BeEngine: 2D Game Engine

Final University Project

Video Games Design and Development Degree, UPC

Student: Sunyer Caldú, Guillem

Student’s plan from 2019

Director: Díaz García, Jesús
Summary

This document contains the description of the development of a C++ game engine named BeEngine, as the final university project. The engine is focused on 2D game development and aims to provide all the necessary components and tools to create and deploy a video game from start to finish.

The result is a standalone program that can be executed in any Windows machine, that has the ability to load and manage resources (such as images, scripts, audio, etc.), and allows the user to implement the logic and test the results before generating the final game.

This project goes through some of the techniques and the logic behind the modules and tools of this engine, and the process of implementation followed to accomplish the final results.

The source code for this project: BeEngine 2D Game Engine, it’s public and under the Creative Commons Attribution-ShareAlike 4.0 International License in the GitHub repository (W.2) BeEngine-2D. The build of the engine and the sample project can be found as a GitHub release at (W.21) BeEngine-2D 1.0

Keywords

Game Engine, Video Games, C++, C#, Scripting, GLSL, Physics
License

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.
# Index

1. **Introduction**
   - 1.1 Motivation
   - 1.2 Problem statement
   - 1.3 General objectives
   - 1.4 Specific objectives
   - 1.5 Project scope

2. **State of the art**
   - 2.1 Game Engines
   - 2.2 Libraries and Technology
   - 2.3 Conclusion

3. **Project management**
   - 3.1 Procedure and tools
     - 3.1.1 Gantt
     - 3.1.2 Github and version control
   - 3.2 Validation tools
   - 3.3 Swot analysis
   - 3.4 Risks and contingency plans
     - 3.4.1 Risks during development
     - 3.4.2 Risk of the tasks

4. **Methodology**
   - 4.1 SCRUM
   - 4.2 Tools used for the tracking of the project

5. **Engine development**
   - 5.1 Skeleton
     - 5.1.1 External libraries
     - 5.1.2 Expansion
   - 5.2 Editor
   - 5.2.1 Editor windows
   - 5.3 Resources
     - 5.3.1 Meta files
     - 5.3.2 Asset / Library
     - 5.3.3 File watcher and Asset / Library integrity

---

Sunyer Caldú, Guillem  
**BeEngine: 2D Game Engine**
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4 Entity Component System</td>
<td>45</td>
</tr>
<tr>
<td>5.4.1 GameObjects and Components</td>
<td>45</td>
</tr>
<tr>
<td>5.4.2 Prefabs</td>
<td>45</td>
</tr>
<tr>
<td>5.4.3 Scenes</td>
<td>46</td>
</tr>
<tr>
<td>5.5 Rendering</td>
<td>47</td>
</tr>
<tr>
<td>5.5.1 Debug lines and Gizmos</td>
<td>47</td>
</tr>
<tr>
<td>5.5.2 Scene renderers</td>
<td>48</td>
</tr>
<tr>
<td>5.5.3 Camera rendering</td>
<td>48</td>
</tr>
<tr>
<td>5.6 Scripting</td>
<td>50</td>
</tr>
<tr>
<td>5.6.1 Visual Studio Project</td>
<td>51</td>
</tr>
<tr>
<td>5.6.2 Mono</td>
<td>52</td>
</tr>
<tr>
<td>5.6.3 Script compiling</td>
<td>54</td>
</tr>
<tr>
<td>5.6.4 Game Loop</td>
<td>54</td>
</tr>
<tr>
<td>5.6.5 Editor references</td>
<td>57</td>
</tr>
<tr>
<td>5.7 Physics</td>
<td>58</td>
</tr>
<tr>
<td>5.8 Build</td>
<td>60</td>
</tr>
<tr>
<td>6. Sample Project</td>
<td>62</td>
</tr>
<tr>
<td>6.1 Project development</td>
<td>62</td>
</tr>
<tr>
<td>6.2 Learnings</td>
<td>64</td>
</tr>
<tr>
<td>7. Conclusions</td>
<td>65</td>
</tr>
<tr>
<td>8. Acknowledgments</td>
<td>69</td>
</tr>
<tr>
<td>9. Webgraphy</td>
<td>70</td>
</tr>
</tbody>
</table>
1. Introduction

Every good worker that wants to create, build or repair something always has their tools close: a plumber has a wrench; a painter, his brush; a woodcutter, an ax, and an architect, the pencil. Those tools define the work that they do, the process, its limitations, and its strengths. This is true for every discipline, even for game development.

If you are mildly interested in the video game world or industry, you have probably heard the word "game engine" somewhere; maybe in a new game release, a game studio’s conference, or a console advertisement. A lot of people have heard of it, but not everybody knows exactly what it is, or which role they play inside the video game industry.

A game engine is a software that provides game creators with the necessary set of features to build games quickly and efficiently (we can see an example of a game engine editor on F.1.1). It is a framework for game development that supports and brings together several core areas. You can import art and assets (2D and 3D) from other software, such as Maya or 3s Max or Photoshop; assemble those assets into scenes and environments; add lighting, audio, special effects, physics and animation, interactivity, and gameplay logic; and edit, debug and optimize the content for your target platforms.

F.1.1 - The game engine editor for, Godot Engine (W.13).
1.1 Motivation

Since a very young age, I have been very interested in computers, software, and coding, and I also loved to play videogames. The combination of these two interests made me ask the question of how video games were made, and how was it possible to create such things only with a computer. Therefore, I started researching and digging for answers, and rapidly a whole bunch of game engines was now available for me to start playing with. I did not know exactly what I was doing, but it was fun, and it was the closest thing I had ever been to making video games. Still, the way in which game engines were created and worked was unknown to me, it was just like magic that worked under the hood.

Through my degree in University, I was very happy to start learning how to code, and how game engines worked on the inside. While learning and creating some basic ones, I always felt that I wanted to experiment more and create something that I could use to build video games, without missing any of the tools or utilities that other widely used pre-existing engines include, having control over all the features, knowing exactly how they work.

In addition to that, I have had increasing frustration using standard engines like Unity or Unreal. As they want to please everybody (every platform, and every genre) they end up lacking in some important aspects (such as the barely usable input system from Unity, or the overkill rendering of Unreal for low-end devices).

Finally, while working on the last big project of University (W.1), we used the base code of a student's game engine that had a lot of code structure problems, so we had to improve it and add a lot of tools and optimizations to be able to finish the game in time. With all that new experience, I learned a lot about the things that worked and the things that did not, and how to better structure a big program.

With all of this in mind, the idea of creating my own game engine from scratch, with proper bases and with all the features I wanted, became a reality with this project.

---

W1. https://project-3-upc-ddv-bcn.github.io/Project3/
1.2 Problem statement

The process of creating video games is not trivial and can vary widely depending on a lot of factors, like scope, time, budget, etc., and facing such uncertainty, a highly dynamic tool is necessary. Game engines allow all the different roles involved in game development to merge all their efforts into a final peace, and it must make that process as intuitive, fast, and robust as possible.

When you start getting into game engines, there are a lot of already created options that cover different ranges: Unreal Engine for 3D high-end visuals; Unity for 3D and 2D, especially mobile; Game Maker for fast 2D; Godot for 3D and 2D with an open source approach, and many more. These type of engines offer a lot of advantages since everything is already made for you to start working, but they also have some disadvantages. One of the biggest problems is that as they generally try to please every aspect of game development, they end up falling short in a lot of important aspects, which can cause major time loss and frustration. Another drawback is that on some commercial engines the source code is not accessible since it’s completely private, where it is not allowed or possible to inspect or modify, difficulting the development process.

Facing these lacks, the only other option available is to create a custom game engine from scratch, built specifically for a project, genre, or view.

This project wants to solve the frustrations and problems that come from using generic and bloated engines. It aims to create, a small but complete product, focused on 2D development, with all the tools necessary to build videogames. Instead of trying to please every aspect, it intends to contain a small but fully functional and useful toolset that works as expected. Also, there are a lot of engines that try to be drag-and-drop type (for example, Game Maker), with premade modules that are easy to use, but that at the same time limit the creativity and the options of the developer. This is also something that this project avoids.

As a summary, the solution is to create a balanced engine, that has all the necessary working tools to make 2D games, without limiting the creator.
1.3 General objectives

The objective of this project is to develop an open and public C++ game engine from scratch, with all the necessary tools to create 2D games from start to finish, and also have the capability to deploy on the Windows platform.

O1. Design and develop a 2D game engine with C++ and OpenGL (with shaders).
O2. Build all the necessary tools inside of the engine to allow the user to create a 2D game.
O3. The ability to easily create builds targeting the Windows platform.
O4. Create one or two simple games to demonstrate the capabilities of the engine.

1.4 Specific objectives

To be able to develop and finish the general objectives explained in the last point, we need to break the processes in some more specific tasks and technologies, and define some possible challenges that may appear during development.

1.4.1 Filesystem

SO.1.1 File loading: the most basic functionality of a game engine is the ability to load user files like images, scripts or fonts, and also engine created files like scenes or prefabs

SO.1.2 Resources management: once the files are loaded, it needs to display, serialize, or use them as useful resources for the game creation.

SO.1.3 File watching: If the user or any other program deletes or modifies some of the used files, the engine needs to react accordingly.

SO.1.4 File/Resources persistency: the engine needs to maintain relations between the loaded files and it’s resources.

1.4.2 Component System

SO.2.1 GameObjects: basic entity from where all the game engine logic is based from, that acts as a container for components.

SO.2.2 Components: functional pieces of every game object, with different types of behaviors.

SO.2.3 Hierarchy: game structure and overview of scenes, that defines parent-child connections.
SO.2.4 Inspector: components viewer and logic tweaker.

1.4.3 Rendering

SO.3.1 Shaders: create the necessary ecosystem to work with different types of shaders.

SO.3.2 Debug shapes: create the necessary shaders and geometry to render lines, quads, triangles, circles, etc. to allow debug functionalities.

SO.3.3 Gizmos: build the interactable UI utilities used to transform objects, change shapes, etc.

SO.3.4 Sprites: develop the 2D base image rendering objects.

SO.3.5 Layer management: manage the different layers that 2D rendering demands within all the elements of the pipeline.

1.4.4 Scripting

SO.4.1 Mono implementation: wrap the mono project c# library and its utilities.

SO.4.2 Visual Studio C# solution: automatically create and update a visual studio C# solution with the user code.

SO.4.3 C++ logic mirroring: create the bridge between the C++ and the C# code.

SO.4.4 C++ and C# logic: build the play, stop, and update the scripting logic loop.

1.4.5 Physics

SO.5.1 Box2D implementation: wrap the Box2D physics library around the engine.

SO.5.2 Concave shape management: triangulate 2D polygon shapes to easily use them on Box2D.

SO.5.3 Shapes editing: enable the user to modify the shape and the position of the polygon shape vertices, using gizmos.

1.4.6 UI

SO.6.1 Text rendering: implement the font management and the text rendering utilities for the in-game UI.

SO.6.2 Canvas and scaling: enable the UI under a canvas to resize and scale depending on the screen ratio and screen size

SO.6.3 UI Widgets: create basic UI widgets like buttons, sliders, progress bars, etc...
1.4.7 Build

**SO.7.1 Game / Engine logic separation:** manage the differences between the engine running in editor mode, and running in game mode.

**SO.7.2 Deployment:** create an easy game deployment system, with all the necessary files to run as a standalone application.

1.4.8 Demo games

**SO.8.1 Games showcasing the engine:** Develop one or two simple and fully playable games, that demonstrate all the capabilities of the engine.

1.5 Project scope

Creating a game engine it’s not a trivial job, it requires a lot of time, iteration and exploration. The majority of modern game engines are created by hundreds of different people with different specializations, that work over the years, improving and perfecting their work. Since the amount of time provided for this project is limited, and the number of hands restricted to two, the resultant product may lack polish, accessibility, features and have bugs, it’s still going to be being capable of doing its core job.

This project is created with game developers in mind, for those who are deceived by modern engine standards and want to spend time creating, rather than fighting with their tools. BeEngine is targeted for any small studio or individual that wants a base engine that already works, to create their own games, or modify the source code for further modifications, to decrease production time, and improve workflow and final results.

Finally, this is a project of exploration, personal growth, and self-improvement towards a discipline, that it’s also a passion.
2. State of the art

From the beginning of game engines to the recent 3D high-performance ones, the goal has always remained the same i.e. to give developers a platform to turn their game ideas into reality. But as there are so many types of games and genres, it’s difficult for a single software to cover all the necessities that they may have. In addition to that, with the advances in technology over the years, we find ourselves in a sea of game engines to look at, all of them with different approaches to the different problems that they try to solve.

But that has not always been like that. In the early days, it was all about making individual games, with its individual features and technology, and as hardware grew in capability and games grew in complexity, more and more time was spent wrestling with the technology behind the games. This lead to a phase in the 90's and 2000's where companies with successful products and who were particularly proud of their technical achievements would license their engines out to other studios to be used on other products. It was a natural progression and it allowed many game developers to stay focused on the important part of their jobs - the games.

2.1 Game Engines

Nowadays, we can find a lot of different types of engines, some of them are public and can be downloaded any time, and some others are private and can only be used by the studio or under regulated permissions.

2.1.1 Unreal Engine (W.14)

The Unreal Engine is a game engine developed by Epic Games, first showcased in the 1998 first-person shooter game Unreal. Although primarily developed for first-person shooters, it has been successfully used in a variety of other genres, including stealth, fighting games, MMORPGs, and other RPGs. With its code written in C++, Unreal Engine features a high degree of portability and is a tool used by many game developers today, with it being source-available. The most recent version is Unreal Engine 4, which was released in 2014.

<table>
<thead>
<tr>
<th>Strong Features:</th>
<th>Weak Features:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Powerful 3D rendering and lighting.</td>
<td>- Outdated networking system.</td>
</tr>
<tr>
<td>- Visual scripting (Blueprints).</td>
<td>- Bad social integration.</td>
</tr>
<tr>
<td>- Visual shaders editor.</td>
<td>- Not suitable for low-end devices and 2D.</td>
</tr>
<tr>
<td>- C++ scripting.</td>
<td></td>
</tr>
</tbody>
</table>
Sunyer Caldú, Guillem
BeEngine: 2D Game Engine

| - Good open world tools. | - Outdated Editor UI. |
| - Visual effects editor. | |
| - Access to the source code. | |

We are not going to use any of the Unreal Engine features in our project, since it’s a 3D focused engine, and features like visual scripting, or a visual shaders editor, are out of the scope.

Used in games like: Fortnite (2017), Bioshock, PUBG (2017)

### 2.1.2 Unity (W.15)

Unity is a cross-platform game engine developed by Unity Technologies, first announced and released in June 2005 at Apple Inc.’s Worldwide Developers Conference as an OS X-exclusive game engine. As of 2018, the engine has been extended to support 27 platforms. The engine can be used to create both three-dimensional and two-dimensional games as well as simulations for its many platforms. Several major versions of Unity have been released since its launch, with the latest stable version being Unity 2018.3.8.

#### Strong Features:
- Great support for either 2D and 3D.
- C# scripting.
- Clean and clear entity system.
- Visual effects editor.
- Visual scripting.
- Text rendering with Text Mesh Pro (W.3).
- Efficient file system.
- Clean editor UI.
- 2D physics with Box2D (W.4), and 3D physics with PhysX (W.5).
- Very nice 2D animation tools.

#### Weak Features:
- Outdated networking system.
- Outdated input system.
- Bad support for 2D tiling.
- The source code is private.

We will use a lot of interesting features from Unity. Their file system and file system watcher is a reliable method to keep file and resources consistency. Also, Unity uses Mono Project (W.6) to handle their scripting in C#, an intuitive language that’s faster to use than C++, while keeping its familiar syntax, improving it with a lot of useful features. The entity system used by unity, with game objects and components, it’s a very clean way to organize logic, while keeping it simple. The text rendering system named Text Mesh Pro provides very memory efficient text rendering, while also being very fast and customizable, using a GPU vector rendering technique (W.7). For 2D physics, it uses a highly popular physics library named Box2D, with great
Source is a 3D video game engine developed by Valve Corporation. It debuted as the successor to GoldSrc with Counter-Strike: Source in June 2004, followed shortly by Half-Life 2 in November, and has been in active development since. Source does not have a concise version numbering scheme; instead, it is designed in constant incremental updates. The successor, Source 2, was officially announced in March 2015, with the first game to use it being Dota 2, which was ported over from Source later that year.

Strong Features:
- Fully coded in C++ (including logic).
- Very efficient 3D rendering.
- Distance field text rendering (W.8).
- A lot of great games made with the engine.
- Access to the source code.

Weak Features:
- 3D only
- Complex to start using.

The text rendering techniques developed by Valve and applied to their engine it’s indeed a very interesting and valid option to use as main text rendering technique, and can work as an alternative to GPU vector rendering.


Godot is a 2D and 3D cross-platform compatible game engine released as open source software under the MIT license. It was initially developed for several companies in Latin America before its public release. The development environment runs on Linux, macOS, Windows, BSD and Haiku (both 32 and 64-bit) and can create games targeting PC, mobile and web platforms.

Strong Features:
- C# scripting.
- Open source
- Great 2D features.
- Multi-scene work.
- Nice 2D animation system.

Weak Features:
- Not very good compatibility while working in 2D and 3D at the same time.
- Missing 3D features.
- Messy entity organization.
- Visual scripting.

The multi-scene work support of Godot engine it’s a fantastic and necessary solution towards working with different people in the same project, allowing each of one to have its own scene, without overlapping the work on version control.

Used in games like: Daemon vs Demon

### 2.1.5 Amazon Lumberyard (W.18)

Amazon Lumberyard is a free cross-platform game engine developed by Amazon and based on the architecture of CryEngine, which was licensed from Crytek in 2015. Lumberyard launched on February 9, and as of March 2019, the software is currently in beta status and can be used to build games for Microsoft Windows, PlayStation 4, Xbox One, with limited support for iOS and Android and the support of Linux and Mac being planned for future releases.

#### Strong Features:
- Cascade prefab system.
- Great 3D world editing tools

#### Weak Features:
- Still in early development.
- Lua scripting.

The cascade prefab system it’s of tremendous help and a state-of-the-art feature on modern game engines that provides the ability to create prefabs from prefabs, allowing the modification of all the instances when the parent prefab is changed.

Used in games like: Star Citizen and The Grand Tour Game

### 2.1.6 Game Maker (W.19)

GameMaker Studio (formerly Animo until 1999, Game Maker until 2011, GameMaker until 2012, and GameMaker: Studio until 2017) is a cross-platform game engine developed by YoYo Games.

GameMaker accommodates the creation of cross-platform and multi-genre video games using a custom drag-and-drop visual programming language or a scripting language known as Game Maker Language, which can be used to develop more advanced games that could not be created just by using the drag and drop features.

#### Strong Features:
- Easy if the user doesn’t know how to code.
- Artist-friendly.
- Simplicity for 2D.

#### Weak Features:
- Limited for advanced users.
- Custom scripting language.
- Visual scripting.
- Exceptional 2D tiling support.
- Physics with Box2D (W.4)
- Fantastic 2D animation support.

Game Maker has a great simplicity that combined with powerful 2D tools, becomes the perfect engine to create 2D games, if you don’t want to code. The problem appears when the user wants more customization or more features, a field where it lacks a lot of dynamism.


2.2 Libraries and Technology

Inside a game engine, we can find a lot of different pieces with a lot of different functionalities and implementations. Depending on those implementations, and the interconnection between those pieces, the engine is going to behave in some way, and output certain results. Sometimes those pieces are made from scratch by the developer building the engine, or sometimes already created and tested code is used to improve development time and final results. Those collections of premade code are named libraries, and are a very important part of game development.

2.2.1 Box2D (W.4)

Box2D is a free open source 2-dimensional physics simulation engine written in C++ by Erin Catto and published under the zlib license. It has been used in Crayon Physics Deluxe, Limbo, Rolando, Incredibots, Angry Birds, Tiny Wings, Transformice, Happy Wheels, and many online Flash games, as well as iPhone, iPad and Android games using the Cocos2d or Moscrif game engine and Corona framework.

Strong Features:  
Weak Features:
BeEngine: 2D Game Engine

- Widely used.
- Great documentation.
- Great performance.

An easy to integrate, proven and well-documented library that makes implementing 2D physics feel like a child's game.

Used in games like: Angry Birds, Limbo

2.2.2 Project Mono (W.6)

Mono is a free and open-source project to create an Ecma standard-compliant .NET Framework-compatible software framework, including a C# compiler and a Common Language Runtime. Originally by Ximian, it was later acquired by Novell, and is now being led by Xamarin, a subsidiary of Microsoft and the .NET Foundation. Mono allows code to be written in multiple languages: components can be authored in C# that is a language with strong support for best engineering practices (your core libraries, and reusable components can be authored in it) and yet allow the flexibility of a scripting language like Python or C for code that you must quickly prototype or alter.

**Strong Features:**
- Already some game engines have adopted it (Unity, Godot, CryEngine).
- Brings great flexibility to scripting capabilities.
- Great performance.

**Weak Features:**
- Poor official documentation.
- Has some important bugs.
- Not debug friendly.

Mono acts like a double-edged sword while implementing it. As it provides almost non-existent official documentation, the process can become slow and painful, but once the implementation is right, it works very nicely and allows a very fluid scripting experience between C++ and C#.

2.2.3 Dear ImGUI (W.9)

Dear ImGui is a bloat-free graphical user interface library for C++. It outputs optimized vertex buffers that you can render anytime in your 3D-pipeline enabled application. It is fast, portable, renderer agnostic and self-contained (no external dependencies).
**Strong Features:**
- Very easy to set up.
- Very performant.
- Customizable.

**Weak Features:**
- Complex or custom UI is a bit messy to create.

Dear ImGui is designed to enable fast iterations and to empower programmers to create content creation tools and visualization / debug tools (as opposed to UI for the average end-user). It favors simplicity and productivity toward this goal, and lacks certain features normally found in more high-level libraries.

It’s particularly suited to integration in games engine (for tooling), real-time 3D applications, fullscreen applications, embedded applications, or any applications on consoles platforms where operating system features are non-standard.


**2.2.4 Vector art rendering on GPU (W.7)**

Vector representations are a resolution-independent means of specifying a shape. They have the advantage that at any scale, content can be displayed without tessellation or sampling artifacts. This is in stark contrast to a raster representation consisting of an array of color values. Raster images quickly show artifacts under scale or perspective mappings.
2.3 Conclusion

The objective of the research in the state of the art and the technologies that are currently being used, is to explore and learn which are the best tools and information at our side while creating the engine, to accomplish the best product possible.

As can be observed in point 2.1, the described engines are either, generic 2D and 3D engine with tons of features, very powerful 3D engines oriented to AAA development, or more simple engines, but that aim a more inexperienced user group. There’s never a middle point, a smaller engine with a known and useful scripting language, that is focused on experienced developers that want to create simpler games. With that said, it’s important to define the internal tools and techniques that our project is going to implement:

Box2D it’s a very widely used library that works very well and it’s very performant, so it’s an easy choice for the project.

Next, after searching the best solution for scripting, Mono Project came up the winner for its reliability and flexibility while adapting it to the engine, and it provides the user with the C# language, a very useful and established tool that is very familiar for C++ programmers, and it’s very fast to prototype with.
Another easy choice for the project is the Dear ImGui UI library that’s used for the engine editor since it’s very easy to implement, fast and reliable, and provides a lot of flexibility while writing tools.

Finally, for the text rendering, the vector art rendering technique gets out of the scope, since it’s too costly regarding time for this project, and there are other solutions that are not as performant, but that are faster to implement.
3. Project management

The keys for a successful project are great planning and time estimation accuracy. This section describes the thoughts about task deadlines and organization that it's followed by this project.

3.1 Procedure and tools

3.1.1 Gantt

To organize and distribute all the work through time, a Gantt diagram is used. A Gantt diagram shows the assigned time for every one of the tasks, and the dates in which those tasks should be finished. Also, it shows dependencies between tasks that need to be finished, before other ones can start. This tool allows the measurement of the speed in which the project is advancing, and helps comparing it to the original planning and adapt or change objectives in consequence. All of this is represented on the Gantt in F.3.1, and it's created using a free tool named GanttProject (W.10).
3.1.2 Github and version control

As this project is based on the creation of a software program, version control it’s an indispensable tool to help handle the large growing code base. Since the beginning of the development, a repository was created, and all the updates and new features were uploaded there to keep track of changes, and being able to return to older versions if problems occurred. Since the code base is only touched by one developer, all the code is being pushed on the master branch. The Github repository can be found here: (W.2).

3.2 Validation tools

As development goes, a lot of features are added in short periods of time, and it’s very important to assure a consistent working project at the end, that those features are tested and validated.

3.2.1 Definition of done

When the implementation of any of the tasks ends, it’s not directly finished, as it needs to be tested. With the big amount of code and different functionalities and case scenarios, it’s not trivial to test and find all the errors on the code, with only one person to do it.

Tests are produced at the end of every new feature implemented, to spot bugs, crashes or any malfunction that may occur. There are also general regular tests, to
prove that all the functionalities that are already completed are still working as expected.

As said before, it’s not easy to spot all the bugs for all the case scenarios To solve this problem, once development it’s over, the showcase games created with the engine will also serve as a final test barrier for the engine, since all the modules are going to be used in different scenarios and working at the same time, exposing any bugs that may still be present.

With all that in mind, it’s still expected to leave some unfound bugs unsolved at the end of the development, since it’s almost impossible to have a full bug-free project, and the available time is limited. The final objective is that the program is fully functional and does its job properly.

3.2.2 Validation of parts
Here are listed all the expected results of every important project part:

- **Part 1: File System and Resources Manager**
At the end of this part, we should be able to import files into the engine, and they should be properly managed and loaded as a resource. Also, if any change happens to the files loaded, the engine should react accordingly. Finally, all the resources should be properly mirrored in the library folder.

- **Part 2: Scripting**
The engine can load files and detect changes, so it should detect changes on the scripts used for the logic, and recompile them on the fly. Also, the play, stop and pause scripting logic should be working. Communication between C++ and C# is already robust.

- **Part 3: Entity System**
The user can create, delete, parent and child game objects easily in the inspector, and also their components such as the transformation component, the script component, or the sprite renderer component.

- **Part 4: Physics**
The physics system is working under the hood and the game objects physics bodies and colliders can interact with each other, and events are properly sent when collisions happen. Also, physics shapes should be editable.
- Part 5: In-game UI
The in-game UI is properly scaling to the different window sizes, and all the widgets (buttons, texts, scroll bars, etc.) are working as expected and firing events when necessary.

- Part 6: Games showcase
Here the engine is ready to create full games and deploy builds on Windows properly. To demonstrate that, two games are created, Pong (a smaller one), and Flappy Birds (a bigger one).

3.3 Swot analysis

<table>
<thead>
<tr>
<th>Strengths</th>
<th>Weaknesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>- I’ve already studied this topic previously, even if it was with less depth.</td>
<td>- A very long project for being developed with one person and the amount of time given.</td>
</tr>
<tr>
<td>- Already 4 years of experience in C++.</td>
<td>- A lot of poorly undocumented topics, like scripting with Mono Project (W.6).</td>
</tr>
<tr>
<td>- Capacity to try new things and take risks.</td>
<td>- Never had experience in real Game Engine projects, so the code could be not optimal enough.</td>
</tr>
<tr>
<td></td>
<td>- Cross-platform software with a lot of different platforms is very common in modern game engines, but very difficult to accomplish by a small project.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opportunities</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>- There is not a game engine that accomplishes all the features that this project wants to achieve.</td>
<td>- There’s already a lot of game engines that have been around for years, with a lot of people behind, that are very widely used and have a lot of tools.</td>
</tr>
<tr>
<td>- As it is a student’s project, there’s any possibility of financial risk, so all the risks can be taken, without consequences.</td>
<td>- With the limited time, it may come up very basic for creating big games.</td>
</tr>
</tbody>
</table>
3.4 Risks and contingency plans

Before starting a project, and more importantly if the project is big and over a large period of time, it’s key to identify all the problems that could appear during the course of the development, mark those possible problems as risks, and find contingency solutions to apply if necessary. This section analyzes all the risks taken into account in this project, and it’s possible solutions.

3.4.1 Risks during development

- **Lack of time**: time estimations are a really difficult task to pull off properly at the start and during long and complex projects. It’s very easy to underestimate the work that has to be done, and in consequence very easy to get wrong estimations on the tasks. In addition to that, this project duration is way over the time specified for a final degree project, due to its complexity and workload.

To solve this problem, I’ve:
- Added a time modification to all the task estimations, always adding time to the planning, to avoid or minimize the time deviation.
- Started working on the practical part way before the theoretical start of the project.
- Milestone settling, revision, and modification depending on the speed of the project.
- The project estimations are made to have time and the end for polish and cleanup.
- As the project is open source, there’s always the possibility of external users to expand the content in the engine in case there are some features missing.

If the problem can’t be solved:
- The estimations include two full games to showcase the engine at the end of the development. One or both could be cut out in case of lack of time, allowing the engine to still be feature complete.

- **Lack of knowledge**: working with new technologies, in this case in the field of coding, there’s always the possibility that there is not a clear or possible way to continue working, and because of time restraints, the approach has to be changed or totally discontinued.

To solve this problem, I’ve:
- Chosen a field that I’ve already worked at, allowing a faster approach to known tasks, and giving a better idea in how to accomplish unknown ones.
• A lot of prior investigation and exploration about the current state of the art.

If the problem can’t be solved:
• Contemplate different technologies that solve the same problem, in the possibility that some are too complex or out of the scope.

3.4.2 Risk of the tasks

In the F.3.2 we have different tasks that need to be done to finish the project, and here are listed the ones with higher risks, and in consequence, the ones with higher time estimations and higher time deviation potential.

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Estimated time (days)</th>
<th>Potential deviation</th>
<th>Planned time (with deviation) (days)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T.1 PROJECT SETUP</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T.1.1 Repository and base code</td>
<td>7</td>
<td>None</td>
<td>7</td>
</tr>
<tr>
<td>T.1.2 Base engine modules</td>
<td>4</td>
<td>None</td>
<td>4</td>
</tr>
<tr>
<td>T.1.3 Editor UI and docking</td>
<td>5</td>
<td>None</td>
<td>5</td>
</tr>
<tr>
<td><strong>T.2 SPECIFIC ENGINE MODULES</strong></td>
<td>22</td>
<td>Medium</td>
<td>30</td>
</tr>
<tr>
<td>T.2.1 File System</td>
<td>4</td>
<td>None</td>
<td>4</td>
</tr>
<tr>
<td>T.2.2 Project Manager</td>
<td>4</td>
<td>None</td>
<td>4</td>
</tr>
<tr>
<td>T.2.3 Shaders Manager</td>
<td>5</td>
<td>None</td>
<td>5</td>
</tr>
<tr>
<td>T.2.4 Profiler</td>
<td>2</td>
<td>None</td>
<td>2</td>
</tr>
<tr>
<td>T.2.5 Resources Manager</td>
<td>7</td>
<td>Very High</td>
<td>15</td>
</tr>
<tr>
<td><strong>T.3 SCRIPTING AND MONO</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T.3.1 Mono Integration</td>
<td>3</td>
<td>Very High</td>
<td>7</td>
</tr>
<tr>
<td>T.3.2 C# File watcher</td>
<td>5</td>
<td>Very High</td>
<td>10</td>
</tr>
<tr>
<td>T.3.3 C# Compiler</td>
<td>2</td>
<td>Very High</td>
<td>4</td>
</tr>
<tr>
<td>T.3.4 C# visual studio solution controller</td>
<td>3</td>
<td>Very High</td>
<td>5</td>
</tr>
<tr>
<td>T.3.5 Class and garbage management</td>
<td>4</td>
<td>Very High</td>
<td>8</td>
</tr>
<tr>
<td>T.3.6 Bridge between C# and C++ classes</td>
<td>10</td>
<td>Very High</td>
<td>20</td>
</tr>
<tr>
<td>Component</td>
<td>T.4 ENTITY COMPONENT SYSTEM</td>
<td>16</td>
<td>None</td>
</tr>
<tr>
<td>-----------</td>
<td>----------------------------</td>
<td>----</td>
<td>------</td>
</tr>
<tr>
<td>T.4.1 Game Objects</td>
<td>6</td>
<td>None</td>
<td>6</td>
</tr>
<tr>
<td>T.4.2 Components</td>
<td>6</td>
<td>None</td>
<td>6</td>
</tr>
<tr>
<td>T.4.3 Hierarchy</td>
<td>4</td>
<td>None</td>
<td>4</td>
</tr>
<tr>
<td>T.4.4 Inspector</td>
<td>4</td>
<td>None</td>
<td>4</td>
</tr>
<tr>
<td>T.4.5 Game Object serialization</td>
<td>3</td>
<td>None</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Render</th>
<th>T.5 RENDERING</th>
<th>14</th>
<th>None</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>T.5.1 Line rendering</td>
<td>3</td>
<td>None</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>T.5.2 Shapes rendering</td>
<td>3</td>
<td>None</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>T.5.3 Sprite rendering</td>
<td>2</td>
<td>None</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>T.5.4 Gizmos</td>
<td>6</td>
<td>None</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Physics</th>
<th>T.6 PHYSICS</th>
<th>16</th>
<th>Low</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>T.6.1 Box2D implementation</td>
<td>2</td>
<td>None</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>T.6.2 Physics Body implementation</td>
<td>3</td>
<td>None</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>T.6.3 Physics Shape implementation</td>
<td>3</td>
<td>None</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>T.6.4 Simulation and scripting logic</td>
<td>3</td>
<td>None</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>T.6.5 Shape editing</td>
<td>5</td>
<td>Medium</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UI</th>
<th>T.7 INGAME UI</th>
<th>15</th>
<th>Low</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>T.7.1 Canvas implementation</td>
<td>3</td>
<td>None</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>T.7.2 UI Widgets</td>
<td>16</td>
<td>None</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>T.7.3 Text Rendering</td>
<td>12</td>
<td>Medium</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Build</th>
<th>T.8 BUILD SYSTEM</th>
<th>18</th>
<th>None</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>T.8.1 Build creator</td>
<td>3</td>
<td>None</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>T.8.2 Engine Game Mode</td>
<td>15</td>
<td>None</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>
### T.9 SHOWCASE GAMES

<table>
<thead>
<tr>
<th></th>
<th>25</th>
<th>None</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>T.9.1 Pong</td>
<td>10</td>
<td>None</td>
<td>10</td>
</tr>
<tr>
<td>T.9.2 Flappy Birds</td>
<td>15</td>
<td>None</td>
<td>15</td>
</tr>
</tbody>
</table>

#### Added deviation

- 125% +
- 150% +
- 200% +

**F.3.2 - Deviation table used in the project.**

- **T.2.5 - Resource Manager - Risk: High**

One of the most important parts, and usually assumed or overviewed, it’s the file system of a game engine. It ensures consistency between files and resources, marks loading times, portability, and it’s the base of a lot of the engine’s functionality. There are a lot of examples of how Resource Managers work in the outside, simply by looking at how an engine like Unity or Unreal does it. The problem is that there is no real information on the topic in how to implement it in code, and how to interconnect the system properly with the rest of the engine. Because it’s a crucial part, and it has to be done from scratch, it has a high risk of deviation.

- **T.3 - Scripting and Mono - Risk: High**

The scripting system it’s also a key part of a game engine functionality. In this case, we use the Mono Project library for scripting in C#. It is a library with a very nice functionality but it lacks a lot of documentation and examples. In addition, I have no prior experience implementing scripting on a game engine.

- **T.6.5 - Physics shape editing - Risk: Medium:**

A Box2D shape needs certain requirements to create a physics shape: it doesn’t allow concave shapes, vertices that are too close, and very small shapes. To solve this, while also having a smooth editor shape editing experience, certain calculations, triangulation, and tricks have to be made. This is time costly because it’s not documented in the library, and it has to go with trial and error and external sources.

- **T.7.3 - Text rendering - Risk: Medium:**

Text rendering it’s also a task that looks easier than it really is. There are a lot of documented techniques available for rendering text using OpenGL, but they are
complex and only explained on paper. Depending on the methodology chosen for
the implementation it will be easier or more difficult, so in front of the uncertainty,
it’s a medium risk task.

3.5 Costs analysis

For the costs estimates, we are going to assume that the average work per day is 4h
and that the cost per hour is 9€.

<table>
<thead>
<tr>
<th>Type</th>
<th>Subject</th>
<th>Price</th>
<th>Type</th>
<th>Years of amortization</th>
<th>Total price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct costs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Personal</td>
<td>Salary / Time</td>
<td>9.720,00 €</td>
<td>Total</td>
<td></td>
<td>9.720,00 €</td>
</tr>
<tr>
<td>Equipment</td>
<td>Desk</td>
<td>150,00 €</td>
<td>Amortization</td>
<td>5</td>
<td>22,50 €</td>
</tr>
<tr>
<td></td>
<td>Chair</td>
<td>100,00 €</td>
<td>Amortization</td>
<td>5</td>
<td>15,00 €</td>
</tr>
<tr>
<td></td>
<td>Computer</td>
<td>1.400,00 €</td>
<td>Amortization</td>
<td>5</td>
<td>210,00 €</td>
</tr>
<tr>
<td></td>
<td>Screen</td>
<td>150,00 €</td>
<td>Amortization</td>
<td>2</td>
<td>56,25 €</td>
</tr>
<tr>
<td></td>
<td>Mouse</td>
<td>60,00 €</td>
<td>Amortization</td>
<td>2</td>
<td>22,50 €</td>
</tr>
<tr>
<td></td>
<td>Keyboard</td>
<td>100,00 €</td>
<td>Amortization</td>
<td>2</td>
<td>37,50 €</td>
</tr>
<tr>
<td>Consumables</td>
<td>Notebook</td>
<td>5,00 €</td>
<td>Unique</td>
<td></td>
<td>5,00 €</td>
</tr>
<tr>
<td></td>
<td>Pencils</td>
<td>7,00 €</td>
<td>Unique</td>
<td></td>
<td>7,00 €</td>
</tr>
<tr>
<td>Software</td>
<td>Visual Studio</td>
<td>0,00 €</td>
<td>Monthly</td>
<td></td>
<td>0,00 €</td>
</tr>
<tr>
<td></td>
<td>GitHub</td>
<td>0,00 €</td>
<td>Monthly</td>
<td></td>
<td>0,00 €</td>
</tr>
<tr>
<td></td>
<td>Trello</td>
<td>0,00 €</td>
<td>Monthly</td>
<td></td>
<td>0,00 €</td>
</tr>
<tr>
<td>Indirect costs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maintenance</td>
<td>Electricity</td>
<td>20,00 €</td>
<td>Monthly</td>
<td></td>
<td>180,00 €</td>
</tr>
<tr>
<td></td>
<td>Water</td>
<td>12,00 €</td>
<td>Monthly</td>
<td></td>
<td>108,00 €</td>
</tr>
<tr>
<td></td>
<td>Food</td>
<td>30,00 €</td>
<td>Monthly</td>
<td></td>
<td>270,00 €</td>
</tr>
<tr>
<td>Total:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10.653,75€</td>
</tr>
</tbody>
</table>

Project duration: 9 Months

F.3.3 - Costs.

The first thing that can be observed on F.3.3, is that the most impactful cost is the
salary (or invested time on the project), as the project is about software
development, and no further costs are necessary on materials or equipment. Next, it’s a student’s project, all the needed software is totally free and available to use. The first necessity costs like water or food are calculated approximately using the base cost for every month and multiplying by the number of months. Same goes for the equipment, where the total price comes from the amortization time of the project duration.
4. Methodology

To build this project, the agile methodology named SCRUM has been used to control the tasks milestones and keep track of the progress in every moment of the development. It is designed for teams of three to nine members, who break their work into actions that can be completed within timeboxed iterations, no longer than one month and most commonly two weeks, then track progress and re-plan in 15-minute time-boxed stand-up meetings.

As this is a project developed only by one individual, the group part of the method does not make much sense, but it’s still very useful to divide the work on sprints and check the progress once every week or two.

4.1 SCRUM

F.4.1 - SCRUM loop.

In the Scrum methodology, there’s always a loop that keeps iterating during the development of the software, and in this loop, several steps are taken, as we can observe in F.4.1.

4.1.1 Sprint

This loop is named sprint (or iteration) and it’s is the basic unit of development. The sprint is a timeboxed effort, that is restricted to a specific duration that is fixed in advance for each sprint and is normally between one week and one month, with two weeks being the most common.
4.1.2 Product Backlog

The product backlog is a model of work to be done and contains an ordered list of product requirements that a scrum team maintains for a product.

4.1.3 Sprint Backlog

Each sprint starts with a sprint planning event that aims to define a sprint backlog, identify the work for the sprint, and make an estimated forecast for the sprint goal. Each sprint ends with a sprint review and sprint retrospective, that reviews progress to show to stakeholders and identify lessons and improvements for the next sprints.

4.1.4 Sprint Review and Retrospective

At the end of a sprint, two events are held: the sprint review and the sprint retrospective. At the sprint review, the work that is already completed and the planned work that was not finished is evaluated. Then, at the sprint retrospective, the past sprint is reflected and the next improvement actions are agreed on.

4.2 Tools used for the tracking of the project

To keep track of the project tasks and the overall process, a tool named Trello (W.12) is used, allowing an easy, fast and efficient control of the tasks, while also providing the element to follow the Scrum methodology.

F.4.2 - Trello.
The F.4.2 shows the interface of the Trello web application, and with a first glance, it’s very easy to see the way in which cards (or tasks) are organized in the different columns, depending on its state. The different columns named: “Deadlines”, “To do (or current sprint)”, “In progress”, “Ready to review” and “Reviewed”, allows for a very clear and fast overview of the state of the project.

Who is in charge of the task
Name of the task
Deadline
Task panel
Task group
Task description
Sub tasks

F.4.3 - Trello task.

In Trello, as we can see in F.4.3, every task has a lot of features to help the user organize and keep things clean. First of all, we have the name of the task and it’s the description, with the panel (or column) where the task is located, next we have the group in which this task falls in (Ex: Code, Art, Design, etc.), and the deadline marked for this specific task. A very useful feature it’s to assign a task to a user, so everybody can easily see which tasks have assigned. Finally, we can define sub-tasks in the form of checkboxes at the bottom.
5. Engine development

At this point, all the necessary initial research is finished, and it’s time to start with the actual engine implementation. This section goes through the preparation of the first codebase skeleton from which the engine is going to start growing, and also through some of the most interesting or challenging core modules that power the core engine functionalities.

5.1 Skeleton

To start with any software application, a base general code structure is needed to ensure that the project grows in a way that is consistent, scalable and organized. In a C++ environment, there are a lot of ways to accomplish this objective, and in this case, the following described method is used.

As can be seen at F.5.1 the execution starts on a base object named Application that contains all the necessary logic to control the workflow of the whole engine. This is done with a subset of objects, named modules, that are stored inside the Application object, where each one of them has a distinct core and differentiated function. During the execution, the Application object has the objective of calling the following base functions from each one of the different modules:

- Called only one time at the beginning of the execution, when the software is started:
  - **Awake()**: during this function call, the module has to set-up all the necessary elements to start working.
  - **Start()**: here the module retrieves necessary information from other modules since they are already set-up.

- Called once every frame:
  - **PreUpdate()**: update the module logic that is necessary for the main update loop.
  - **Update()**: update main module logic.

![Diagram showing the update flow of the engine modules](image)
- **PostUpdate()**: update the module logic before the update loop is finished.

- Called only once at the end of the execution, when the software is closed:
  - **CleanUp()**: de-init all the modules from the software, and frees all the used memory.

The different modules that are contained on our skeleton (F5.2) are the most basic ones and need to perform basic logic on a software application such as creating the application window, managing input, create-load-destroy files, etc. In this case, the base modules are the following.

- **Module Window**
  
  This module manages all the low-level creation of the application window, its size, the title and the icon that are displayed.

- **Module Input**
  
  Controls all the user input that is happening on the application, and keeps the state of those inputs so the application can access them.

- **Module Renderer**
  
  Initiates all the necessary environment to start with the rendering, and abstracts all the rendering functions from OpenGL, so they are easier to use.

- **Module Scene Renderer**
  
  Keeps track of all the different types of rendering pipelines, and merges them all to create a final render for the game scene.

- **Module Event**
  
  On a big project with a lot of functionalities, it’s very useful to have events. Any object of the engine can subscribe to an event, and then when that event is called, all the subscribed objects will...
be noticed. This module makes sure those events keep the same structure and that are sent properly.

- **Module Json**

A lot of times is very useful for a game engine to save some information on a file, and then load this file and this information at any given time. This is called serialization, and in this engine, it’s all done in a format named Json. This module manages all the utilities to make this process easier.

- **Module Editor**

Manages all the User Interface that is going to be used to create the engine editor and it’s different windows, like the Hierarchy, the Explorer or the Inspector.

- **Module FileSystem**

Has all the necessary functions for the engine to interact with the Operative System file system, to allow the creation, modification or movement of files from the engine.

- **Module Camera**

Controls all the cameras that the engine uses to represent the scene point of view, and the basic controls for the editor camera.

### 5.1.1 External libraries

A library is a collection of implementations of behavior, written in terms of a language, that has a well-defined interface by which the behavior is invoked. For instance, people who want to write a higher level program can use a library to make system calls instead of implementing those system calls over and over again. With the objective of speeding development, a subset of well-established libraries is going to be used in this software:

- **SDL**: library designed to provide low-level access to audio, keyboard, mouse, joystick, and graphics.
- **ImGui**: library used to create immediate mode user interfaces.
- **Devil**: library that helps with the save and load of 2D images.
- **MathGeoLib**: mathematical library to help with complex operations, vectors and geometry.
- **Glew**: helper library for OpenGL.
- **Box2D**: a 2D physics library.
- **MapBox Triangulation**: library that provides easy and fast triangulation for complex polygonal shapes.
- **Tiny FileDialogs**: library that allows for easy windows-like dialogs for file selection.
- **FreeType**: a font engine that is designed to produce high-quality output glyph images.
- **FMod**: sound effect and audio engine for videogames.
- **Mono**: open source implementation of Microsoft’s .NET Framework that allows bridging C# and C++ code.

### 5.1.2 Expansion

Once the skeleton is set-up and ready to use, from there, the engine starts to grow in different directions, incrementing its functionalities and codebase. A good base it’s important to ensure that this evolution is fast, intuitive, and keeps the overall project organized and coherent. The application skeleton result can be seen at F.5.3

The next points of this project are going to go through the development of the structure and the process of some of the most important modules of the engine, how they were created, and which were the problems found during the process.

![F.5.3](image-url) - Game engine at skeleton state, rendering a simple triangle.
5.2 Editor

The editor is the part of the game engine that is visible to the end user and allows for interaction with the application. A game engine editor has to be intuitive and easy to use, without losing any functionality, to enable fast learning and workflow with the software (F.5.4).

Since in a game engine, several tools in different windows can be needed on the screen showing at the same time, a clear organization strategy has to be present to keep the application screen optimized. A very nice solution to allow this is the use of dockable windows, which are already used in a high variety of software and other game engines. A dockable window can exist in a floating state or be attached to the main application window, or even be collapsed into other windows, keeping easy access, while being hidden.

F.5.4 - Unity editor using docking windows.

Also, as there are a lot of different types of workflows and preferences depending on who is using the engine, the layout of the docking needs to be able to be saved. In the case of BeEngine engine, the docking layout is saved using a custom file that is stored on the engine folder and saves all the custom and default layouts created by the users (F.5.5).
5.2.1 Editor windows

To organize and distribute the editor UI, the different functionalities of the engine are divided into different windows. Here are described and explained some of the most important ones in the engine.

- **Explorer:**

  The function of the explorer is to be able to manage the assets that belong to your project. As we can see on F.5.6, the left panel of the browser shows the folder structure of the project as a hierarchical list. When a folder is selected from the list by clicking, its contents will be shown in the panel to the right. You can click the small triangle to expand or collapse the folder, displaying any nested folders it contains. The individual assets are shown in the right-hand panel with extensions that indicate their type (script, material, sub-folder, etc). At any time the user can import a new asset, create folders, rename assets, and delete them.

  All the information that appears on the assets folder is a direct and real-time mirror of the real folders and files stored in the user hard drive, meaning that if a user modifies any of the folders or files within the asset folder using the operative system, the engine will automatically update those changes on the engine.

  This is an indispensable and very useful feature, since any change in an image file, script, or any other filetype is modified from external software, it is automatically re-imported and ready to use without the user needing to update the file.
- Scene:

The Scene view is the interactive view of the world that is being created. It is used to select and position scenery, characters, cameras, and all other types of GameObject. It allows the user to view the current progress of the game from any position, in real time, at any moment, without needing to move any in-game cameras. It also shows some important gizmos that facilitate the use of certain tools like the transform gizmo or the physics editor gizmo (F.5.7).
- Hierarchy:

The Hierarchy window (F.5.8) contains a list of every GameObject in the current Scene. As objects are added and removed in the Scene, they will appear and disappear from the Hierarchy as well. It provides a way to observe the general state of the game scene, while also serve as a selection tool, to change GameObject properties. In addition, a GameObject can be dragged on top of another GameObject to create a parent-child connection.

- Inspector:

The Inspector window (F.5.8) displays detailed information about the currently selected GameObject, like the name and its current state, and also all attached components and their properties, allowing the user to modify the functionality of any GameObject in the Scene (F.5.7).

F.5.8 - Inspector and Hierarchy windows.
- **Console:**

The Console Window (F.5.9) shows messages generated by BeEngine. It allows fast communication between the engine and the users in the case it wants to transmit some information, an error occurs, script compilation errors, etc. It also allows filtering between the different type of messages (Messages, Warnings or Errors), to show only the desired information.

In addition to that, the user can print their own messages using scripting with the BeEngine C# API, using the command Debug.Log(“message”), as a debug utility.

![F.5.9 - Console window.](image-url)
5.3 Resources

In the process of video game creation, there is a large variety of different files that can be used to craft the different parts of the product: images, music, fonts, scripts, scenes, shaders, etc. In a game engine, it’s very important that those files are recognized and loaded accordingly in a way that they can be easily used in the editor.

As there are files of a large variety of sizes and types, they can take an undefined amount of time to load into the engine. This is important to be taken into account because if files are very big, this can create an important bottleneck. Not addressing this could create problems such as large loading times, engine freezing, or crashes. A way to avoid that is using the Assets / Library methodology and meta files.

5.3.1 Meta files

When the user loads an Asset such as a texture in the Assets folder, BeEngine will first detect that a new file has been added, and assign it a new unique ID. This ID is used internally by the engine to refer to the asset, allowing it to be, moved, renamed or reimported without references to the asset breaking, since it will always be identified by its unique ID. The ID assigned to each Asset is stored inside the .meta file which BeEngine creates alongside the asset file itself. This .meta file must stay with the asset file it relates to.

F.5.10 - Asset folder representation on Windows and on BeEngine.

In the image above F.5.10 can be seen that .meta files are listed in the file system for each asset and folder created within the Assets folder. These are not visible in BeEngine Explorer Window because the user does not need to know they exist, they are only used internally by the engine. If an asset loses its meta file (for example, if the asset is moved or renamed outside of the engine, without moving/renaming the corresponding .meta file), any reference to that asset will be broken. BeEngine
would generate a new .meta file for the moved/renamed asset as if it were a brand new asset, and delete the old “orphaned” .meta file.

5.3.2 Asset / Library

The engine reads and processes any files that you add to the Assets folder, converting the contents of the file to internal game-ready versions of the data. The actual asset files remain unmodified, and the processed and converted versions of the data are stored in the project’s Library folder.

Using internal formats for assets allows having game-ready versions of the assets, ready to use at runtime in the editor while keeping your unmodified source files in the assets folder so that the user can quickly edit them and have the changes automatically picked up by the editor. For example, the Photoshop file format is convenient to work with and can be saved directly into your Assets folder, but hardware such as mobile devices and PC graphics cards can’t accept that format directly to render as textures. All the data for BeEngine internal representation of the is stored in the Library folder, as seen at F5.11, which can be thought of as similar to a cache folder. As a user, you should never have to alter the Library folder manually.

F.5.11 - Library textures folder with the compressed .dds images.

5.3.3 File watcher and Asset / Library integrity

To ensure the integrity of our assets, it’s important to check if the resources’ references to the actual files are correct, and that they have all the meta files properly set-up. But those checks cannot be done continuously in runtime since they are computationally expensive and take some time. Instead, they should be done only on specific cases: when the engine starts and anytime a file is modified from outside the engine. When the engine is doing some of those checks, progress bars are shown at the editor (F.5.12).
F.5.12 - Progress bars showing when the engine is checking and loading assets.

For the engine to know when an external file has changed, it needs a File Watcher that sends events every time a change to the assets folder has been made. When BeEngine detects a file change, it waits 2 to 3 seconds to ensure that no other files have been changed, and then it checks the integrity of the changed file, and also from the whole folder where it is contained. The function that checks if any folder needs to be checked can be seen at F.5.13.

F.5.13 - Code showing the procedure taken when files are changed.
5.4 Entity Component System

For the game logic architecture, BeEngine uses the pattern named Entity Component System. It follows the composition over inheritance principle that allows greater flexibility in defining entities where every object in a game’s scene is an entity (e.g. enemies, bullets, vehicles, etc.). Every entity consists of one or more components which add behavior or functionality as can be seen in F.5.14. Therefore, the behavior of an entity can be changed at runtime by adding or removing components. This eliminates the ambiguity problems of deep and wide inheritance hierarchies that are difficult to understand, maintain and extend.

5.4.1 GameObjects and Components

In the BeEngine context, entities are named GameObjects, and every GameObject acts as a container for an infinite number of Components. By default, all GameObjects automatically have a Transform Component. This is because the Transform dictates where the GameObject is located, and how it is rotated and scaled. Using components allows for the particular GameObjects to have just the desired functionalities, without constraining or affecting the other GameObjects.

5.4.2 Prefabs

A critical point while creating the GameObject system of a game engine is the ability to abstract itself so it’s able to duplicate or serialize itself. This is accomplished with Prefabs. A prefab system allows the user to create, configure, and store a complete GameObject with all its components, property values, and child GameObjects, all as a reusable Asset. The Prefab Asset acts as a template from which you can create new Prefab instances in the Scene. Saving a prefab is as
simple as choosing the game object that we want to abstract, and save it as Prefab (F.5.15).

The power of a Prefab object also resides in the ability to change the parent prefab from where the instance has been made, so all the instances change with it. It can also be used on the game logic from the scripting language, spawning new instances of any GameObject prefab on demand. For example, on a shooter game, every bullet that a weapon spawns can be a prefab previously saved, that it’s instantiated from logic every time the player presses the shoot key.

5.4.3 Scenes

The Scenes contain the environments and menus of your game. Think of each unique Scene file as a unique level. In each Scene, you place your environments, obstacles, and decorations, essentially designing and building your game in pieces. In the engine, every GameObject has to be stored inside a Scene, and it can be saved or loaded at any moment, even during runtime. They help to divide the game into different, smaller parts that can be loaded on demand, improving projects overall organization, while also keeping the simultaneous object count much lower, thus improving performance.
5.5 Rendering

Rendering in BeEngine is the process in which all the geometry of our scene is shown into the appropriate cameras. In this case, the engine uses OpenGL 4.4 to take advantage of shaders. Shaders are little programs that rest on the GPU and have the function of running for each specific section of the graphics pipeline. In a basic sense, shaders are nothing more than programs transforming inputs to outputs. Shaders are also very isolated in that they’re not allowed to communicate with each other; the only communication they have is via their inputs and outputs.

The way in which the rendering pipeline of the engine is organized is by dividing the different types of rendering methods and shader code, and then print them all on the camera in a specified order.

5.5.1 Debug lines and Gizmos

For the rendering of debug objects like debug lines and gizmos, every shape is organized on different renderers. To send these shapes to the GPU, all the vertices are stored on the same buffer with its color and position and then sent with one draw call to the shader, to speed draw time avoiding having to iterate over every line.

Then, to accomplish the rendering of gizmos, it’s just a matter of combining different types of shapes like rectangles, triangles, and lines, and changing its position and color depending on the input of the user. The results can be seen at F.5.17 and F.5.18.
5.5.2 Scene renderers

For the rendering of scene objects like sprites or text, another type of rendering technique is necessary. As they can have transparency on the alpha channel, the order of rendering is very important, since OpenGL does not take care of providing a correct alpha representation when using depth testing. Also, the vertices of this geometry are going to be created only one time at the start of the engine execution and then reused using VBO and VAO objects, as they all use the same shape (a rectangle), and the only thing that changes is the texture rendered on top.

In the case of BeEngine, the sprites and the text are ordered from bottom to top following the order on the hierarchy, while also providing different layers that can be changed on the component properties. With all those variables, the engine has to decide the order in which every element on the scene has to be rendered, as can be observed at F.5.19.

F.5.19 - Example showing how the sprites change their drawing depth position.

5.5.3 Camera rendering

Since the scene can contain several cameras, as can be seen on F.5.20, all the rendering has to be performed on each one of them. To achieve this, the easiest way is to render every camera view of the scene on different frame buffers that then can be rendered as different textures. This also allows the engine to choose which things are rendered in every camera. A very clear case of that can be seen is the Scene Window and the Editor Window, where both are shown on the screen at the same time showing the same scene, but one contains the gizmos and the other does not (F.5.21).
F.5.20 - Piece of code showing the rendering loop over the different cameras.

F.5.21 - Example showing the rendering differences between the scene window (left), and the game window (right).
5.6 Scripting

The best way to stay competitive in the race for bigger and better games and game engines is to keep the engine as flexible, expandable, and robust as possible. An internal scripting language allows the developer to create a separate, crash-proof environment inside the game engine. This protected virtual machine executes the complex and frequently changing gameplay code, protected from the "real" machine running the game engine. As can be observed at F.5.22, partitioning the code in this way, the complexity of the engine core is significantly reduced, resulting in fewer bugs and a more robust game. And since a language system is far more flexible than a collection of "canned" effects, the engine will be able to do more interesting things.

Using a scripting language allows the engine programmers to focus on what is important to them - refining and optimizing the core technology of the game - while the game designers can handle the gameplay details. If the language is simple and well-designed, non-programmers can implement their designs directly in the script language without endangering the core engine code or involving the engine programmers. And since programmer time on a project is usually limited, recruiting designers as scriptwriters allows more of the original design to be realized, resulting in a more interesting final game.

Since it’s important that the scripting language is fast and easy to use, BeEngine uses C#, instead of the same language in which the core engine is written, C++. C# is a far more productive language, it has a more beautiful syntax, better libraries, and frameworks, and fewer headaches and pitfalls. Also, it has garbage collection, so there are way fewer troubles with memory management. As for speed, C# is very optimized. It’s not as fast as C++, but what it’s lost on speed, it’s gained on portability and easier development.

To allow the bridge between C++ code and C# code, a library mentioned previously it’s used: Mono.

F.5.22 Layered approach provided by scripting.
5.6.1 Visual Studio Project

While implementing scripting, it’s also important to ask the question of which text editor is going to be the default used to modify the scripts of the game engine. This is important since a good integration of the engine and the IDE can improve the workflow dramatically. In the case of BeEngine, Visual Studio is used as the default scripting editor, since it offers a very easy to modify and understandable project solution. The project solution of Visual Studio es entirely made using XML markup, so the engine can load it at any time and change its properties. This is very useful when, for example, a script is removed or updated inside the engine, and the scripts that are shown on the Visual Studio editor need to be updated too. Also, when some external library reference is added to the engine, it can also be added automatically to the project solution.

The strategy here is to create manually a Visual Studio project, suited for C#, and then clean everything from it. From there, the project can be stored inside an engine folder to use it as a template that can be copied and pasted automatically into the project folder every time a new engine project is created (F.5.23). From there the engine can load the solution and modify it at its will. This offers a very smooth experience since the scripts are updated automatically and the user can enjoy a very complete external editor like Visual Studio, or any other IDE that loads Visual Studio projects.

Finally, since the engine contains a File Watcher, if at any moment the scripts of the project are changed from within the external editor, a file changed event is raised internally, and the scripts are automatically recompiled.

F.5.23 - Folder of the Visual Studio solution, and the actual project inside the editor
5.6.2 Mono

Mono allows us to do a huge variety of things once it is properly set up and running. To start working with C# code inside the library, it needs to know which codebase needs to be used. To feed mono with this information, the scripts information needs to be delivered as a precompiled library or .dll, and then load it in the following way: F.5.24.

```
if (!loaded)
{
    char* assembly_data = nullptr;
    uint assembly_data_size = 0;
    if (App->file_system->FileRead(path, assembly_data, assembly_data_size))
    {
        MonoImageOpenStatus status;
        image = mono_image_open_from_data_with_name
            (assembly_data, assembly_data_size, true, &status, false, path.c_str());
        if (status == MONO_IMAGE_OK || image != nullptr)
        {
            assembly = mono_assembly_load_from_full(image, path.c_str(), &status, false);
            if (assembly != nullptr)
            {
                loaded = true;
            }
        }
        RELEASE_ARRAY(assembly_data);
    }
}
```

F.5.24 - Code snippet from the assembly loading function using mono.

This process can be repeated an infinite number of times to load as many assemblies as wanted. Once all the code is loaded to mono, the desired classes can be extracted and saved for each assembly, to start working with them. Extracting classes (F.5.25) allows the engine to get class properties like variables, properties, namespaces, names, parent classes, etc.

```
MonoClass* cl = nullptr;
cl = mono_class_from_name_case(image, class_namespace, class_name);
if (cl != nullptr)
{
    class_returned = ScriptingClass(cl);
    ret = true;
}
```

F.5.25 - Code showing how to extract a class given an assembly image, the class namespace, and the class name.
From there, it’s very straightforward to create instances of any class that’s needed. There are two types of class instances: strong reference and weak reference. Creating a strong reference will avoid the C# garbage collector to delete or move the memory of the object that’s been instantiated; that’s useful where the engine wants full control of the creation and destruction of the object. From the other side, a weak reference won’t prevent the C# garbage collector to delete it, but will maintain the object in memory until it’s no longer needed; that’s useful, for example, when creating temporary variables (F.5.26).

F.5.26 In this code, a C# instance of a class is created with: a weak reference or a strong one.

Finally, once an instance it’s created, all the necessary variables are ready to start calling functions from the loaded classes (F.5.27).

F.5.27 Here, C++ code is calling a C# function is called from a class instance given the function name.
5.6.3 Script compiling

To compile the user scripts and use them with mono, the engine detects any change on the created scripting solution. If a change is detected, since the library does not allow the removal of assemblies once they are loaded, all the mono environment needs to be unloaded and loaded again, but this time with the new script changes. To compile the changed code and generate a .dll from it, the scripts are fed into a custom C# class instance (F.5.28). This class instance, with the help if the internal mono and C# compilers and with all the external references set up, outputs a .dll file if the compilation is successful, and the compilation errors if it fails. This newly generated dll is the one that is loaded again as an assembly.

F.5.28 Here, C# code is compiling some given scripts and returning the compiled .dll.

5.6.4. Game Loop

The scripts that are updated on the game loop and make use of the BeEngine object class, all need to inherit from the component ComponentScript class. This class contains the virtual functions for Awake, Start, Update and OnDestroy that can be overridden and are automatically called once the engine is in play mode, as seen at F.5.29.

F.5.29 BeEngine script with its default functions.

The Awake() and Start() function are called at the beginning of the execution, or when the object is created and the execution is already running. Then, the Update() function is called every frame, and the OnDestroy() function is called when the GameObject that contains the script gets destroyed.

This class allows the user to have all the flexibility that C# offers, while still proving with the necessary tools to interact with the engine.
Since the entity-component logic it’s all implemented in C++ to ensure the best performance, it’s mandatory to create the mirror classes in C#, so the user can easily interact with them in the scripting environment. Thanks to mono, those mirror classes have the ability to directly call C++ code using C++ and C# static bindings (F.5.30).

![C#](image1)

![C++](image2)

F.5.30 Here, C# code referencing an external C++ logic, and C++ code binding the C# function to the C# one.

If close attention is taken, it can be seen that a big problem arises here. Since instances C# code need to call other instances from C++, how is it possible to know which C++ instance is the one that the C# instance is representing? To put it as an example, if a C# GameObject class needs to call SetName(), how do we know which C++ GameObject instance is the one linked to the C# one.

The solution to this problem is not trivial, and it’s not contemplated by mono, so a custom solution is needed.

Once a C++ class instance, that also needs a C# mirror instance, is created, the C# instance is automatically created by the engine, and the C++ pointer address is stored in a string in the following way: F.5.31.

Storing the C++ pointer in a string allows the code to, at any needed moment, convert the pointer string into a pointer that properly references the C++ object. C# code does not have any specific way of handling C++ pointer, so storing it as a string works as a valid and fast solution. The only thing that needs to be taken into account is that the C++ pointer could be destroyed, and the string pointer would then point to unmanaged memory. Thankfully, this can be avoided with some error checking and consistency.
Procedure to store a C++ pointer address inside a string, and then convert this string into a C# string.

Then this string is automatically sent to a C# object (F.5.32) that is the parent of all the C# objects that need to be mirrored, and it’s stored there. With the C++ pointer stored on the C# class, we can now do the inverse operation and retrieve the C++ pointer at any time that we want, only having the C# class. With that, we have seamless and fast communication with C# and C++ instances, as can be seen at F.5.23.

Base class for all the objects that need to be C++ mirrored. The function to store the pointer address is also shown.
5.6.5. Editor references

Creating a script is essentially like creating a new type of component that can be attached to Game Objects. Just like other components often have properties that are editable in the Inspector, it’s also possible to allow values in the scripts to be edited from the Inspector too. This is achieved using a custom property that is part of the BeEngine scripting API named [ShowOnInspector], that once used on a variable on one of the scripts, if the engine supports it, the value will be shown as editable on the editor once the scripts are re-compiled, as can be seen at F.5.33. This is very useful to rapidly assign references to actual script values.

F.5.32 Function directly called by the C# code once the bindings are set up, where can be observed how the object sender is retrieved using the string address method.

F.5.33 At the left, script view of the elements that can be shown on the inspector, and on the right the actual inspector.
5.7 Physics

A physics engine is a computer software that provides an approximate simulation of certain physical systems, such as rigid body dynamics (including collision detection), soft body dynamics, and fluid dynamics, of use in the domains of computer graphics, video games, and film. Their main uses are in video games (typically as middleware), in which case the simulations are in real-time. The term is sometimes used more generally to describe any software system for simulating physical phenomena, such as high-performance scientific simulation.

BeEngine uses the well-tested and well-known 2D physics library named Box2D. Box2D provides a full set of features to make the lives of the developers easier at the time of integrating it into a game engine, it’s very easy to use, and it’s also very performant. A body in Box2D is defined by two things: a physics body, and its shapes, where a physics body can have an infinite amount of simple shapes, to create other more complex shapes (F.5.34).

In BeEngine, to create a nice and smooth integration with the engine editor, those shapes can be edited live using a custom gizmo, that provides the ability to move the vertices of the shapes, and also create new vertices (F.5.35).
Since Box2D only accepts concave shapes, and the user can modify the vertices of the polygon as he wants, it’s important to process and take care of the resultant shape, to avoid some problems that could appear. Another inconvenient of allowing any type of shape is collision detection. In the case of Box 2D, this process is only allowed using triangular shapes, so the objective is to process the initial shape that the user creates, and transform it into triangles, which are very easy to manage and always concave. In computational geometry, polygon triangulation is the decomposition of a polygonal area, finding a set of triangles with pairwise non-intersecting interiors.

There are several algorithms to transform an arbitrary polygonal shape into a concave one, but none of them are easy to implement. To solve this, BeEngine uses a library that performs earcut triangulation, which is performant and accurate. Once the shape is triangulated, all the resultant physics shapes are added to the physics body with the vertices of the triangulation, removing all possible convex polygons (F.5.36).

Once the shapes are created and are processed properly, they are ready to use with the physics body, and full physics simulations can be easily run adding a Polygon Collider Component and a Physics Body Component to any GameObject. Also, the engine has integration for overlapping detection, so detecting any type of collision using the scripting API is very easy and reliable.

F.5.36 At the left, actual shape of the polygon, and at the right, the triangulation performed by the engine.
5.8 Build

At any time while creating a game in a game engine, it is important to be able to execute and play it as a build outside of the editor. One of the most important systems of the engine is the build system. If a game is finished but it can’t be exported to be played as a standalone application, it’s worth nothing. Generally bigger engines have the ability to build games to a large number of platforms, but in the case of BeEngine, it can only export to Windows due to time restraints and complexity.

Generating a build (F.5.37) may seem trivial, but it can actually be a very difficult process depending on the technique that is going to be used and the chosen platform. In the specific case of BeEngine, a build is no other than the game engine loading a predefined scene at the beginning of execution, while having the Game window at fullscreen and hiding the editor UI. Using this simple approach, the engine code modifications have to be minimal, and the performance is increased due to the decrease of the UI calculations and unnecessary tools that are not used on build mode.

Once all those steps are done, the build is ready to be used. When the game is loaded, the engine looks for a build.bebuild file and, if it finds it, executes itself in game mode, and automatically starts all the necessary tasks to execute the game.
The first thing that the engine does, once it detects that is in game mode, is load the resources as fast as it can from the library folder, without making any integrity checks (unlike when it runs on editor mode). When all the game resources finish loading, the engine then looks for the scene that was predefined to load at the beginning. Once the scene is loaded the editor goes into play mode and the game actually starts normally as it would do in editor mode. The final result can be seen in F.5.38.

F.5.38 Engine running on game mode.
6. Sample Project

Since a game engine contains a large number of different functionalities that serve a variety of purposes, it’s difficult to properly test that all those tools work properly on a real environment without actually making a real game. Also, having all the components working in parallel allows spotting bugs or problems more easily, thus making the engine more stable.

The creation of the engine is, by itself, very time consuming, so creating a full videogame could be also very costly. To be able to finish everything on time, the chosen game has to be small and very fast to make, while also having all the necessary technical challenges to make use of all the features of the engine. Viewing the specifications of the engine and everything that had to be tested, the chosen game was a Pong type game. It basically allows for the main menu with buttons and text, a game scene with some moving parts, collision detection, and scripting, and some end game UI to finish the game loop.

6.1 Project development

To start with, all the art and the text font that appears on the game are from the Kenney Assets website (W.20), which provides free assets for video game creation. The first thing that was built in is the main menu scene. It contains a simple logo text, and the buttons for playing, credits and exit the game (F.6.1).

![Main menu from the game shown on the editor.](image)

F.6.1 Main menu from the game shown on the editor.
The animations from the menu fade in and out were made with a small scripting tool created specifically for this project, that uses easing functions to provide the smooth scrolling. Also, all the logic of the buttons is very simple and easy to use thanks to the BeEngine scripting API (F.6.2).

```csharp
if(play_button_go != null)
{
    ComponentButton button = play_button_go.GetComponent<ComponentButton>();
    button.SubscribeToOnClick(OnPlayButtonPressed);
}
```

F.6.2 C# code that shows how to bind to the OnPressed event on a Button Component.

Once the player presses the play button, the current scene is unloaded and the game scene is loaded, all with a smooth fade to hide the sudden change. Then, a timer starts counting down and the pong match actually starts. Every player is controlled by two keys on the keyboard, “q” and “a” for the player one, and “o” and “l” for the player two (F.6.4).

F.6.4 Main game on the editor
When the match starts, the ball starts moving, choosing from a random direction: left or right, and then, when it detects collision with one of the up and down walls the angle in which is moving is mirrored to simulate a real bounce. None of the actual movement is done by physics, in this project they are only used to detect overlapping between objects. Then, when a player touches the ball, the new movement angle of the ball is chosen depending on the place of the where the ball collides, just as the original Pong. Finally, when the ball touches one of the left or right walls, a point to the corresponding player is added and the ball is reset in the middle. Once one player gets 3 points, the match finishes and the player that wins is announced with a screen, moments before returning to the main menu, as seen at F.6.5.

![Player wins scene on the editor](image)

6.2 Learnings

Making this small game turned out being very useful for the development. It showcased some bugs that were hidden on the normal engine development, and it allowed to see every piece working properly, and the actual performance on the engine running with several scripts and from 15 to 25 game objects, which was around a stable 250-300 fps. Finally, it did not take much time, and it serves as a testing and showcase project for the final product.
7. Conclusions

As mentioned before, normally game engines are huge tools created by hundreds of different people with a big variety of skill sets and experience, so the creation of an engine with only one person during the span of several months is a very ambitious task.

One of the principal objectives of this project was to aim towards simplification while not losing any of the functionalities that a commercial engine could have. In this project, this his is accomplished keeping the different functionalities of the different tools and modules clearly separated (Inspector, Hierarchy, Scene, etc.), while also providing a quick workflow managing files and creating new objects. Also being able to save and update the progress in real-time and previewing it at any moment allows to spot problems and observe the progress more effectively. Overall, the experience of creating a simple 2D game with BeEngine is very easy and fast.

From the other side, aiming for this simplification in addition to the big scope of the project, also brought some problems. Even if the number of tools and modules that were planned to be implemented are much smaller than in a big commercial engine, there are still a lot of necessary utilities that are indispensable to make games, that must be reliable and well thought. As the time for creating all those tools and modules is limited, it’s difficult to go technically and technologically deep on the development of everyone them, so only the ones that were more interesting were explored with more emphasis, such as the scripting module or the file system. Others like the physics or the rendering were created only with the basic functionalities.

With all the work, time and knowledge needed to create a game engine, I asked myself the same questions several times during the process of development: Is this all really worth it? Do I really need to do all this work, when the final objective of my endeavor is to create video games? Making a game engine from scratch is not for the faint of heart, and if you’re not an experienced programmer you probably shouldn’t attempt it. If you are an experienced programmer, then you still probably shouldn’t do it. Not that it’s unachievable, it just might be a huge waste of time.

If you want to create and publish a game as quickly as possible, whether for profit or as a portfolio project, then you should almost certainly use a proprietary engine. The tools they provide can cut down on development time enormously, and streamline many of the more complicated aspects of graphics programming. The (rare)
exception to this rule would be if your game has specific performance requirements beyond the scope/capabilities of a general-purpose engine.

If your goal is to build your skills as a programmer and deepen your understanding of underlying game architecture, then making your own engine can be extremely rewarding. Also, if you have the extra time and skill necessary, there can be practical benefits to rolling your own engine. You’ll control the game logic at every level, nothing will be obfuscated, or black-boxed, as it is in many commercial engines. You won’t have to worry about licensing, revenue sharing, or extra per-platform costs. There won’t be any question as to whether your engine will still be supported a couple of years down the line. Finally, it can be a lot easier to identify bugs and performance issues when you’ve built everything yourself. This is, of course, assuming you make your engine well. A poorly designed, inefficient engine might not run Pong at a stable frame-rate.

Next, the following table goes through the objectives settled at the beginning of the project, to analyze its final state and the progress overall.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>O1. Design and develop a 2D game engine with C++ and OpenGL (with shaders).</strong></td>
<td></td>
</tr>
<tr>
<td>This objective has been achieved, since it’s the one that’s more generic and broad. The engine is functional and it’s capable of produce basic 2D games; it’s written using C++ and C#, and uses OpenGL as the base rendering backend.</td>
<td></td>
</tr>
<tr>
<td><strong>O2. Build all the necessary tools inside of the engine to allow the user to create a 2D game.</strong></td>
<td></td>
</tr>
<tr>
<td>The engine contains all the basic tools to create simple 2D video games such as sprite rendering, 2D physics, a scene system that can be used to load and save progress, and file management, all presented within a clean and clear editor with different dockable windows that can be adapted to any workspace. Also, to control the game logic, it contains a complex and powerful scripting system on top of C# that surfaces only the necessary functionalities, to allow the user to build clear and concise logic, without the engine getting in the middle.</td>
<td></td>
</tr>
<tr>
<td>From the other side, due to time constraints, some tools or systems could have been more explored or developed, such as the audio system, that is only capable of reproducing simple audio, or the UI system that only contains very basic widgets. Still, with all of that, the engine is completely usable.</td>
<td></td>
</tr>
</tbody>
</table>
O3. The ability to easily create builds targeting the Windows platform.

At any moment, given a starting scene, the engine is capable of deploying a build version of the game in which the editor is hidden and the game takes full screen. The chosen scene is automatically played and the application plays like a standalone game. The creation of a build is very simple, since the engine generates all the necessary files only with the click of a button, with the capability of deploying only for the Windows platform. All the settings for deploying the final game are packed inside the Build window of the engine.

O4. Create one or two simple games to demonstrate the capabilities of the engine.

To test the engine performance and usability in a real environment, a pong-like game has been made, trying to utilize all the different aspects that the engine has, while fitting inside the already packed time constraints. The game is simple but fully playable, running in a more than exceptional performance in lower-end machines.

Next, talking about the original time schedule, all the tasks have been completed in time, without any delays or pushes on the schedule. From one side, this is thanks to the generous time spans given to the most critical or difficult tasks, that allowed for exploration and problem solving if needed, and also thanks to the early start of the project, allowing enough room to explore and develop all the different possibilities.

Even if all the modules of a game engine are equally important and equally necessary, for the sake of learning and interest some of them were more deeply explored and developed than others. One of them is the scripting module, where a lot of research had to be done since there was little to no information in this aspect. It is an interesting field and it was given priority since it can serve as useful information for people that, in the future, want to implement the same system or do a similar project, and do not have the time to research and test blindly.

The thought behind creating an engine that was simpler and easy to use, and that I could use for any of my future projects, sounded like a very interesting and pretty idea at the beginning. The truth and experience that I stumbled upon could only have been found pursuing a project like this, so even if the result was different from the expected, the learnings that I take will surely serve me for all my future game development career.
The result of this project: BeEngine 2D Game Engine, it’s public and under the Creative Commons Attribution-ShareAlike 4.0 International License in the GitHub repository (W.2) BeEngine-2D.
8. Acknowledgments

First, I would like to express my deep gratitude to my project director Jesús Díaz, for his patient guidance, advice, enthusiastic encouragement and useful critiques given during all the project.

Also, to my friends and college mates, thank you for listening, offering me advice, and supporting me through this entire process.

To my family, thank you for encouraging me in all of my pursuits and always inspiring me to follow the things that I love.

And finally, special thanks to Emma Callau, for helping me with the writing and the revision process, and for always giving me invaluable support.
9. Webgraphy

W.1 Final project of the third degree of Videogames Design and Development at the University Politecnica de Catalunya. Landing page. Webpage, URL <https://project-3-upc-ddv-bcn.github.io/Project3/>


