. Figure 13 and 14 are used to depict the navigation method.

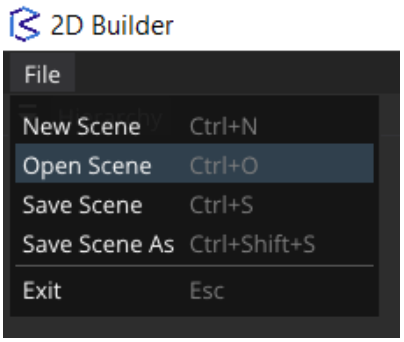

Figure 13. Open a scene through the assets management subsystem

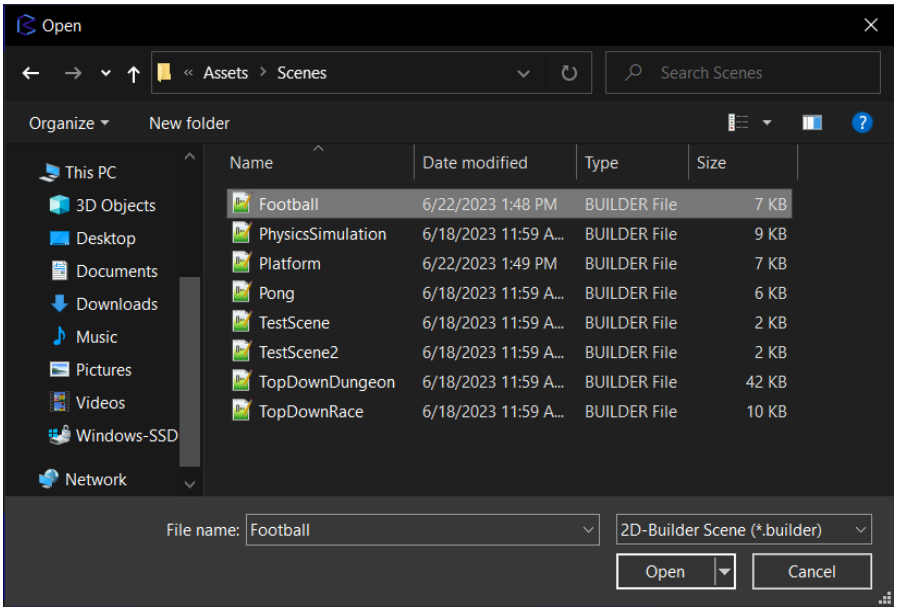

Figure 14. Choose a Scene to open in the editor

4.2.2.10 Serialization

The objective of this subsystem is to facilitate the serialization and deserialization of scenes and projects within a game engine. This functionality makes it possible to store and retrieve scenes, other assets and entities into a format that can be easily saved, loaded and edited. The approach we chose to go with to serialize the scenes within our game engine involves utilizing a text format scene serialization, which creates human-readable text files. The choice to avoid binary serialization in this scenario is primarily driven by the desire for human readability and the ease of both editing the files and merging any changes made to them.

For the serialization language, we chose to go with “YAML” [139]. YAML is a human-readable data-serialization language which is preferred due to its inherent readability and superior merging capabilities. YAML offers a straightforward and minimal syntax, utilizing key-value pairs and arrays. In contrast, “JSON” [140], while widely used, is a lot more challenging to read and merge due to its reliance on curly brackets and potential formatting issues. On the other hand, a great choice is to use an existing library instead of a serialization language like “Cereal” [141]. Cereal is a header-only “C++11” library created for serialization [141]. In the end, we chose to go with YAML as our text file format language, as it is very popular within the game engine industry, including its use by the Unity game engine [40].

Whenever we try to save or open a scene, we must serialize or deserialize the entities inside of it. Let’s start with the serialization process. When the user chooses to save a scene, the engine goes through all the entities in the scene and stores them in a file. This process gets the data of each entity and its corresponding components, converts them into human readable values and stores them inside a YAML map structure. As all the scene entities and their components have been serialized, using a YAML emitter they get outputted into the scene file. This makes it possible to store scenes and save the project progress, as the serialized YAML file can be later deserialized, making it possible to continue working from the saved state. In Code 13 is an example of how a component of an entity gets serialized.

```cpp
// Transform Component
if (entity.HasComponent<TransformComponent>())
{
    out << YAML::Key << "TransformComponent";
    out << YAML::BeginMap; // Transform Component Map

    TransformComponent& tc = entity.GetComponent<TransformComponent>();
    out << YAML::Key << "Translation" << YAML::Value << tc.Translation;
    out << YAML::Key << "Rotation"    << YAML::Value << tc.Rotation;
    out << YAML::Key << "Scale"       << YAML::Value << tc.Scale;

    out << YAML::EndMap; // End Transform Component Map
}
```

Code 13. Entity component serialization

On the other hand, when the user chooses to open a scene, the engine performs scene deserialization. First of all, we try to load the YAML scene file. If everything goes correctly, we start iterating through the data to find and deserialize all the entities of the scene. To deserialize the entities, for each one of them a new entity gets created and depending on the components stored in the YAML file, they get added to the entity with their corresponding values. After we have created all the new entities, we add them to the scene, so that they become part of the opened scene.

4.2.3 Implementation

As we had limited experience in the area of game engine development, we thought that it would be best if we followed a more hybrid approach. We intentionally pursued knowledge across all subsystems and since we did not have enough time and knowledge to create all the subsystems that we needed from scratch we both implemented some of them and integrated existing ones for others. In Figure 15, we depict with a green color all the subsystems we have included so far in our engine, with a yellow color the ones we aim to redesign in the near future in order to add more functionality and finally with a red color the ones we plan to integrate. Moreover, underneath each of the subsystems we thought that it would be a great idea to mention some of the technologies that can be used to implement or integrate each of the subsystems.

Even though all these subsystems collectively contribute to the game engine, there is no need to implement all of them to achieve a fully functional game engine. The build subsystem plays a crucial role in simplifying the integration of essential APIs and libraries required for most subsystems. Moreover, without the event subsystem it would have been very difficult to connect the various subsystems and effectively dispatch events to them. Certain subsystems share a very close relationship. For instance, the math engine serves as a fundamental component for nearly every other subsystem. The frame timing control subsystem determines when the physics, rendering, scripting engines, scene management, assets management subsystems, user interface, and the entity component system come into play. Another close relationship exists between the serialization and the scene management subsystems, as there is the need to serialize and deserialize each scene. Similarly, a close association is observed between the input and the event subsystems, as every input triggers a notification to the event dispatcher in order to take the appropriate action.

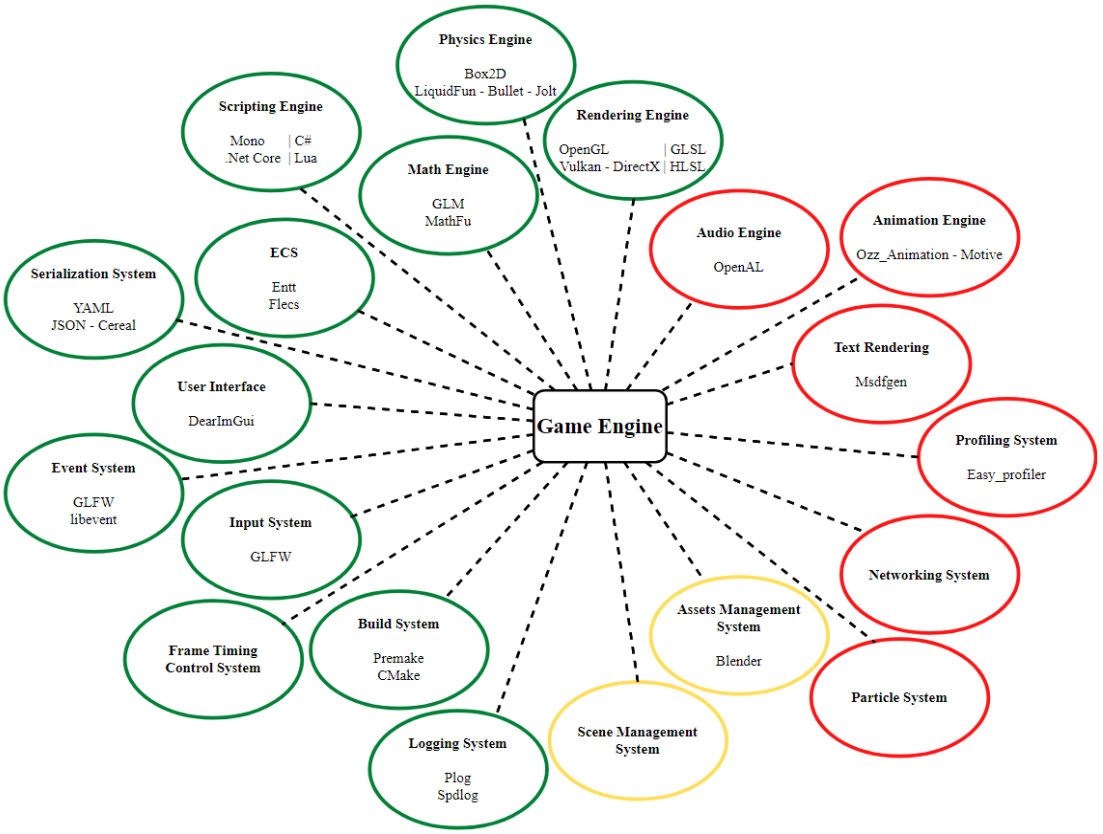

Figure 15. Our game engine subsystems.

After having finished implementing the first functional version of our engine, we calculated some statistics. The engine consists of 65 classes, while there are 12.768 lines of code written. There are also around 475.000 lines of code integrated in our project, by the use of APIs and libraries. Figure 16 shows a graph used in order to depict the way the 12.768 lines of code have been distributed. Within the breakdown, the "Core Engine" refers to the code dedicated to creating engine subsystems, connecting them and making sure the whole engine is functional. We exclude pre-existing API code used for most of the subsystems, and the code written to develop the scripting engine. The "Scripting Engine" refers to the code responsible for developing the scripting engine, establishing communication with the Core Engine and creating build-in classes, methods, and other features. The "C# Scripts" refers to all the scripts written in C#, in order to update the behavior of the entities during gameplay. Finally, the "Scenes - YAML" refers to the code that is automatically generated in YAML format whenever a scene is created or modified.

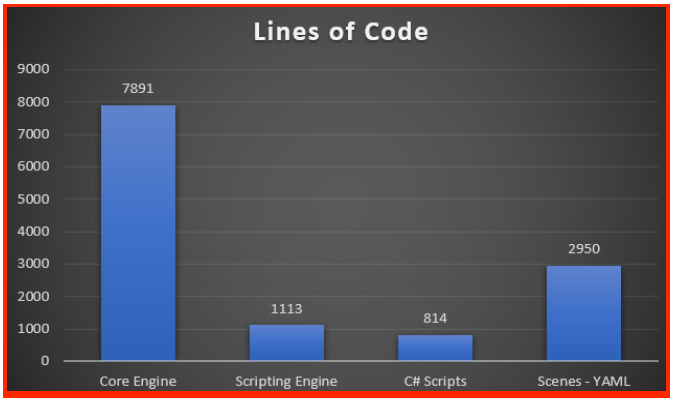

Figure 16. The way the lines of code written for the engine have been separated

5 Classes

In Table 3, we present a table showcasing all the classes used till now in order to develop our game engine. The left column contains the class names, while the right column provides a brief explanation of the class core function and purpose.

Table 3: Classes

Class Name	Brief Explanation
Application	The main project application. Initializes all the required systems and variables at the start of the project, processes the command line arguments and starts the main game loop.
Assets	This class is used to represent the Assets panel of the Assets Management System.
BodyType	Used to store the body type of an entity with a rigid body component.
Camera	It is used to create a camera and store its projection. Gets implemented by the EditorCamera and SceneCamera classes.
Component	The component part of the Entity Component System. Used to hold each of the supported components.
Components	A C# class to represent the C# entity components.
Core	A C# class used as the base class from which all the game scripts will be inheriting the scripting engine methods.
Editor	The main project editor. It is used to create the editor layer of the game engine.
EditorCamera	It is used to initialize and store all the necessary values for the editor camera. It is also used to render the camera each frame and handle events related to it.
EditorLayer	A layer of the frame timing control system. It is used to update the editor window, camera and scene for each frame. Moreover, it is used to render the scene and the UI panels. Finally, it provides the functionality of opening and creating a project, opening, creating and saving a scene, playing and stopping a scene and it also handles all the events related to the game engine editor.
Entity	The entity part of the Entity Component System. Used to create entities and store their corresponding data. For example give them names, add components, remove components etc.
Event	Used to store events and iterate through them.
EventDispatcher	Used to dispatch events for correct event handling.
EventTypes	Used to store the types of the events.

Files	Used to open, save and read a file.
FrameBuffer	Used by the OpenGL API for rendering. Its purpose is to store and choose the color format and depth format of each texture.
GameCamera	It is used to initialize and store all the necessary values for the camera the scene uses when the play button gets clicked. It also handles rendering the camera each frame while the scene is being played.
Hierarchy	This class is used to represent the Hierarchy panel. The hierarchy is used to render all the entities of the scene.
ImGuiFont	Used by DearImGui to render the font of the panels.
ImGuiLayer	A layer of the frame timing control system. Used to initialize the ImGui library, render the ImGui user interface (UI) and process all the UI events.
IndexBuffer	Used by the OpenGL API for rendering. Its purpose is to create a buffer which contains the necessary indices and their corresponding data to render an entity.
Input	Used to process keyboard and mouse input events.
InternalCalls	A C# class to store all the internal calls to make it possible to communicate between the core game engine and the scripting engine.
Key	Used to store and return the code of each key.
KeyPressed	Checks if a key has been pressed.
KeyReleased	Checks if a key has been released.
KeyTyped	Checks if a key has been typed.
Layer	Used for the frame timing control system. Its purpose is to establish a definite order for the events, physics update, scripts update and rendering each frame. It also supports a list to store all the layers, keep track of them and iterate through them with ease.
Log	Used to create our own logging macros instead of using “std::cout” or “std::printf”.
MouseButton	Used to store and return the code of each mouse button.
MouseButtonPressed	Checks if a mouse button has been pressed.
MouseButtonReleased	Checks if a mouse button has been released.
MouseMoved	Checks if the mouse has been moved.
MouseScrolled	Checks if the mouse has been scrolled.

MyContactListener	Implements the “b2ContactListener”. It is used to detect collisions between scene entities, stores the IDs of entities that collided and sends them to the scripting engine for further processing. It supports functions for when two fixtures begin to touch and cease to touch.
Project	It is used to create different projects inside the game engine.
ProjectSerialization	It is used to serialize and deserialize the game engine projects.
Renderer	The renderer class. Calls the appropriate renderer initialization and shutdown methods.
Renderer2D	This class makes rendering possible. It initializes the Vertex Arrays, Vertex Buffers and Index buffers. It also initializes the Uniform Buffer, Textures and Shaders. Its main purpose is to start a new scene every frame, collect all the necessary data, and start rendering the scene entities in batches.
RendererOpenGL	Used to initialize all the OpenGL API necessary methods and variables.
Scene	The class that represents the game scene. It is used to create different scenes. It supports creating, deleting, duplicating and finding entities. But its main purpose is to update physics and scripts and render the correct scene depending on its state.
SceneSerialization	It is used to serialize and deserialize a scene.
ScriptClass	Used to store the C# classes as Mono classes. Supports invoking and retrieving functions belonging to the C# classes through C++ such as “OnCreate()” and “OnUpdate()”.
ScriptConnector	Connects the scripting engine with the game engine core. It allows communication between C# scripts and the C++ core.
ScriptingEngine	It is used for the scripting engine initialization. Furthermore, it is used to shutdown the engine, load and unload the C# assembly. It also provides the ability to add scripts to entities and makes it possible to change their behavior through them.
Settings	This class is used to initialize essential settings for the core engine.
Shader	Used by the OpenGL API for rendering. Its purpose is to read GLSL shader files, create a shader program and upload uniforms to the correct shader.
Texture	Used by the OpenGL API for rendering. Its purpose is to create and store textures from file paths. It also makes it possible to add textures to UI buttons, folders, files and to entities.

Timer	It is used in order to create a timer. It is also used as the timestep, which is needed for the updates. It supports returning the elapsed time and resetting the timer.
UniformBuffer	Used by the OpenGL API for rendering. Its purpose is to create a container which holds the necessary data related to camera projection and view matrices for a shader program.
UUID	It is used to create a universal unique identification (uuid) for each entity in the scene.
VertexArray	Used by the OpenGL API for rendering. It supports creating a vertex array object, binding and unbinding it and linking vertex buffer objects and index buffer objects to it.
VertexBuffer	Used by the OpenGL API for rendering. Its purpose is to create a buffer which contains the necessary vertices and their corresponding data to render an entity.
Window	The main application window. Also used to store the main GLFW window and set the GLFW event callbacks.
WindowClose	It is used to check if the window has been closed.
WindowResize	It is used to check if the window has been resized.

6 Our Projects

6.1 Introduction

In this section, we thought that we should showcase and give a small explanation of some of the games we have created using our game engine. We also thought that it would be a good idea to start by showing the process of developing a game, focusing specifically on the creation of a basic 2D Top-Down Dungeon game. With this guide we aim to cover everything about making a simple 2D game using our engine and show how a straightforward yet highly entertaining game can be developed with remarkable ease and efficiency. Figure 17 shows what the end game will look like.

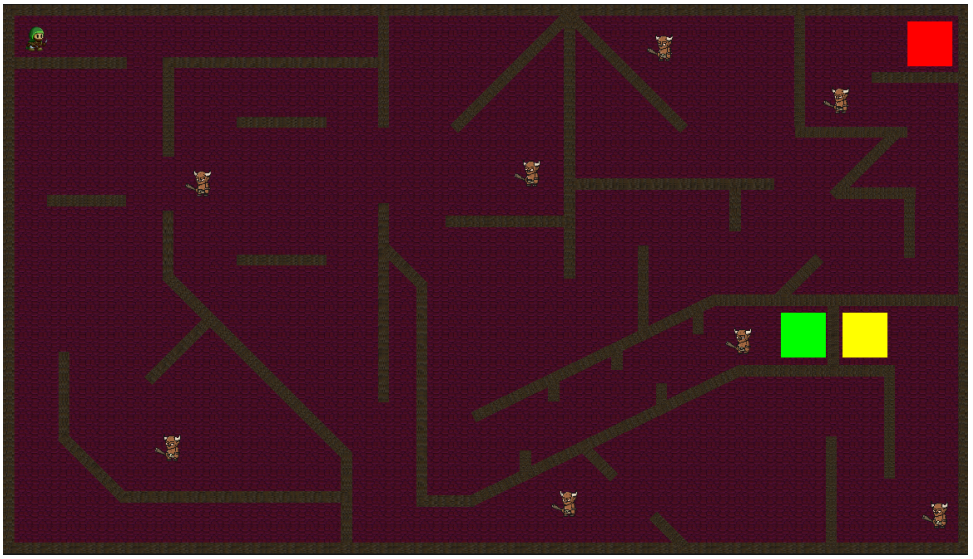

Figure 17. 2D Top-Down dungeon game

6.2 2D Top-Down Dungeon

The objective of the game is to stand on top of all three colorful rectangles while evading potential threats from enemies. To win the game and survive, the player needs to navigate the game environment skillfully and avoid getting attacked by more than five enemies. Failure to do so will result in the player's defeat and subsequent loss of the game.

6.2.1 Background

Immersive backgrounds are integral components of 2D games, including our top-down dungeon game. In this section, we will discuss the simple process involved in creating a captivating background for a game. To create the game floor, which serves as both the game background and the platform on which the player and enemies will walk, we begin by creating a game entity. We adjust the entity's 'X' and 'Y' scale, and modify the 'Z' translation value to make sure it renders behind other game objects. Figure 18 shows the entity creation, Figure 19 shows how to add a component to an entity, and Figure 20 the entity's transform values.

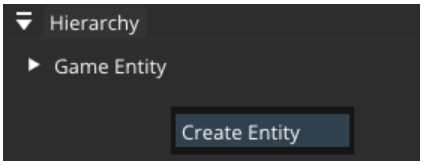

Figure 18. Create a game entity

Figure 19. Add component to an entity

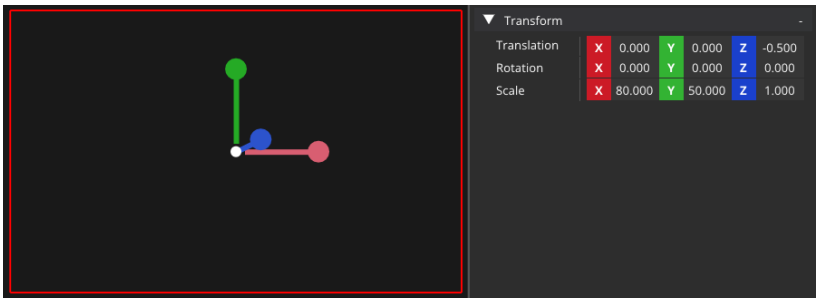

Figure 20. Background entity transform

As we have created our background, the next step is to add a texture to it to make it look like a floor. To achieve this, we need to add a sprite renderer component to the entity. After adding the component, we drag and drop a texture from the assets panel on top of the entity and we give the texture a red-purple color to make the floor more dungeon realistic. The final step is to increase the tile amount to a number that will make the background image more realistic. Figure 21 shows the sprite renderer component of the background entity.

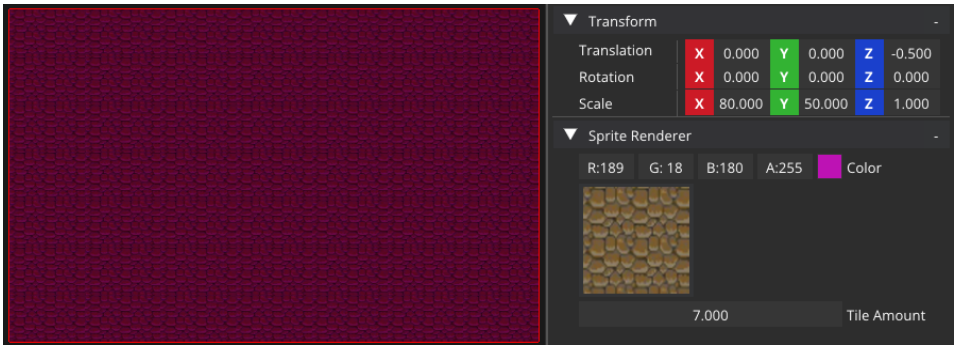

Figure 21. Background entity texture

6.2.2 Walls

The game world has multiple walls. There are 4 main exterior walls, which serve as a boundary and encapsulate the game's environment and then a lot more inside the game world to confuse the player. For each wall of the game, we need to create a new entity in the hierarchy and modify it according to our needs so it will look like Figure 22.

Figure 22. Game walls

After adjusting the entity transform we need to add the essential components with the "Add Component" button. Those include:

- Sprite Renderer: To add the wall texture and the texture tile amount to the entity.
- RigidBody: To give the wall a physical body and make it possible to react to real-time physics.
  - Type: Static, so the entity will stay still at all times.
  - Fixed Rotation: Unchecked, the entity is static so it will not rotate.
  - Fixed X/Y Axis: Unchecked, the entity is static so it will not move.
  - Use Gravity: Unchecked, the entity should not react to gravity forces.
- Box Collider: To give the wall a collider and make it possible to collide with the player else the player would just pass through it.
  - Offset: Modify the position of the collider relative to the entity.
  - Size: Modify the entity's collider size.
  - Density: Set the mass of a body. Higher density means a heavier body.
  - Friction: Determines the amount of resistance to motion between colliding objects. Higher value means higher friction.
  - Bounciness: How much a body will bounce after a collision.

  Figure 23 shows the final components of the game walls.

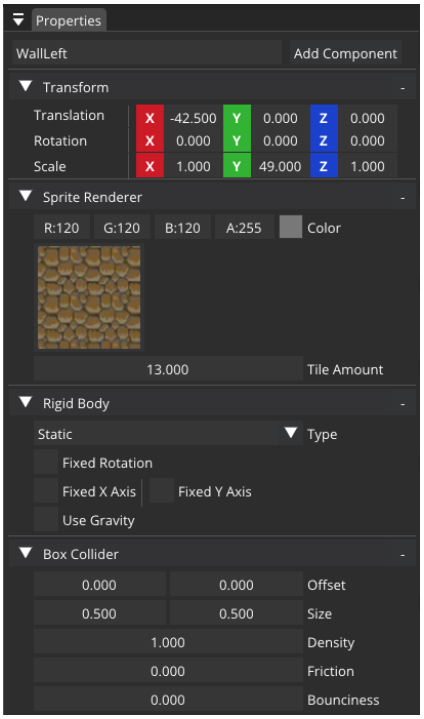

Figure 23. Game wall components

Subsequently, we follow the same steps to create and position the remaining outer and inner walls of the game world.

6.2.3 Camera

To make it possible to play the game and visualize our progress, we must implement a game camera. We chose to create 2 different cameras and give the user the ability to choose which one they want to use while playing the game. The first camera is an orthographic camera. It is used in order to capture the entirety of the game scene as shown in Figure 17, which allows a comprehensive view of the game world. Figure 24 is used to show the orthographic camera components.

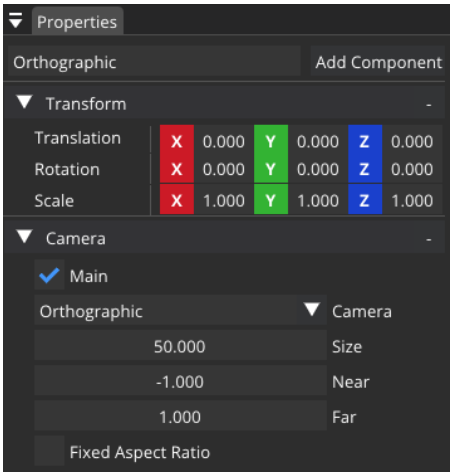

Figure 24. Orthographic camera components

On the other hand, we have implemented a perspective camera for our second game camera. This camera is locked on to the player and moves when the player moves. It is used in

order to make the game more difficult, as the user now will not be able to see the whole game map. Figure 25 shows the perspective camera components.

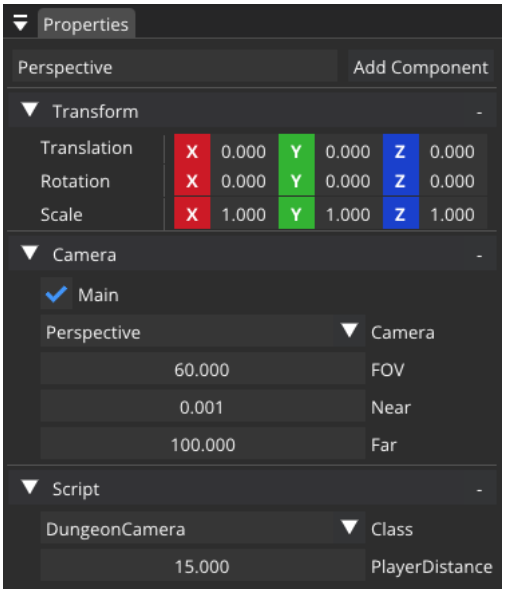

Figure 25. Perspective camera components

For the camera component, we have the following values:

- Main: Checked, so the game will use this camera while being played.
- Camera: A menu to decide the camera type.
- Orthographic Camera:
  - Size: Size of the camera.
  - Near: The distance of the near clipping plane from the camera. The camera cannot see entities closer than this distance.
  - Far: The distance of the far clipping plane from the camera. The camera cannot see entities further than this distance.
- Perspective Camera:
  - FOV: The extent of the observable world that is seen at any given time.
  - Near: It determines how close the camera is to the scene.
  - Far: It determines how further the camera is from the scene.
- Fixed Aspect Ratio: Checked, so there is a consistent width-to-height ratio for the camera's viewport.

As it can be seen in the perspective camera components, to make it possible for the camera to lock on the player and move with him, we had to write a script. We added a script component, created a C# script and then searched through the list to find our script class and add it to the entity. Figure 26 shows searching for the correct script.

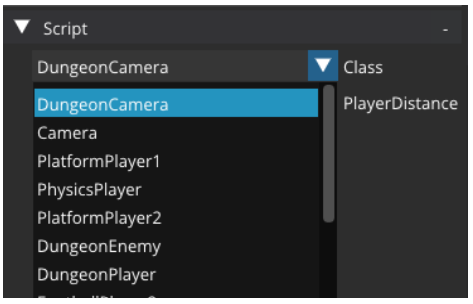

Figure 26. Camera script

After finding the script, we need to write some code. We create a “PlayerDistance” variable which will be used as the camera translation ‘Z’ value, so that the camera is always “PlayerDistance” far from the player and we set it on the editor as already shown in Figure 25. On the “OnCreate()” function when the camera gets created, we search for the player game object. Then, every frame using the “OnUpdate()” method, if we have successfully found the player entity, we set its ‘X’ and ‘Y’ translation values to the camera, so that the camera always tracks the player. Code 14 shows the script written for the perspective camera while Figure 22 has already depicted how the perspective camera will look like inside the game.

```
public class DungeonCamera : Core
{
    public float PlayerDistance;

    private Core player;

    void OnCreate()
    {
        player = FindObjectByName("Player");
    }

    void OnUpdate()
    {
        if (player != null)
            Translation = new Vector3(player.Translation.XY, PlayerDistance);
    }
}
```

Code 14. Camera script

6.2.4 Player

To create the player entity, the majority of the steps remain consistent with the previous instructions. However, certain modifications are necessary. These include:

- A script, so the player can move and change behavior when colliding with entities.
- The player’s texture to the sprite renderer component.
- The rigidbody type is dynamic so the player can move.

Figure 27 is used to show the player components.

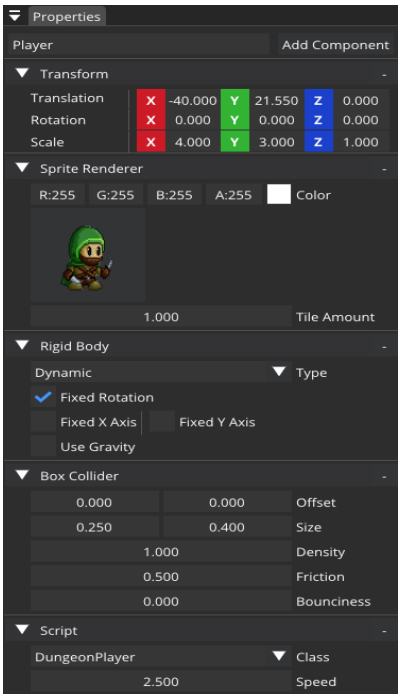

Figure 27. Player components

As it can be seen in the figure 27, the player has a script in order to be able to move around the game world. When initializing the player entity, we create some variables needed for both the game and the player, like the player speed and lives. Then, in the "OnCreate()" method, we retrieve the player transform and rigid body components, we also retrieve the transform of the images and we search through the scene to find other needed game objects. More about the images in Section 6.2.6. Code 15 is used to show the script written to initialize the player and Code 16 shows the player "OnCreate()" function.

```
class DungeonPlayer : Core
{
    public float Speed;

    private bool yellow = false;
    private bool green = false;
    private bool red = false;
    private bool gotHit = false;

    private int life = 6;

    private TransformComponent transform;
    private RigidBodyComponent rigidBody;

    private TransformComponent winTransform;
    private TransformComponent looseTransform;

    private Core winY, winG, winR;
    private Core enemy1, enemy2, enemy3, enemy4, enemy5, enemy6, enemy7, enemy8;
```

Code 15. Player script initialization

```
void OnCreate()
{
    transform = GetComponent<TransformComponent>();
    rigidBody = GetComponent<RigidBodyComponent>();

    winTransform   = FindObjectByName("Image").GetComponent<TransformComponent>();
    looseTransform = FindObjectByName("Image2").GetComponent<TransformComponent>();

    winY = FindObjectByName("WinYellow");
    winG = FindObjectByName("WinGreen");
    winR = FindObjectByName("WinRed");

    enemy1 = FindObjectByName("Enemy1");
    enemy2 = FindObjectByName("Enemy2");
    enemy3 = FindObjectByName("Enemy3");
    enemy4 = FindObjectByName("Enemy4");
    enemy5 = FindObjectByName("Enemy5");
    enemy6 = FindObjectByName("Enemy6");
    enemy7 = FindObjectByName("Enemy7");
    enemy8 = FindObjectByName("Enemy8");
}
```

Code 16. Player OnCreate() method

Moreover, each frame we need to update the player's behavior. In the "OnUpdate()" method, the following functionalities are performed:

- Move the player:
  - Move the player using WASD by applying linear impulse to the entity.
  - Flip the player to always look at the correct direction
- Check for the player's win or loss condition:
  - Make sure the player loses a life when gets hit.
  - Verify if the player has not yet achieved victory or suffered a loss.
- Display the appropriate win or loss image:
  - If the player has achieved victory or encountered a loss, present the corresponding image by changing its translation and placing it at the middle of the game world.
- Halt player movement:
  - When the game is won or lost, cease the player's movement.

- Prevent further input or actions from affecting the player.

Code 17 shows the script written inside the “OnUpdate()” method to change the player and game world behavior.

```
void OnUpdate(float ts)
{
    if (gotHit) { life--; gotHit = false; }

    if ((!yellow || !green || !red) && life > 0)
    {
        rigidBody.SetLinearVelocity(Vector2.Zero);

        Vector2 velocity = Vector2.Zero;

        if (Input.IsKeyDown(KeyCode.W))
            velocity.Y = 700;
        else if (Input.IsKeyDown(KeyCode.S))
            velocity.Y = -700;

        if (Input.IsKeyDown(KeyCode.A))
        {
            velocity.X = -700;
            transform.SetScale(new Vector3(-4, 3, 1)); // Flip player
        }
        else if (Input.IsKeyDown(KeyCode.D))
        {
            velocity.X = 700;
            transform.SetScale(new Vector3(4, 3, 1)); // Flip player
        }

        velocity *= Speed * ts;
        rigidBody.ApplyLinearImpulse(velocity, true);
    }
    else
    {
        if (life == 0)
            looseTransform.Translation = Vector3.ZOne;
        else
            winTransform.Translation = Vector3.ZOne;

        rigidBody.SetLinearVelocity(Vector2.Zero);
        rigidBody.SetAngularVelocity(0);
    }
}
```

Code 17. Player OnUpdate() method

Finally, we need to check every frame for player collisions. If the player has collided with one of the three winning rectangles, we need to store a value to represent the collision with that rectangle, so we can use it in the “OnUpdate()” method to check if the player has won the game or not. On the other hand, if the player has collided with an enemy, we again need to store the collision in a variable, so we can use it in the “OnUpdate()” method to remove a life from the player. Code 18 is used to show the script written inside the “OnCollisionBegin()” method.

```
void OnCollisionBegin(ulong ent1ID, ulong ent2ID)
{
    if (HasCollided(ID, ent1ID, ent2ID))
    {
        if (HasCollided(winy.ID, ent1ID, ent2ID))
            yellow = true;
        else if (HasCollided(wing.ID, ent1ID, ent2ID))
            green = true;
        else if (HasCollided(winr.ID, ent1ID, ent2ID))
            red = true;

        if (HasCollided(enemy1.ID, ent1ID, ent2ID) || HasCollided(enemy2.ID, ent1ID, ent2ID) ||
            HasCollided(enemy3.ID, ent1ID, ent2ID) || HasCollided(enemy4.ID, ent1ID, ent2ID) ||
            HasCollided(enemy5.ID, ent1ID, ent2ID) || HasCollided(enemy6.ID, ent1ID, ent2ID) ||
            HasCollided(enemy7.ID, ent1ID, ent2ID) || HasCollided(enemy8.ID, ent1ID, ent2ID))
        {
            gotHit = true;
        }
    }

    bool HasCollided(ulong a, ulong b, ulong c) { return a == b || a == c; }
}
```

Code 18. Player OnCollisionBegin() method

6.2.5 Enemies

To create the enemies entities, we follow the same steps as creating the player. The only difference lies in the script and the texture of the enemy. Figure 28 is used to show the enemy components.

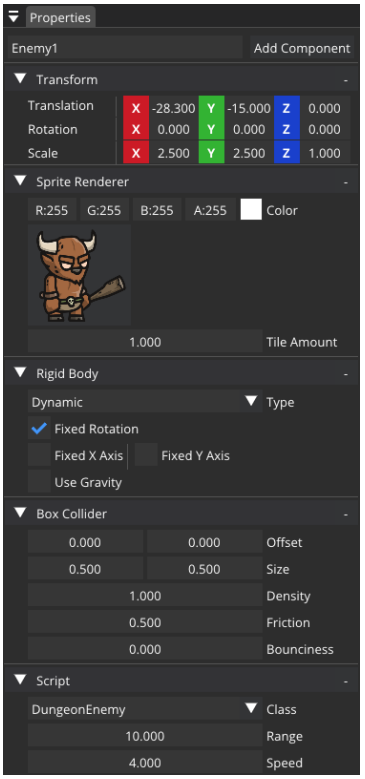

Figure 28. Enemy components

The enemies have their own C# script. We again start by initializing some variables needed for the enemy entities and then we invoke the "OnCreate()" method. Inside this method, we retrieve and store the enemy's transform and rigid body components as well as the player's. Code 19 shows the enemy initialization.

```
class DungeonEnemy : Core
{
    public float Speed;
    public float Range;

    private TransformComponent transform;
    private RigidBodyComponent rigidBody;

    private Core player;
    private TransformComponent playertr;
    private RigidBodyComponent playerRb;

    private bool hasCollided;

    void OnCreate()
    {
        transform = GetComponent<TransformComponent>();
        rigidBody = GetComponent<RigidBodyComponent>();

        player   = FindObjectByName("Player");
        playertr = player.GetComponent<TransformComponent>();
        playerRb = player.GetComponent<RigidBodyComponent>();
    }
```

Code 19. Enemy script initialization

Then, each frame we need to update the enemy's behavior. In the "OnUpdate()" method, the following functionalities are performed:

- Move the enemy:
  - Make the enemy look at the player direction at all times
  - Calculate the enemy position relevant to the player position.
  - If their distance is lower than a fixed distance, move the enemy towards the player to attack him.
- Kill the enemy:
  - If the player collides with an enemy, remove the enemy from the game world and push the player backwards with a negative force.

Code 20 shows the script written inside the "OnUpdate()" method.

```
void OnUpdate(float ts)
{
    Vector2 enemyToPlayer = playerRb.Position - rigidBody.Position;
    float distance = enemyToPlayer.Length();

    if (rigidBody.Position.X > playerRb.Position.X)
        transform.SetScale(new Vector3(-2.5f, 2.5f, 1)); // Flip enemy
    else
        transform.SetScale(new Vector3( 2.5f, 2.5f, 1)); // Flip enemy

    if (distance <= Range)
    {
        Vector2 direction = enemyToPlayer.Normalize();

        rigidBody.SetLinearVelocity(direction * Speed); // Follow player
    }

    if (hasCollided)
    {
        Vector2 difference = (Translation.XY - playertr.Translation.XY).Normalize();
        difference *= 10000 * 10000;
        playerRb.ApplyForce(difference.Negative()); // Push player
        transform.SetScale(Vector3.Zero);
        rigidBody.SetLinearVelocity(Vector2.Zero);
        rigidBody.SetTransform(new Vector2(-60f, 0f)); // Remove enemy
        hasCollided = false;
    }
}
```

Code 20. Enemy OnUpdate() method

Finally, we need to check every frame for enemy collisions. If the enemy has collided with the player, we need to store the collision in a variable, so we can use it in the "OnUpdate()" method to kill the enemy and push the player. Code 21 is used to show the script written inside the "OnCollisionBegin()" method.

```
void OnCollisionBegin(ulong ent1ID, ulong ent2ID)
{
    if (HasCollided(ID, ent1ID, ent2ID) && HasCollided(player.ID, ent1ID, ent2ID))
        hasCollided = true;

    bool HasCollided(ulong a, ulong b, ulong c) { return a == b || a == c; }
}
```

Code 21. Enemy OnCollisionBegin() method

6.2.6 Victory

As we have made everything, the only thing left to do is to create the 3 colorful rectangles, which are used as the places the player has to visit in order to win the game. For each of the rectangle entities, we position them somewhere in the map and then we add a sprite renderer component to give them a specific color. Figure 29 shows the components of one of the end goals.

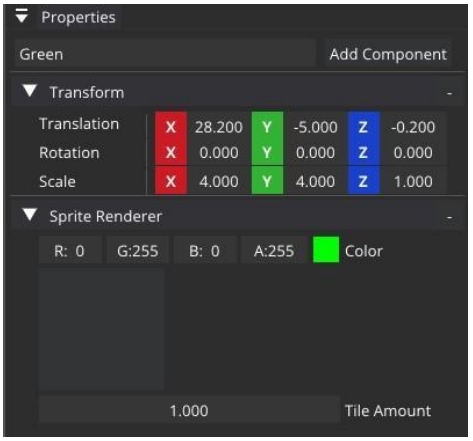

Figure 29. Goal components

Finally, somewhere outside of the camera view, we create and place 2 entities that are used as our win or lose images. As the engine does not currently support showing and hiding a game object, we will just place these entities outside the camera view, and when the game ends, depending on the state (win or lose) we will move the corresponding entity to the middle of the game world. Figure 30 is used to show what happens if the player manages to visit all the colorful goals and win the game.

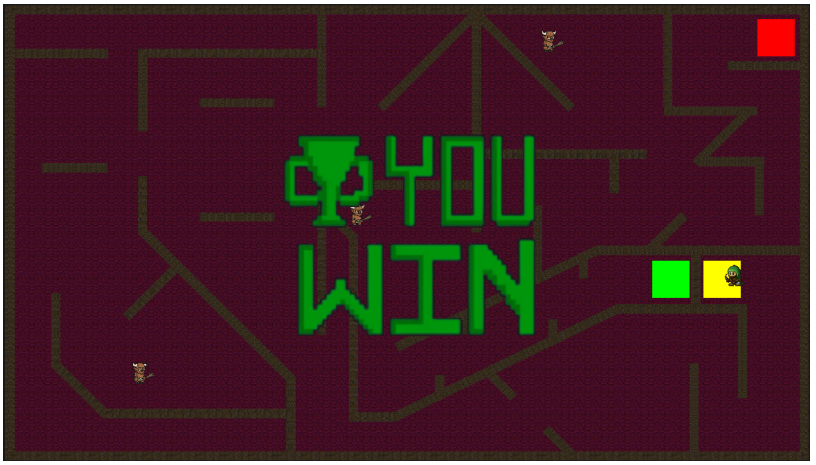

Figure 30. End of game

To make the game more fascinating we could add a script to each of the goals and when the game starts, position both the goals, the player and the enemies to random locations inside the map. To do that we just need to make sure that there is no wall in that position and then we can set the entities transform.

We should note that the assets used for the player and enemy models as well as the wall texture were all downloaded from the "Craftpix" website [147]. On the other hand, the "You Win" image was downloaded from the "VHV.RS" website.

6.3 Alpha Testing - Other Projects

Following the development of each engine component, we conducted tests to ensure optimal functionality and identify any potential issues. After most of the game engine parts were developed, we carried out alpha testing by creating some scenes and games with it. So, to ensure our game engine functions properly, we developed a variety of games and scenes designed as test cases rather than products to publish. This approach is likely the most effective to find and

resolve bugs and issues, while also identifying potential features that might need to be implemented and added to the engine. A good way to start is by creating a simple first game and afterwards continue with more advanced tests like stress testing. Our strategy involved creating games targeted at specific subsystems and features, such as testing the assets management subsystem, testing if entity batch rendering works, and even if physics features work like applying impulses to entities. To further motivate us, we gave ourselves a time limitation of just one month to build as many games and scenes as we could while also solving issues, adding new features, and improving the overall performance of the engine. The results of this alpha testing served as critical feedback for us and our game engine. The next subsections are used to showcase some of the projects, tests and improvements we did during the alpha testing.

6.3.1 Pong

We will start with the simplest game we developed, which was also the first game we made, and it is none other than the classic game released by Atari: “Pong” [142]. Pong is as simple as a 2D table tennis themed game could be. There are two players, four walls and one ball. The objective of the game is to be the first one to score 11 points by hitting the back wall of the opponent's paddle. To add our touch, we chose to implement it vertically instead of horizontally as it can be seen in Figure 31.

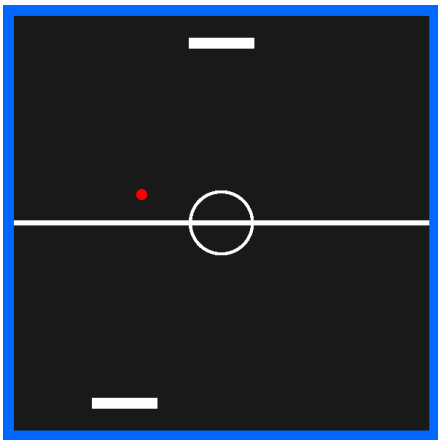

Figure 31. Pong game

6.3.2 Football

The second game we made, was created to test the world gravity, as well as if the textures given on entities worked as expected. With that goal in mind, we chose to create a fun two-player game by the name of: “Sports Heads Football” [143]. This football themed game is very simple as there are only two players and one ball. The objective is to be the first one to score 5 goals, but this time, compared to Pong, the ball can bounce around the field and the players can both jump and kick the ball, instead of just colliding with it and moving it around. The assets used for the players and the ball were downloaded from “Vecteezy” [144] while the background is a texture created by Qureshi and has been uploaded to the “Behance” website [145]. Figure 32 is used to depict the football game.

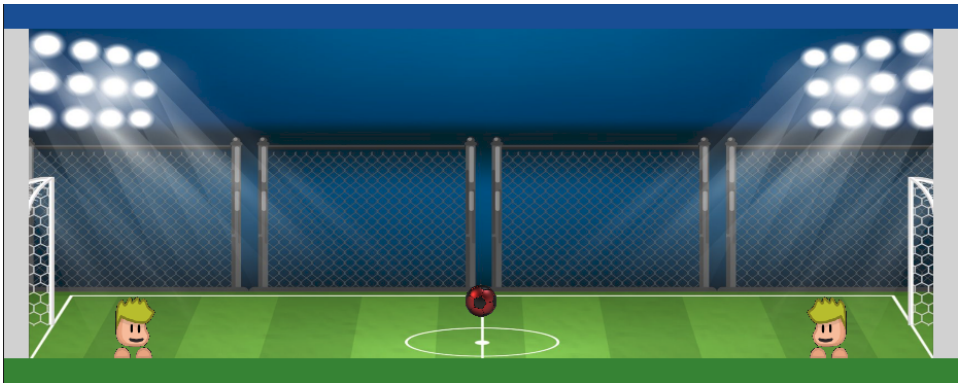

Figure 32. Football game

6.3.3 Platform - Fighting Game

Our next game was inspired by the "Super Smash Bros" game series which were published by Nintendo [146]. It is a basic platform fighting game which consists of multiple platforms and two players. The objective of the game is to push the opponent outside of the game screen so that they cannot return to the platforms. When the two players collide, they get pushed back with a negative force, but each player can also attack the other and push them even further back when hitting them. All the assets used in this game were downloaded from "Craftpix" [147]. In Figure 33 we present the platform game.

Figure 33. Platform game

6.3.4 Top-Down Race

Another thought we had was to test our collision system, some other physics forces like the linear and angular velocity, and some of the features provided by the scripts of the scripting engine. We chose to build a simple game with multiple colliders. To make it entertaining, we thought that we should make a racing game. Its objective is to drive the car using the arrow keys, from the start to the finish line without touching any of the walls. Every time the car collides with a wall, the car position resets to the starting position. All assets used for this game were downloaded from OpenGameArt [148]. Figure 34 shows a snapshot of the racing game we developed.

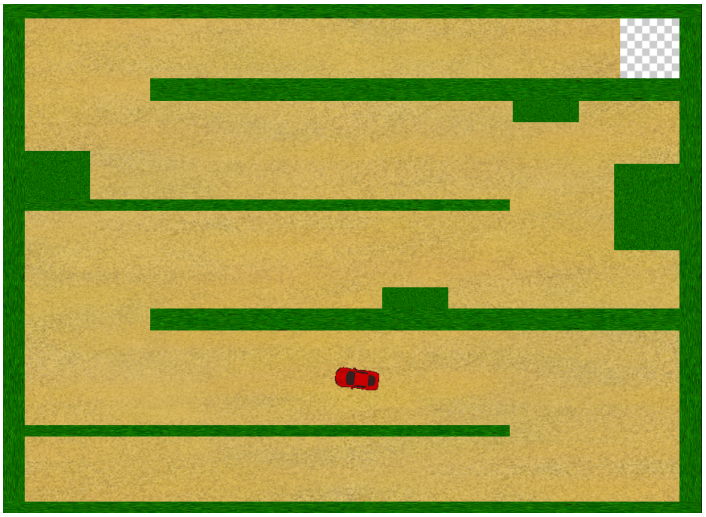

Figure 34. Racing game

6.4 Bugs and Improvements

While testing the engine by creating the games mentioned above and some other scenes, we made the following improvements and additions to our game engine:

- Physics features: While we did not add many features during the creation of our game engine, we found most of them essential for our game development. By making use of the functionality of our physics engine (Box2D), we were able to add gravity, entity position and rotation freezing, entity forces and many more necessary features, enhancing the realism and physics-based interactions within the games.
- Script features: By adding more features to the scripting engine, we managed to create a dynamic and interactive menu in the user interface of the engine, where the developers can now change the transform and other entity fields during both the development and the playtime of the scene.
- Optimizing our editor: The editor interface underwent a significant revamp to improve usability and make the game development process smoother. This includes changes such as a more intuitive layout, improved navigation, and additional features to enable efficient game development, editing and testing.

These optimizations were added to make our game engine more efficient and flexible. As this paper is intended for someone who wishes to create their own custom engine, it is crucial not to overlook this step, as it is essential to further improve the game engine and the products developed by it. In Figure 35, we present an example of how the engine behaves when we tested it by playing the “Platform - Fighting” game while also trying to cap the scene frames at 60 fps.

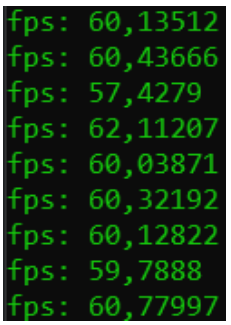

Figure 35. Trying to cap the frames of the Platform - Fighting game at 60fps

During the testing phase, we identified some issues and certain anomalies such as crashes or non-desirable results. For example, the scene would crash if the end-user attempted to pause the game while multiple physics calculations were occurring simultaneously, resulting in a complete failure of the engine. Moreover, certain features would not work as expected, such as physics collisions failing to calculate correctly causing errors to appear, and lines not rendering in the correct layer. We have worked on fixing these problems and many more, but it is essential for the end-users to further test the engine for any future improvements. Our future plans also involve adding more subsystems and features to the engine, hoping to expand it and improve it even more.

6.5 Future Expansions and Extensions

One crucial addition we want our engine to support is Audio. We plan on integrating OpenAL Soft [149] which would make the overall game experience more immersive and enjoyable. Additionally, we aspire to integrate one of the most important subsystems, an "Animation" subsystem. Animated scenes, assets and characters can truly transform a game. Furthermore, we plan on redesigning our "Assets Management" subsystem to improve it and make it possible to import models from well-known graphics softwares like "Blender" and "Autodesk Maya". This expansion will provide game developers with greater flexibility when including custom models into their games.

Moreover, we recognize the importance of "Text Rendering" in a game engine. Allowing game developers to create game menus and other user interface elements like hearts for the players' lives or numbers for a game score, will enhance the overall user experience. We plan on including a MSDF generator [150] to make text rendering possible. To further improve the gaming experience, we are thinking of implementing a "Particle" subsystem. This subsystem will make it possible to create effects like smoke, explosions, sparks and more, making the games we will create more enjoyable. We also plan on redesigning our "Scene Management" subsystem. Even though we currently can create, save and open a scene there are necessary features missing, with the most important the scene transitions. Right now, it is not possible to transition between different scenes or levels inside a game.

Finally, another important subsystem we want to integrate to the engine is the "Profiling" subsystem. Profiling makes it possible to analyze both the performance of the engine and a standalone game. This way we will be able to identify performance bottlenecks, optimize resource usage, and improve the overall efficiency of the engine. We are thinking of integrating a lightweight profiler library such as "Easy_Profiler" [151].

7 Limitations

The game engine presented was implemented by the first two authors in the context of their Bachelor thesis under the supervision of the third author. The developers were finishing their studies in Applied Informatics, but they had no prior experience in the field of game programming and game engine development.

In addition to the aforementioned limitations, there is also a lack of extensive literature and tutorials dedicated to building complete game engines from scratch. Even though there are limited resources available, for those interested in learning more in-depth about game engine development, we highly recommend exploring the ones we have presented in the related work and throughout the presentation of the various subsystems of the game engine implemented. As already mentioned in Section 2, these resources played a crucial role in expanding our knowledge

and providing guidance throughout the development of our engine. Without the information, instructions and directions provided by these sources, we would have faced major obstacles in our progress and the end product would not have been the same. Although the game engine presented does not implement all the potential subsystems of a game engine, it is a functional engine and the roadmap proposed for implementing it can be easily replicated by interested readers.

It should also be noted that as outlined in Section 6, only the functionality of the engine was checked through the development of applications designed as test cases. There has not been any comprehensive metric testing, such as evaluations of code quality, coverage, or even qualitative metrics. Even though we recognize the necessity of these assessments and we are committed to conducting further testing in the future, by adding a profiling subsystem, to make it possible to analyze both the performance of the engine as well as standalone games, we should note that the main goal we had in mind while creating the game engine, was not to implement a more efficient engine in comparison to others, but to provide a method of developing a game engine and to validate the proposed methodology by leveraging free software.

Finally, as our engine supports physics calculations, and game scripting, making it possible to real-time update entity behavior, which means that a wide range of 2D games could be developed. We should mention though, that certain limitations affect the scope and features of the games that can be created. Notably, the absence of an Audio subsystem results in a dull game experience. Furthermore, without an Animation subsystem, we cannot create animated scenes and assets, and as a Particle subsystem has not been implemented yet we also cannot create special effects. Moreover, we cannot develop games with multiple levels as we do not have a Scene management subsystem to make it possible to smoothly transition between scenes. Lastly, the lack of a Text Rendering subsystem prevents the development of game menus and other essential user interface elements.

8 Conclusion

In this paper, we have documented the development process of our own 2D game engine and how it is possible for anyone else interested in the subject to create a custom one according to their needs. Game engines are a topic that holds significant interest for game developers. Whether driven by curiosity, dissatisfaction with existing engines, or a desire to avoid licensing costs, we firmly believe that if someone is a game developer or aspires to become one, learning how the subsystems of a game engine work and how they are tied together, can greatly benefit them in the future.

Throughout this educational journey, we gained valuable and unique knowledge which we hope to pass forward. We managed to understand what a game engine is, what are the essential subsystems it requires to become fully functional, and which other additional subsystems it could have, to offer further useful features and tools. Although we did not focus on a specific subsystem for in-depth implementation, for example the rendering engine, we intentionally pursued knowledge across all subsystems even if it meant using existing APIs and libraries instead of implementing them ourselves. However, this approach will not be proved to be any less educational and useful for someone who is interested in the subject and wishes to create a similar or a completely different engine.

We have come to the conclusion that, even though developing a game engine from scratch is a challenging and time-consuming project, it has proven to be a remarkable learning experience. To anyone considering starting their own game engine development journey, we encourage pursuing it. The knowledge and personal growth attained will create a rewarding experience.

9 References

1. What is a Game Engine?. Available online: https://usv.edu/blog/what-is-a-game-engine (accessed on 01 March 2023).
2. What is a Game Engine?. Available online: https://fullscale.io/blog/what-is-game-engine (accessed on 10 March 2023).
3. Chernikov Y. What is a GAME ENGINE?. Available online: https://www.youtube.com/watch?v=vtWdgtMo1T4 (accessed on10 March 2023).
4. Making Your Own Video Game Engine: The Beginners Guide. Available online: https://www.gamedesigning.org/learn/make-a-game-engine (accessed on 10 March 2023).
5. What is a Game Engine?. Available online: https://gamescrye.com/blog/what-is-a-game-engine (accessed on 10 March 2023).
6. Waikar T. How I made a game engine from scratch?. Available online: https://medium.com/the-virtual-diary/how-i-made-a-game-engine-from-scratch-bcacb2df0503 (accessed on 10 March 2023).
7. Serrano H. How does a Game Engine work? An Overview. Available online: https://www.haroldserrano.com/blog/how-do-i-build-a-game-engine (accessed on 10 March 2023).
8. Halpern J. The What and Why of Game Engines. Available online: https://medium.com/@jaredehalpern/the-what-and-why-of-game-engines-f2b89a46d01f (accessed on 10 March 2023).
9. Violini R. What Is A Game Engine And How Does It Work?. Available online: https://gamedevelopertips.com/what-is-a-game-engine-and-how-does-it-work (accessed on 10 March 2023).
10. The Complete Game Engine Overview. Available online: https://www.perforce.com/resources/vcs/game-engine-overview (accessed on 10 March 2023).
11. Gregory, J. Game Engine Architecture, 3rd ed.; CRC Press: London, UK, 2018; 1240.
12. Thorn, A. Game Engine Design and Implementation, 1st ed.; Jones & Bartlett Learning: MA, USA, 2010; 594.
13. Bishop, L.; Eberly, D.; Whitted, T.; Finch, M.; Shantz, M. Designing a PC game engine. IEEE Computer Graphics and Applications 1998, vol. 18, pp. 46-53.
14. Guana, V.; Stroulia, E.; Nguyen, V. Building a Game Engine: A Tale of Modern Model-Driven Engineering. IEEE/ACM 4th International Workshop on Games and Software Engineering 2015, pp. 15-21.
15. Godot Engine. Available online: https://godotengine.org (accessed on 11 March 2023).
16. Bevy Engine. Available online: https://bevyengine.org (accessed on 11 March 2023).
17. OpenRA Engine. Available online: https://www.openra.net (accessed on 11 March 2023).
18. Pyxel Engine. Available online: https://github.com/kitao/pyxel (accessed on 11 March 2023).
19. Minetest Engine. Available online: https://www.minetest.net (accessed on 11 March 2023).
20. Ebitengine Engine. Available online: https://ebitengine.org (accessed on 11 March 2023)
21. PlayCanvas Engine. Available online: https://playcanvas.com (accessed on 11 March 2023).
22. Flame Engine. Available online: https://flame-engine.org (accessed on 11 March 2023).
23. GDevelop Engine. Available online: https://gdevelop.io (accessed on 11 March 2023).
24. Stride Engine. Available online: https://www.stride3d.net (accessed on 11 March 2023).
25. Game Engine Programming. Available online: https://www.youtube.com/playlist?list=PLU2nPsAdxKWQYxkmQ3TdbLsyc1l2j25XM (accessed on 12 July 2023).
26. Game Engine. Available online: https://www.youtube.com/playlist?list=PLlrATfBNZ98dC-V-N3m0Go4deliWHPFwT (accessed on 12 July 2023).
27. Coding a 2D Game Engine in Java. Available online: https://www.youtube.com/playlist?list=PLtrSb4XxIVbp8AKuEAlwNXDxr99e3woGE (accessed on 12 July 2023).
28. Tutorial: Create 2D Game Engine using C++ and SDL. Available online: https://www.youtube.com/playlist?list=PL-K0viiuJ2RctP5nlJlqmHGeh66-GOZR_ (accessed on 12 July 2023).
29. Let's make an engine!. Available online: https://www.youtube.com/playlist?list=PL7lh9ryRNHSIzqKzEdYPG94B0uvfqhHpb (accessed on 12 July 2023).
30. Eberly, D. H. 3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics, 2nd ed.; CRC Press: London, UK, 2006; 1040.

31. Bauchinger, M. Designing a Modern Rendering Engine. Master's Thesis, Vienna University of Technology, Vienna, Austria, 2007.
32. Coding a 2D Physics Engine. Available online: https://www.youtube.com/playlist?list=PLtrSb4XxIVbpZpV65kk73OoUcIrBzoSiO (accessed on 13 July 2023).
33. Gutekanst S. Let's build an Entity Component System from scratch. Available online: https://devlog.hexops.com/2022/lets-build-ecs-part-1/ (accessed on 13 July 2023).
34. Colson D. How to make a simple entity-component-system in C++. Available online: https://www.david-colson.com/2020/02/09/making-a-simple-ecs.html (accessed on 13 July 2023).
35. Learn C++. Available online: https://www.learncpp.com/ (accessed on 01 September 2022).
36. Learn-cpp.org. Available online: https://www.learn-cpp.org/ (accessed on 01 September 2022).
37. C++ Programming Course - Beginner to Advanced. Available online: https://www.youtube.com/watch?v=8jLOx1hD3_o (accessed on 07 September 2022).
38. C++ FULL COURSE For Beginners (Learn C++ in 10 hours). Available online: https://www.youtube.com/watch?v=GQp1zzTwrIg (accessed on 08 September 2022).
39. C++ Full Course for free. Available online: https://www.youtube.com/watch?v=-TkoO8Z07hI (accessed on 09 September 2022).
40. UnityYAML. Available online: https://docs.unity3d.com/2023.2/Documentation/Manual/UnityYAML.html (accessed on 14 April 2023).
41. What are AAA Games?. Available online: https://www.arm.com/glossary/aaa-games (accessed on 17 April 2023).
42. What are the best cross-platform 3D graphics APIs?. Available online: https://www.slant.co/topics/5346/cross-platform-3d-graphics-apis (accessed on 17 April 2023).
43. DirectX. Available online: https://en.wikipedia.org/wiki/DirectX (accessed on 17 April 2023).
44. White S., Hickey S., Bridge K., McClister C., Wojciakowski M. Windows game development guide. Available online: https://learn.microsoft.com/en-us/windows/uwp/gaming/e2e (accessed on 28 April 2023).
45. White S., Jahiu D., Coulter D., Batchelor D., Jacobs M., Satran M. Graphics APIs in Windows. Available online: https://learn.microsoft.com/windows/direct3darticles/graphics-apis-in-windows (accessed on 28 April 2023).
46. Tuttle W. Defining the Next Generation: An Xbox Series X|S Technology Glossary. Available online: https://news.xbox.com/en-us/2020/03/16/xbox-series-x-glossary (accessed on 28 April 2023).
47. Metal. Available online: https://developer.apple.com/metal (accessed on 17 May 2023).
48. Galvan A. Raw Metal. Available online: https://alaingalvan.medium.com/raw-metal-a64b861bcdeb (accessed on 17 May 2023).
49. Metal (API). Available online: https://en.wikipedia.org/wiki/Metal_(API) (accessed on 17 May 2023).
50. Vulkan. Available online: https://en.wikipedia.org/wiki/Vulkan (accessed on 17 May 2023).
51. Vulkan. Available online: https://www.vulkan.org (accessed on 17 May 2023).
52. Vulkan. Available online: https://vulkan.lunarg.com/sdk/home (accessed on 17 May 2023).
53. Vulkan Driver Support. Available online: https://developer.nvidia.com/vulkan-driver (accessed on 17 May 2023).
54. Seedhouse A. Nintendo Switch Supports Vulkan, OpenGL 4.5 And OpenGL ES 3.2. Available online: https://www.nintendo-insider.com/nintendo-supports-vulkan-opengl-4-5 (accessed on 20 May 2023).
55. Strickland D. Nintendo Switch certified for Vulkan and OpenGL 4.5. Available online: https://www.tweaktown.com/news/55537/nintendo-certified-vulkan-opengl-4-5 (accessed on 20 May 2023).
56. Palumbo A. Nintendo Switch Officially Supports Vulkan, OpenGL 4.5 & OpenGL ES. Available online: https://wccftech.com/nintendo-switch-supports-vulkan (accessed on 20 May 2023).
57. OpenGL. Available online: https://www.opengl.org (accessed on 20 May 2023).
58. Platform Specific. Available online: https://www.khronos.org/opengl/wiki/Platform_Specific (accessed on 20 May 2023).
59. White S., Coulter D., Batchelor D., Jacobs M., Satran M. OpenGL. Available online: https://learn.microsoft.com/en-us/windows/win32/opengl/opengl (accessed on 20 May 2023).
60. About OpenGL for OS X. Available online: https://developer.apple.com/documentation/GraphicsImaging/OpenGL (accessed on 20 May 2023).
61. Displaying graphics with OpenGL ES. Available online: https://developer.android.com/develop/ui/views/graphics/opengl (accessed on 20 May 2023).
62. The Standard for Embedded Accelerated 3D Graphics. Available online: https://www.khronos.org/opengles (accessed on 20 May 2023).

63. LOW-LEVEL 3D GRAPHICS API BASED ON OPENGL ES. Available online: https://www.khronos.org/webgl (accessed on 20 May 2023).
64. WebGL: 2D and 3D graphics for the web. Available online: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API (accessed on 20 May 2023).
65. Deveria A. WebGL - 3D Canvas graphics. Available online: https://caniuse.com/webgl (accessed on 20 May 2023).
66. Iliev H. OpenGL vs Vulkan. Available online: https://thatonegamedev.com/cpp/opengl-vs-vulkan (accessed on 26 May 2023).
67. Ullevig E. OpenGL vs Vulkan: What are the Key Differences?. Available online: https://history-computer.com/opengl-vs-vulkan-what-are-the-key-differences (accessed on 26 May 2023).
68. Wenger K. Clearing Up Vulkan Misconceptions – Why Is It a Step Above OpenGL? Available online: https://coreavi.com/clearing-vulkan-misconceptions-why-is-it-above-opengl (accessed on 26 May 2023).
69. Vries J. D. Learn OpenGL. Available online: https://learnopengl.com (accessed on 12 July 2023).
70. Overvoorde A. OpenGL - Introduction. Available online: https://open.gl (accessed on 12 July 2023).
71. Chernikov Y. Welcome to OpenGL. Available online: https://www.youtube.com/playlist?list=PLlrATfBNZ98foTJPJ_Ev03o2oq3-GGOS2 (accessed on 12 July 2023).
72. Shah M. docs.GL. Available online: https://docs.gl (accessed on 12 July 2023).
73. Jorge Rodríguez. Docg.gl, Available online: https://docs.gl (accessed on 12 July 2023).
74. Vertex Shader. Available online: https://www.khronos.org/opengl/wiki/Vertex_Shader (accessed on 29 May 2023).
75. Rendering Pipeline Overview. Available online: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview (accessed on 29 May 2023).
76. Fragment Shader. Available online: https://www.khronos.org/opengl/wiki/Fragment_Shader (accessed on 29 May 2023).
77. Tukalo A. Introduction to Shaders. Available online: https://lightningchart.com/blog/introduction-to-shaders (accessed on 29 May 2023).
78. Shader Compilation. Available online: https://www.khronos.org/opengl/wiki/Shader_Compilation (accessed on 29 May 2023).
79. GlTexImage2D. Available online: https://registry.khronos.org/OpenGL-Refpages/es3.0/html/glTexImage2D (accessed on 29 May 2023).
80. Krzeminski M. OpenGL Batch Rendering. Available online: https://www.gamedev.net/tutorials/programming/graphics/opengl-batch-rendering (accessed on 29 May 2023).
81. Dear ImGui. Available online: https://github.com/ocornut/imgui (accessed on 25 August 2023).
82. Filesystem library. Available online: https://en.cppreference.com/w/cpp/filesystem (accessed on 25 August 2023).
83. ImGuizmo. Available online: https://github.com/CedricGuillemet/ImGuizmo (accessed on 25 August 2023).
84. ImGui Extension Plus Imguizmo, Implot, Imnodes. Available online: https://evergine.com/imgui-extension-imguizmo-implot-imnodes (accessed on 29 August 2023).
85. Physics engine. Available online: https://en.wikipedia.org/wiki/Physics_engine (accessed on 07 June 2023).
86. Physics engine. Available online: https://www.computerhope.com/jargon/p/physics-engine (accessed on 07 June 2023).
87. Mousa H. 16 Open-source Physics Simulation Engine. Available online: https://medevel.com/os-physics-engine (accessed on 07 June 2023).
88. Jolt Physics. Available online: https://github.com/jrouwe/JoltPhysics (accessed on 07 June 2023).
89. Bullet Real-Time Physics Simulation. Available online: https://pybullet.org/wordpress (accessed on 07 June 2023).
90. Welcome to LiquidFun!. Available online: https://github.com/google/liquidfun (accessed on 07 June 2023).
91. Box2D. Available online: https://box2d.org (accessed on 07 June 2023).
92. Box2D Overview. Available online: https://box2d.org/documentation/index.html (accessed on 07 June 2023).
93. Hogan R. Local Gravity: How to Calculate Yours in 3 Minutes. Available online: https://www.isobudgets.com/how-to-calculate-local-gravity (accessed on 02 September 2023).
94. Martin D. M. Mathematics for Game Development: What to Learn (And Why!). Available online: https://www.matecdev.com/posts/math-for-game-development (accessed on 02 September 2023).
95. Goodman D. The Use of Mathematics in Computer Games. Available online: https://nrich.maths.org/1374 (accessed on 02 September 2023).
96. GLM. Available online: https://github.com/g-truc/glm (accessed on 02 September 2023).
97. MathFu. Available online: https://github.com/google/mathfu (accessed on 02 September 2023).

98. Rungta K. Entity Component System. Available online: https://www.guru99.com/entity-component-system (accessed on 02 September 2023).
99. Entity component system. Available online: https://en.wikipedia.org/wiki/Entity_component_system (accessed on 02 September 2023).
100. Newton C. Main Principles for Using ECS in Game Development. Available online: https://bagogames.com/main-principles-for-using-ecs-in-game-development (accessed on 02 September 2023).
101. Mertens S. Building Games in ECS with Entity Relationships. Available online: https://ajmmertens.medium.com/building-games-in-ecs (accessed on 02 September 2023).
102. Fox M. Game Engines 101: The Entity/Component Model. Available online: https://www.gamedeveloper.com/programming/game-engines-ECS (accessed on 02 September 2023).
103. Chernikov Y. Entity Component System | Game Engine series. Available online: https://www.youtube.com/watch?v=Z-CILn2w9K0 (accessed on 02 September 2023).
104. Entt. Available online: https://github.com/skypjack/entt (accessed on 02 September 2023).
105. Ragdoll Dynamics. Available online: https://ragdolldynamics.com/ (accessed on 02 September 2023).
106. Flecs. Available online: https://github.com/SanderMertens/flecs (accessed on 02 September 2023).
107. Patel M. A Guide to Event-Driven Architecture Pros and Cons. Available online: https://solace.com/blog/event-driven-architecture-pros-and-cons (accessed on 05 September 2023).
108. Matthew R. A Student's Take: Implementing Event Systems in Games and Using the Preprocessor. Available online: https://www.linkedin.com/pulse/implementing-event-systems-games (accessed on 05 September 2023).
109. Nystrim R. Event Queue. Available online: http://gameprogrammingpatterns.com/event-queue (accessed on 05 September 2023).
110. Kryvytskyi D. Event System [Game engine]. Available online: https://denyskryvytskyi.github.io/event-system (accessed on 05September 2023).
111. Mathewson N., Khuzhin A., Provos N. libevent – an event notification library. Available online: https://libevent.org (accessed on 05 September 2023).
112. Tor. Available online: https://www.torproject.org/ (accessed on 05 September 2023).
113. Chromium. Available online: https://www.chromium.org/Home/ (accessed on 05 September 2023).
114. Input System. Available online: https://docs.unity3d.com/Packages/inputsystem@1.5/manual (accessed on 01 April 2023).
115. Input System. Available online: https://ezengine.net/pages/docs/input/input-overview (accessed on 05 April 2023).
116. Input. Available online: https://topdown-engine-docs.moremountains.com/input (accessed on 05 April 2023).
117. GLFW. Available online: https://github.com/glfw/glfw/blob/master/include/GLFW/glfw3.h (accessed on 05 April 2023).
118. Scripting. Available online: https://docs.unity3d.com/2023.2/Documentation/Manual/ScriptingSection (accessed on 08 September 2023).
119. Blueprints Visual Scripting. Available online: https://docs.unrealengine.com/5.2/blueprints-visual-scripting-in-unreal-engine (accessed on 08 September 2023).
120. How to Start Learning Programming for Game Development. Available online: https://intogames.org/news/getting-started-with--game-development (accessed on 08 September 2023).
121. Huebner R. Adding Languages to Game Engines. Available online: https://www.gamedeveloper.com/programming/add-languages-to-game-engine (accessed on 08 September 2023).
122. Anderson E. F. A Classification of Scripting Systems for Entertainment and Serious Computer Games. Available online: https://core.ac.uk/download/pdf/9599349.pdf (accessed on 08 September 2023).
123. Visual Scripting - How Noodl was inspired by the world of game engines. Available online: https://www.noodl.net/visual-scripting-world-of-game-engines (accessed on 08 September 2023).
124. Bay W. J. What is visual scripting, and how is it used to make video games?. Available online: https://www.gameindustrycareerguide.com/how-is-visual-scripting-used (accessed on 08 September 2023).
125. What Is Visual Scripting & How It Works. Available online: https://www.tabnine.com/blog/what-is-visual-scripting (accessed on 08 September 2023).
126. Unity Visual Scripting. Available online: https://unity.com/features/unity-visual-scripting (accessed on 08 September 2023).
127. C Sharp (programming language). Available online: https://en.wikipedia.org/wiki/C_Sharp_(programming_language) (accessed on 10 September 2023).

128. Warren G., Klonowski B., Pine D., Wagner B., Dykstra T., Lander R., Killeen S., Abraham I., Jawahar T., Sharkey K., Victor Y., Wenzel M., Pierce T., Coulter D., Chowdhury A.R., Card J. What is .NET? Introduction and overview. Available online: https://learn.microsoft.com/en-us/dotnet/core/introduction (accessed on 10 September 2023).
129. .Net. Available online: https://en.wikipedia.org/wiki/.NET (accessed on 10 September 2023).
130. Home repository for .NET Core. Available online: https://github.com/dotnet/core (accessed on 10 September 2023).
131. Mono Project. Available online: https://www.mono-project.com (accessed on 10 September 2023).
132. Mono (software). Available online: https://en.wikipedia.org/wiki/Mono_(software) (accessed on 10 September 2023).
133. Mono. Available online: https://github.com/mono/mono (accessed on 10 September 2023).
134. De George A. S., Schonning N., Coulter D., Bargaoanu L., Lee D., Wenzel M., Aymeric A., Dev A., Jones M., Hoffman M., Latham L., Shengjin Y., Pratt T., 2021. Best Practices for Assembly Loading. Available online: https://learn.microsoft.com/dotnet/framework/deployment/assembly-loading (accessed on 10 September 2023).
135. Shpilt M. Understanding How Assemblies Load in C# .NET. Available online: https://michaelscodingspot.com/assemblies-load-in-dotnet (accessed on 10 September 2023).
136. Nilsson P. Mono Embedding for Game Engines. Available online: https://nilssondev.com/mono-guide/book (accessed on 13 September 2023).
137. Documentation. Available online: https://www.mono-project.com/docs (accessed on 13 September 2023).
138. Mono Documentation. Available online: http://docs.go-mono.com (accessed on 13 September 2023).
139. YAML. Available online: https://yaml.org (accessed on 13 September 2023).
140. JSON, Available online: https://www.json.org/json-en.html (accessed on 13 September 2023).
141. Cereal - A C++11 library for serialization. Available online: https://github.com/USCiLab/cereal (accessed on 13 September 2023).
142. Pong. Available online: https://en.wikipedia.org/wiki/Pong (accessed 26 September 2023)
143. Sports Heads Football. Available online: https://www.twoplayergames.org/game/sports-heads-football (accessed 26 September 2023)
144. Football PNG. Available online: https://www.vecteezy.com/free-png/football (accessed 26 September 2023)
145. Qureshi R.A. Soccer, football game ui/ux, stadium 2d design. Available online: https://www.behance.net/gallery/76998715/soccer-football-game-uiux-stadium-2d-design/modules/447207145 (accessed 26 September 2023)
146. Super Smash Bros. Available online: https://en.wikipedia.org/wiki/Super_Smash_Bros. (accessed 26 September 2023)
147. CraftPix. Available online: https://craftpix.net (accessed 26 September 2023)
148. Open Game Art. Available online: https://opengameart.org (accessed 26 September 2023)
149. OpenAl. Available online: https://github.com/kcat/openal-soft (accessed 26 September 2023)
150. Msdfgen. Available online: https://github.com/Chlumsky/msdfgen (accessed 26 September 2023)
151. Easy_Profiler. Available online: https://github.com/yse/easy_profiler (accessed 26 September 2023)
152. CRYENGINE 5.7 Long Term Support is here!. Available online: https://www.cryengine.com/news/view/cryengine-5-7-long-term-support-is-here (accessed 07 November 2023)
153. CryEngine. Available online: https://en.wikipedia.org/wiki/CryEngine (accessed 07 November 2023)
154. Godot Engine. Available online: https://github.com/godotengine/godot (accessed 07 November 2023)
155. Unreal Engine. Available online: https://en.wikipedia.org/wiki/Unreal_Engine (accessed 07 November 2023)
156. Brodkin J. How Unity3D Became a Game-Development Beast. Available online: https://www.dice.com/career-advice/how-unity3d-become-a-game-development-beast (accessed 07 November 2023)