# Viability of developing a video game with Unreal Engine

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Authors: Félix Hortelano Bronchal

Guillem Bonafonte Pardàs

Director: Lluís Solano Albajés

Bachelor: Computer Engineering

Specialization: Software Engineering

Facultat informàtica de Barcelona (FIB)

Universitat Politécnica de Catalunya (UPC)

## Abstract

This project was conceived as a way to explore Unreal Engine ourselves and learn a bit how this popular engine works. Once we knew what we wanted to do, we decided to focus this project from another point of view. We want to simulate that we are working in a video game company that has asked us to see the viability of developing a videogame.

We want to see what Unreal Engine offers us and what we can achieve with it. In addition, we want to see how hard it is to do certain features of a game.

Therefore, we have done a video game, which tests different Unreal Engine features such as artificial intelligence, shooter character, networking, ability system, gold system, weapon system…

The main difficulty we have found in the project was our knowledge on complex things such as networking and how to properly debug it.

Despite the problems that might appear, we have explored different alternatives on every task in order to choose the best option in any case, so we can say if it is really viable or not to develop a game with Unreal Engine.

## Resumen

Este trabajo ha sido pensado a partir de la idea de aprender Unreal Engine nosotros mismos y ver cómo funciona este motor que tanto ha crecido en popularidad últimamente. Una vez sabíamos que queríamos hacer, decidimos enfocar el proyecto desde un punto de vista más profesional. Queríamos simular que estábamos trabajando en una empresa de videojuegos y que se nos ha pedido que estudiemos la viabilidad de desarrollar un videojuego con Unreal Engine.

Queremos ver que nos ofrece realmente Unreal Engine y que podemos llegar a hacer con él, así como ver la dificultad de este mismo.

Hemos hecho un videojuego que testea diferentes aspectos de Unreal Engine como por ejemplo la inteligencia artificial, un personaje al estilo shooter, el multiplayer de Unreal, un sistema de habilidades, un sistema de armas, etc.

La principal dificultad que hemos encontrado en el proyecto es la falta de conocimiento que teníamos no solo en Unreal Engine sino en temas complejos como hemos visto que ha sido el networking y como debugarlo.

A pesar de los problemas que han aparecido, hemos explorado diferentes alternativas en cada tarea para así intentar coger siempre la mejor opción en cada caso. Así podemos decir cómo de viable es desarrollar un videojuego en Unreal Engine.

## Resum

Aquest treball ha sigut pensat a partir de la idea d'aprendre Unreal Engine nosaltres sols, per així veure com funciona aquest motor que tant ha crescut últimament. Un cop sabíem el que volíem fer, vàrem decidir enfocar el projecte des de un punt de vista més professional. Volem simular que estem treballant en una empresa de videojocs i que ens han demanat que estudiem la viabilitat de desenvolupar un videojoc amb Unreal Engine.

Volem veure que ens ofereix Unreal Engine realment y que podem arribar a fet amb ell, així com quins inconvenients ens hi podem trobar.

Hem fet un videojoc que prova diferents aspectes de Unreal Engine com per exemple la intel·ligència artificial, un personatge a l'estil shooter, el multi jugador de Unreal Engine, un sistema d'habilitats, un sistema d'armes, etc.

La principal dificultat que hem trobat al projecte ha sigut la falta de coneixements que teníem no només amb Unreal Engine, sinó també amb temes com el multi jugador o com debugar-ho. Malgrat els problemes que han aparegut, hem explorat diferents alternatives en cada tasca per intentar agafar sempre la millor opció en cada cas. Així podem dir si és realment viable o no desenvolupar un joc fent servir Unreal Engine.

# Index

# Context

At this section we want to introduce what the project is going to be about; which people is going to get involve with our work and why and what is the actual state of the market in which we are trying to get into.

## Introduction

Video games market is increasing day by day and we want to see how things are done in this market. There are different options when you want to develop a video game, you can write everything on your own without the help of other tools that make things easier for you in order to develop the game you want. Or you can use different engines which helps you to do certain things. This option is one of the most used and that is the reason why we are focusing on one concrete engine, Unreal Engine. In order to make the project as realistic as possible we want to pretend that we are working in a company which has been developing games for some years with Unity 3D, which is one of the most popular game engines.

As our company wants to be a leader company, aware of the actual market and ready for all the future changes, we have been asked about Unreal Engine, in order to be prepared to face all the difficulties that could appear in case we ever want to change our game engine and to know how to solve these difficulties and being able to choose which is the best option for each type of game we want to develop.

## Stakeholders

Primary stakeholders:

- Developers: we are probably the most important actors of the project. Besides our motivation of finishing the degree, we have a personal interest in front the video games world, and that is the reason why we are studying computer engineering. We also have a special interest on knowing how Unreal Engine works.
- The company: as we pretend to be working with them, they also have an interest of discovering if Unreal Engine is going to make them increase their value as company, not just having more earnings due to selling but also increasing their reputation in the video games market and the quality of their video games.
- The director: In this particular case, our director Lluís Solano is also a stakeholder due to his interest on seeing how a game performs in Unreal Engine because he has been using Unity3D for some years.

Secondary stakeholders:

- Clients: The company clients are also part of the stakeholders because they want to take profit of our work by using it.
- The competition: other companies, which are in the same situation as our company, would be interested in our product, because if it goes well they might follow the same steps as we did and starting to launch their product with Unreal Engine, or using some ideas, we created on their own products.

## State of the art

At the moment there are different engines used for developing video games, some of them are Unity3D and Unreal Engine. Many popular games have been developed with Unity3d, such as Pokemon GO or Hearthstone, and millions of not-so-popular games, but not bad at all, that flow the net every day.

Moreover, the same happens with Unreal Engine, which has games such as Rocket League, ARK, Mass Effect, Daylight….

Here we can see an image from the Unity official website, where we can see how they are dominating the industry with a 45% of the market, followed by their closest competitor, which is Unreal Engine, with a 17%.

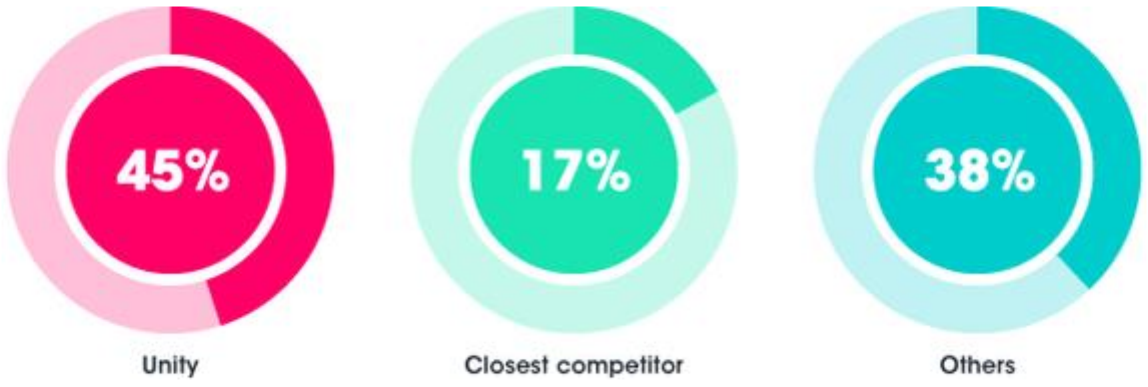


*Figure 1: Market distribution*

The other 38% of the market is composed of others engines like Torque3D, Cry Engine, cocos2d … But an important part to consider is the internal proprietary engine that are the ones that companies create for themselves to fit their needs and improve the performance of their games.

For example, if you are brave enough you can download the source code of Unreal Engine, which is free, and modify it to fit your needs so you can do things that Unreal Engine cannot do, but this has a high cost that not all companies can afford because not everyone knows how to do it correctly.

Those statistics are extracted from the internet, but if we go a little bit deeper and look what big companies use to develop their games we can say that the most common thing is to create their own game engine, as they can adapt their own necessities and functionalities. They use this kind of strategy because they are planning to do a huge game or a series of huge games of the same style, as they can reutilise most part of them and have more specific tools that you know that will be useful in the future.

As we are not going to use this strategy, we are going to focus on Unity3D and Unreal Engine.

## Differences between Unity3D and Unreal Engine

In this study, we are going to focus on Unreal Engine, which is the engine we are going to use, and Unity3D, which is the engine our company actually uses. We can state some facts that make the principal differences between both game engines:

- **Pricing** Unity has a pro version available for $125 per seat per month, while Unreal is free to use, but the 5% of the product revenue after the first $3,000 per game per calendar quarter from commercial products goes to Epic Games, company of Unreal engine.
- **Documentation**: Unreal documentation is a bit more extensive (more screenshots, better explained…), but Unity has many video tutorials on their webpage: learn.unity3d.com. **Update:** After doing this project, we have discovered that Unreal documentation is not what we thought about. Documentation is mainly focus on Blueprints rather than C++ code and this can cause trouble if you are trying to look for a concrete thing. For example, on the networking part Unreal Engine has not the amount of documentation we would like.
- **Platforms**:
    - Unreal: Windows PC, Mac OS X, iOS, Android, VR, Linux, SteamOS, HTML5, Xbox One, and PS4.
    - Unity: Windows PC, Mac OS X, Linux, Web Player, WebGL, VR (including Hololens), SteamOS, iOS, Android, Windows Phone 8, Tizen, Android TV and Samsung SMART TV, as well as Xbox One & 360, PS4, Playstation Vita, and Wii U.
- **Rendering**: Unity allows Per Pixel rendering and Per Object rendering, while Unreal only allows Per Pixel. This is the reason why some platforms are not supported by Unreal.
- **Graphics**: In some graphic aspects Unreal is one-step ahead than Unity (terrain, particles, post processing effects, shadows & lighting, and shaders). Although Unity have an improvement with his last version, Unity 5.

To conclude, the most common opinion when you have to decide what engine you should use in order to develop your game is that, firstly, you have to consider the kind of game you want to create. Once you got this idea clear, you proceed to choose the best engine for your game. For example, if you want to make a mobile game, after a bit of research you will notice that Unity3D is better for this purpose. However, if you want to make a realistic game for PC/PS4/XBOX Unreal is the best option, unless you have the knowledge, money and time to develop your own engine that fully fits your necessities.

## Project description

Here we are going to explain what the project is going to be about and what it is going to have and how we are going to do it.

### Formulation of the problem

One of the common problems companies have is the lack of awareness of different engines that are not the ones they use normally. So that we have been asked to develop a game with Unreal Engine in order to know his pros and cons. Therefore, the first step to do is to get used to Unreal. Every game engine is a whole world and every detail, even if it seems insignificant, may end up meaning a huge result in the final product. We must be aware of the different problems or situations we will have to face, and be conscious that every single decision has an important impact on the project and the development of it.

To solve this, we want to identify every significant part and get a list of the ways to solve each one, and which approach should be used in each situation. Testing them and checking that they are actually viable for a project is necessary in order to make our work meaningful.

### Objective

The main objective is to develop a 1 vs 1 multiplayer game, applying the best techniques we have been learning through the degree. Therefore, we want to achieve a full video game that can actually prove that this could be viable and that every single choice was rightly done and have a reason to be like this.

As it is well known by everyone video games cover a huge range of possibilities, and developing a simple game as Tic-Tac-Toe is not enough to prove ourselves that this engine is going to fit our possibilities if someday we need it.

For this reason, we are making a list of the main features we think our video game must have in order to prove that we are ready to use this engine, and to make our decisions significant in a way it can be reused in a future when developing other complex games in this engine.

First, we need to explain the topic of the video game to understand the features it will have. The game is going to be a multiplayer video game where two different players are going to “fight” each other and use their abilities to prove who will win. One player’s role is to escape of the maze, and the other player is avoiding the first one to escape.

Starting from this point the game will have the following big features:

- Multiplayer: Since we said, it is going to be a 1vs1 game it need a multiplayer system in order to let players interact between them even if they are not in the same network or computer.
- Database: there is also the need to store the information of the games in order to show statistics of the players and show them some interesting information about how well they play.
- Artificial Intelligence: we also said that there is going to be some specific abilities for each player for example spawning bots so we need to create algorithm that make bots behaviour in a certain way depending of the environment.
- Principal scenario: here we have to create a correct scenario to show the capabilities of Unreal Engine in graphic performance and this will be the point of union of all the features we explained.
- Game interface and sound: As every game needs, it has to have an interface, which has to be intuitive, and sound to make the game more comfortable.

Once we know these features, we will break down them and make smaller ones in order to develop the project.

## Scope

As we explained in the previous section the scope of the project is based into achieve the features we explained, now we are going to explain a little bit deeper what we want to do:

- Multiplayer game: so, everybody can use it and we have to consider having a high number of active users.
- Database: to store all user's information considering again to have a high number of users. All these information is saved in order to give the user useful information to help them improve their game experience.
- One player: the player who have to escape of the maze will have different abilities in order to help him escape. The player has to choose a certain number of abilities on each game so he will have different strategies to try.
- The second player: this player has to prevent the first one from escaping. It also has some abilities to help him and he also have to choose a certain number of abilities on each game so he will have different strategies to try.
- Level system that would let the player unlock new abilities.
- Gold system that would let the player use their abilities during the game in exchange of gold.
- Sound appropriate to the player's actions and the environment.
- Graphic user interface: which has to be user friendly and intuitive.
- The maze: a handmade created maze in order to test the graphic performance of the engine.
- Some auto generated mazes in order to make the game less repetitive.
- Single player option: there will be an option to play against the machine, by playing the role of escaping the maze.

As we do this project in pairs, we divided those tasks so each one of us is going to do some of them that at the end we are going to merge in order to have the full game.

The task assignation is the following:

- Felix Hortelano: first player, handmade maze, gold system, database, sound.
- Guillem Bonafonte: second player, auto generated maze, level system, multiplayer server, graphic user interface.

## Possible obstacles

The most important thing we are going to face with will be:

**Time restriction**

Any problem that can appear is going to be decisive to the final result because we have a deadline for the project, so that we have to manage the best we can to make the problem affect us the less possible. Thus, we are going to follow a timetable that we will set weekly and we will ensure the project follows it.

**Ignorance of the engine**

Since we never developed with Unreal Engine we are going to have some difficulties at the beginning, so that we have to make an effort to learn as quick as possible the main features that the engine let us, and decided if they are enough to develop the game we want to.

Thus, we will set some time in our timetable to secure that we know how everything works and avoid getting stuck in this step of the project.

**Internet**

Since it is a multiplayer game people will need Internet to play it. It is not a big deal because nowadays almost everybody has Internet at home and the game will be for PC.

## Unreal Engine description

As we are going to use Unreal Engine as the main tool to develop our project it is worth to make a description of what it is about and what things you can do with it.

Unreal Engine is a game engine made by Epic Games in 1998. It has evolved during the years and that's the reason why we are now at Unreal Engine 4. But a bit of history of what has been changing during the different versions going to help us understand better where we are.

**Unreal Engine 1** appeared in 1998 when it was created and was the first version they release, it includes both software and hardware rendering, texture filtering, collision detection and coloured lightning. It also had a level editor called UnrealEd where developer could construct geometry. Epic added some improvements to it over the years and the engine become popular due to the modular engine architecture and the inclusion of a Scripting language called UnrealScript.

**Unreal Engine 2:** this version saw the light on 2002 and it includes a rendering engine completely rewritten. It also includes an improved level editor called UnrealEd2. After some time appeared UE2.5, an update to the original version of UE2, which improved rendering performance and added vehicles physics.

**Unreal Engine 3** was designed to take advantage of fully programmable shader hardware (in DirectX 9 terms, it required shader model). All light calculations were done per pixel instead of per vertex.

**Unreal Development Kit** (UDK) was available to general public, not only professionals who paid the license.

**Unreal Engine 4** appeared in 2012 under limited attendees at the 2012 Games Development Conference. But the major change appeared in 2014 when Epic Games released Unreal Engine 4, and all of its tools, features and complete C++ source code, to the development community through a new subscription model. Making it available to smaller teams and not only AAA companies which had enough money to afford this powerful tool.

Another important aspect to consider is how Unreal Engine affects to the money you can earn with a game developed with this software, and the fact is that developers are obligated to pay to Epic 5% of all gross revenue after the first $3,000 per game or application per calendar quarter, regardless of what company collects the revenue.

## Visual Studio description

Visual studio is an integrated development environment (IDE) which needs to be used with Unreal Engine in order to develop games in C++. Which is the only language this engine allows us to use.

Although there are two ways of programming in Unreal Engine one with C++ itself and other with blueprint, which is a visual way of scripting that at the end, does more or less the same as C++.

In order to keep talking about Visual Studio explaining a bit more what we are going to do with it, it is going to help us to compile, debug and organise the C++ code we have done.

## Blueprints vs C++

As we already comment in the previous section, Unreal Engine allows developers to develop in two different ways: Blueprints or C++ code:

- Blueprints: is a complete gameplay system based on the concept of node-based interface to create gameplay elements within Unreal Engine. As other scripting languages is it used to define Object oriented classes or objects in the engine.
- C++: it is an imperative, object-oriented programming language which allows us to apply all the things we have been learning during the degree and to refresh it a bit as well.

Once we know the different ways of developing which of them we should really use? Well after some struggling with this question we can clearly say that the main difference between C++ and Blueprints is the performance, is it known that games developed with C++ perform better than the same games done with Blueprints. However, this statement does not mean that we need to do just one or the other, both ways of programming can be combined as well and in fact is very common to do it this way. Developers used to write the C++ implementation and designers used to do the gameplay with Blueprints based on the classes in C++. That is the reason why we have decided to do as much as we can in C++ but doing some things in Blueprints as well because is a quicker way of prototyping your game. We also have to say that in C++ you have available all the source code of the engine while in blueprints there are certain parts that can’t be done yet, so that you might be forced to use C++ from time to time.

## Game description

As we mentioned at objectives we want to do certain things such as multiplayer, database, artificial intelligence, principal scenario, and game interface and sound. Nevertheless, we need to explain them deeper so we know clearly which ones we will do and which ones we will not. We also need to prioritise these different parts in order to know which are more important.

To remind what our game is going to be we will start with a brief description. Our video game is a 1vs1 multiplayer game, where each player will have a different role. Having 2 separated roles allows us to explore more things and different features in video games.

One of the players will be a typical shooter character, with some weapons, abilities, and third person camera. The objective of this player is to escape of the maze in a certain time. As a shooter player, this character will have properties as health, ammo or gold, which player can spend using his abilities, and a custom interaction with the IA, depending if they are allies or enemies.

The other player will be the enemy of the first one. The main objective of this player is to avoid the shooter character to reach the exit of the maze. We have done a different character instead of making another shooter player because we though that it allows us to explore more things in Unreal engine and it helps to divide work as well as we are two people, that is why we decided to create a "god" character. A god character is a first person camera that moves freely around the map, without physic limitations, and has no health nor mesh. This character cannot die, because its only objective is to avoid the shooter character to win, in order to win himself after certain time. You cannot stop a player reaching a point just by moving around, so this player has as well a pack of abilities that put obstacles to the shooter player. Obstacles as might be mines, IA spawns, fake walls, etc.

Both player will have a gold system that regulates their abilities, so they are not allowed to spam them all the time without getting punished. Each ability is going to have a countdown time until you can use this ability again, assuring that you do not abuse a determined ability and win by a combination of them.

After a determinate time, if the shooter player has not been able to reach the exit point, the other player wins. Assigning which role will take each client when entering a game is completely random so you will always have shorter queues and someone willing to face you.

# Planning

## Initial planning

The ***initial planning,*** we did was the following. We are going to analyse how the initial planning was and how it finally went, and explain the reasons of all the deviations and how we dealt with them. The deviations will be explained after the Design and Implementation part.

## Calendar

The project has an estimated duration of 4 months, considering we started at the end of January and we will end at June. This planning may change a bit because some tasks can take more time than expected, we can introduce or delete some tasks as well and this will cause a variation on time.

## Task description

Our project is going to be divided in different tasks that are going to help us know at which phase we are, and how much is left.

### Project planning

This is the initial part of the project and related with GEP subject. We consider it is an important part because we planned everything we are going to do, and without these planning we would not know what to do or in which order.

Once this part is over we will be able to start directly into the project itself, because most of the part is going to be implement everything we need, despite of some previous work we will have to do in order to get used to the new technologies. We considered project planning is one-month length.

## Software analysis and get in touch

Although we had done some work of these part during the project planning we will have some time to get in touch with the technologies we are going to use, Unreal Engine most of the time, and get familiarised with it in order to work better. This part it is necessary but has to be as short as possible in order to not waste too much time here. If we need to learn new things in the future, it is going to be while we develop our project, as we are using an agile methodology.

## Architecture design

Before starting the project, we have to think how we wanted to do the architecture, our actual idea is to develop the game using MVC pattern but we have to get in touch a bit more to know that this is the best approach we can do. In this process we are also going to look for the database we are will use and the server where we will run the game.

## Software Implementation

At this point, we will have all the necessary tools to start implementing the game. Therefore, we are going to divide the work between both of us and explain what we are going to do on each part.

Felix Hortelano tasks

- Handmade maze: Here we have to create the maze and texture it bottom to top in order to be playable by both players. This will be the unique part of the game, which we are going to design by ourselves, because we thought that a handmade maze is more realistic than the auto generated ones.
- First player: this player consists in the one who will try to escape from the maze and all his related activities, such as interaction with other elements, abilities … We have decided that the game style of this player will from a shooter perspective, so we will also have to look for different weapons. All the models needed for the player are going to be taken from Internet since we want to finish the whole game and designing models is out of our speciality.
- Gold system: which is going to be for both players and the main objective of this part is making an intelligent and equitative way to gain gold on both players to do not cause unbalance between them.
- Database: Here we will store all the information we need in order to show statistics at the end of the game and give players feedback about their performance.
- Sound: Include sound in all the elements that need it.

Guillem Bonafonte tasks

- Auto generated maze: Even we want to have a nice map done by ourselves, we think that making auto generated mazes to have more variety is a good idea so it doesn't get to repetitive.
- Second player: this player will be the one trying to stop the first player get out of the maze. He will have a gold system too and it will be played in a "god" camera. The importance here will be making it easy to interact with the rest of the maze and the abilities it will have.
- Level system: We will create a level system and make a gamification system so people gets more involved with the game. This means that every game gives you experience and levels your account up, so you get some kind of rewards that makes the game more enjoyable.
- Multiplayer: Obviously, as we have 2 players, this game will be multiplayer. But we will make this game not only multiplayer but online multiplayer, so you can play without actually being actually in the same device.
- Graphic and User Interface: Our objective is to do a friendly GUI so player knows how to use each option of the game. This GUI will be in game and on menus, so you can navigate easily between both, for example.

All the task of both of us are going to be done in parallel because at the end they can be done separately, we only have to take care that both parts merge in the correct way, and prevent the apparition of bugs.

## Testing

Testing is one of the most important parts when using agile methodology. A continuous process of testing helps you to be sure that you are going in the right way and to check that every sprint has been done properly. This part of testing will be done by ourselves and even that we would like to create auto generated tests we do not think that it will be possible because it is very difficult to achieve when developing video games.

Once we have a playable version of the game we want to grant access to some people in order to have feedback of the game and know what a normal user would think about our game in order to listen his opinion and if possible (due to time restrictions) solve their problems. See how the user interacts with the game give us relevant information of where we should focus to make the game more intuitive and playable.

### Defence of the project

This is the last part of the project but the most important one. At this part, we will show our game and explain every step we have done. We will explain in detail every decision we made for each part and what other options we found during the process. We will also present all the documentation done in GEP and the one we are going to add during the tasks we are going to do.

The defence of the project will be done in June, once the project is over.

## Alternatives and action plan

### Estimated time

| Step | Dedication time |
|---|---|
| **Planning** | 50 |
| **Analysis and design** | 150 |
| **Implementation** | 200 |
| **Testing** | 150 |
| **Defence preparation** | 50 |
| **Total** | **600** |

*Table 1: Estimated time*

### Action plan

Our project has a length of 16 weeks. All the number of hours we stated above were an estimation, because when using the agile methodology, you have to adapt to the requirements of the project. After about 200 hours of planning, analysis and design of the project, we should be able to start implementing and testing. As it is known, agile methodology does the testing and implementation at the same time, so we expect this to be the biggest part of the project.

In order to know we are on the correct way we will make biweekly meetings, and try to correct what is wrong as soon as possible and we also will try to solve the doubts we have.

We have a total of 16 weeks to do the full project, so 16 weeks, doing 18 hours a week each member gives us the total of 600 hours (approximately) which makes the project viable.

### Resources

We will be using definitely Unreal Engine and Maya for developing the videogame. Once we have clear our requirements for the storage of information and others, we will decide the server and the BBDD we are going to use. We will also use our desktop computers to develop rather than our laptops.

## Budget

As it is known, almost every resource when you want to develop a project have a price. Therefore, a project estimation has to be done even though we are students and we have access to some free software. We have to take into account different aspects such as hardware, software or human resources.

As a result, the final budget will be based on real times and risk that can appear during the process; also, unforeseen contingencies have to be handled if they happen.

### Project cost

Even the project is developed by students, we must consider that is develop in a real company, which we already supposed we are working in. For this reason, it is essential estimating the cost of the development project, because is another reason to know the viability of it. At this section, we are going to describe the direct and indirect costs and as we mentioned before we are going to describe the unforeseen contingencies as well.

### Direct costs

These costs are the ones connected with a specific object, which may be a product, department… As this is a software project, direct costs include items like labour, software licenses, hardware…

## Human resources costs

Even this project is going to be developed by 2 students, both of us are going to perform the roles of: project manager, analyst, software developer and tester. So, in the total of 600 hours we are going to do we have to differentiate between roles. In the following table we can see this estimation.

| Role | Estimated hours | Estimated Price per hour | Estimated Cost |
|---|---|---|---|
| **Project manager** | 63 | 25€/h | 1575€ |
| **Analyst** | 97 | 20€/h | 1940€ |
| **Software developer** | 290 | 15€/h | 4350€ |
| **Tester** | 150 | 10€/h | 1500€ |
| **Total cost:** | | | **9365€** |

*Table 2: Human resources costs*

The human resources price per hour are an estimation based on salaries at these roles in Spain.

The previous table can be more specific if we follow the Gantt task we had done previously. And the table will look like that

| Task name | Task hours | Assigned to | Role cost per hour | Task cost by role |
|---|---|---|---|---|
| **Planning** | **50** | **Project Manager & Analyst** | | **1190€** |
| **Scope** | 6 | Analyst | 20€ | 120€ |
| **Temporal planning** | 8 | Project manager | 25€ | 200€ |
| **Sustainability and budget** | 5 | Project manager | 25€ | 125€ |
| **Preliminary presentation** | 9 | Project manager | 25€ | 225€ |
| **Speciality deliverable** | 6 | Analyst | 20€ | 120€ |

| Task name | Task hours | Assigned to | Role cost per hour | Task cost by role |
|---|---|---|---|---|
| **Final documentation** | 8 | Project manager | 25€ | 200€ |
| **Oral presentation** | 8 | Project manager | 25€ | 200€ |
| **Analysis and design** | **150** | **Analyst & Software developer** | | **2550€** |
| **State of art** | 20 | Analyst | 20€ | 400€ |
| **Objectives and requirements** | 40 | Analyst | 20€ | 800€ |
| **Design** | 90 | Software developer | 15€ | 1350€ |
| **Implementation** | **200** | **Software developer** | | **3000€** |
| **Handmade maze** | 35 | Software developer | 15€ | 525€ |
| **First player** | 25 | Software developer | 15€ | 375€ |
| **Gold system** | 15 | Software developer | 15€ | 225€ |
| **Database** | 15 | Software developer | 15€ | 225€ |
| **Sound** | 15 | Software developer | 15€ | 225€ |
| **Auto-generated maze** | 25 | Software developer | 15€ | 375€ |

*Table 3: Human resources detailed(1/2)*

| **Task name** | **Task hours** | **Assigned to** | **Role cost per hour** | **Task cost by role** |
|---|---|---|---|---|
| **Second player** | 20 | Software developer | 15€ | 300€ |
| **Level system** | 15 | Software developer | 15€ | 225€ |
| **Multiplayer** | 20 | Software developer | 15€ | 300€ |
| **GUI** | 15 | Software developer | 15€ | 225€ |
| **Testing** | **150** | **Tester** | | **1500€** |
| **Handmade maze** | 10 | Tester | 10€ | 100€ |
| **First player** | 20 | Tester | 10€ | 200€ |
| **Gold system** | 10 | Tester | 10€ | 100€ |
| **Database** | 10 | Tester | 10€ | 100€ |
| **Sound** | 10 | Tester | 10€ | 100€ |
| **Auto-generated maze** | 10 | Tester | 10€ | 100€ |
| **Second player** | 20 | Tester | 10€ | 200€ |
| **Level system** | 10 | Tester | 10€ | 100€ |
| **Multiplayer** | 40 | Tester | 10€ | 400€ |
| **GUI** | 10 | Tester | 10€ | 100€ |
| **Defence project** | **50** | **Analyst & Project manager** | **25€** | **1125€** |
| **Total** | | | | **9365€** |

*Table 4: Human resources detailed 2/2*

## Software costs

In addition, some software is needed to carry out the project. At the following table, we can see what we need:

| Product | Prices | Units | Months | Total estimated |
|---|---|---|---|---|
| **Unreal Engine** | 0€ | 2 | NA | 0€ |
| **Autodesk Maya** | 242€ | 2 | 4 | 1936€ |
| **Github** | 8.5€ | 1 | 4 | 34€ |
| **Total** | | | | 1970€ |

*Table 5: Software costs*

## Hardware costs

We will also need hardware to develop the project in order to implement, design and test. In the next table, we can see the resources that we need.

| Product | Price | Units | Useful life | Total estimated |
|---|---|---|---|---|
| **WorkStation II Intel i7-7700 / 16GB / 2TB + 240GB SSD M.2 / GTX 1060 6GB** | 1500€ | 2 | 4 | 375€ |
| **Total** | | | | **375€** |

*Table 6: Hardware costs*

## Total direct costs

By adding, all the budgets explained above we have the following cost.

| Concept | Estimated cost |
|---|---|
| **Human resources** | 9365€ |
| **Software resources** | 1970€ |
| **Hardware resources** | 375€ |
| **Total direct costs** | **11710€** |

*Table 7: Direct costs*

## Indirect costs

These costs include office equipment, office rent, electricity, water and other utilities. All these items are not assigned to any service but contribute to the company. In the following table we can see these costs.

| Product | Price | Units | Useful life | Total estimated |
|---|---|---|---|---|
| **Equipment (desks, chairs…)** | 150€ | 2 | N/A | 300€ |
| **Office rent** | 300€/month | 1 | 4 months | 1200€ |
| **Electricity and water** | 100€/month | 1 | 4 months | 400€ |
| **Total:** | | | | **1900€** |

*Table 8: Indirect costs*

## Unforeseen contingencies

As a contingency measure, a margin of 10% is established over the global cost of the project. We established a margin, for cost and time, so every unforeseen contingency can be dealt with enough prevision and does not affect the outcome of the project.

## Total costs

| Concept | Estimated cost |
|---|---|
| **Direct costs** | 11710€ |
| **Indirect costs** | 1900€ |
| **Subtotal** | 13610€ |
| **Unforeseen contingencies(10%)** | 1361€ |
| **Total** | **14971€** |

*Table 9: Project costs*

## Methodology and rigor

### Work methodology

We are going to develop this project based on SCRUM methodology. SCRUM is an iterative and incremental agile software development. It is used when a development teamwork as a unit to reach a common goal as in this case is the game itself.

We are going to do our sprint of 2-week length, where we also have a meeting with our project director in order to see how the project is going.

After the end of a sprint, we will do the retrospective meeting to see what we have done wrong and how we can improve for the next sprint.

We also divided all the features explained previously in user stories prioritized them and put them in the Backlog.

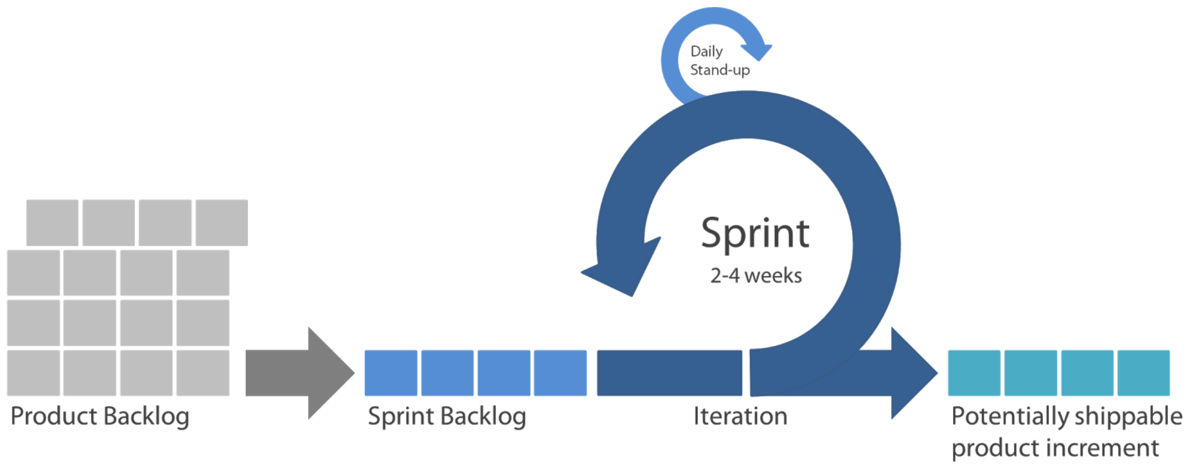


*Figure 1: SCRUM methodology*

#### Tools to monitor evolution

#### Trello

As we are going to use scrum we are going to have our Backlog in Trello which is a web base management application that let us put our backlog into a board and divide the stories in different groups so we know every time what is done, what we have to do and what is in progress.

*Figure 2: Trello logo*

#### Github

As this is a project in pairs we are going to use a version controller which is going to be Github, that let use share the code we are doing and have the most updated version every time.

*Figure 3: Github logo*

### Skype

And to do the retrospective meetings or daily meetings we are going to use Skype to know at every moment how the project is going and to help each other if we need to.

*Figure 4: Skype logo*

### Validation process

In order to know we are on the correct way we will make biweekly meetings, and try to correct what is wrong as soon as possible. At these meetings we will also try to solve the doubts we have.

Also we can talk about what we have done by mail at every time and we will grant access to the director to Github in order to see how the project is going.

## Sustainability

It is very important for every project to check all the dimensions of sustainability such as economic, social and environmental. We need to measure its impact in order to know the project viability.

### Economic dimensions

As we can see in the project cost section, we considered both human and material resources. As the major part is included on the human resources, we can assume that the other costs are minimum and acceptable, and that the project is going to be competitive and viable in this aspect.

We also have to consider that the agile methodology is very effective in the economic dimension. As we can adapt to the necessities of our project while developing it, we will be able to adjust the economic costs to the project and decide if it is going accord to the plans and the viability of the project.

We think that the cost overall of developing this videogame, adding that we are getting information of what a videogame developed with unreal engine could possibly cost, and the limits of this, it's a reasonable price for the value added to the company.

Once talking about the life cycle, we believe that is not going to have a big impact in an economic aspect, because the data we want to store on servers is not that much, and having a maintenance is the minimum that you need to hold this project alive.

About the income, we believe that if the project has a positive result our company will improve the quality of their videogames, so at a long term it is going to have a positive economic effect.

### Social dimensions

Currently, the videogame sector is a bit focused on single player or multiplayer games that have the same role between all players. We think that adding this innovation to a videogame can help the industry to develop new kind of games that add playability to players and make the games more interactive with users.

Also bringing a new engine to our company is very important to diversity so users can choose which kind of games they prefer (implying that the results have notable differences) and that the market opens its limits into a more competitive market where more options are brought to the users.

## Environmental dimensions

As our project is a software system, there will be no environmental impact when implanting it into the society. However, we could consider the electricity of the computers when developing such project and the environmental impact that can have. Anyway, as we are only using 2 computers for the project, that are usually used for other reasons (personal computers), we can assume that this will have no real impact in the environmental aspect and develop our project without having too much consideration in this.

Finally, this project is going to lead into a lot of more projects developed with unreal engine (if the project results successful) so it is a good investment and will make future projects less expensive in term of environmental impact.

## Sustainability matrix

| | **TFG** | **LIFE CYCLE** |
|---|---|---|
| **Economic** | 10 | 15 |
| **Social** | 5 | 20 |
| **Environmental** | X | X |
| **Range** | 15/20 | 35/40 |
| | 50/60 | |

*Table 10: Sustainability matrix*

In this matrix, we can observe that the values given are acceptable. We think that the life cycle of the project has a very good impact in the social dimension because of the multiplayer 2-rol game, even that maybe is not contributing so much at the PPP (TFG in this case) phase.

## Design and implementation

At this section we are going to explain how we have designed the different parts of the project and how we implemented them as well.

We are going to talk about our game architecture and how we have structured our code, but first of all there are some key concepts that we needed to investigate and understand, because they are the main skeleton of every unreal engine project, and will be the base of the architecture.

### Unreal architecture

When programming in C++ you should do different modules, which are going to contain different classes that have their functions and properties. The different classes can also contain other elements like structs, which help with the organization and manipulation of a set of properties.

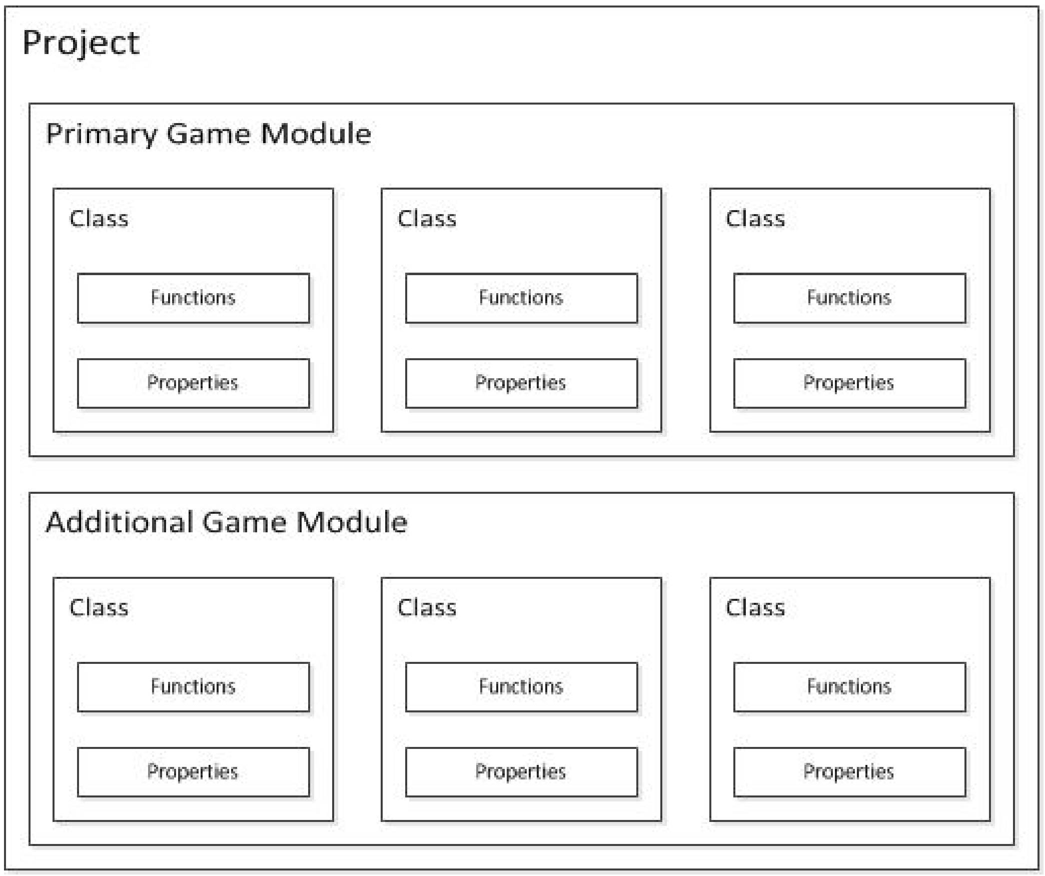


*Figure 5: Unreal architecture*

## Unreal Engine main classes

### Object

As in other programming languages "object" is the base class for almost everything, and Unreal Engine it is not an exception. It is the base class for the rest of Unreal Engine classes. It cannot be added to world (cannot be spawned or placed in level) but it can have data and functions.

### Actor

Actor is the base class for an Object that can be placed or spawned in a level. Actors may contain a collection of Actor Components, which can be used to control how actors move, how they are rendered, etc. The other main function of an Actor is the replication of properties and function calls across the network during play.
Actors inherit from Object so obviously Actors have additional functions and properties.

### Pawn

Pawn is an actor that can be possessed by a Player Controller or AI Controller. All the interactions that a determinate actor can do with the world (collisions, mesh, etc) are represented by the Pawn. Each Pawn is controlled by a Controller, and every Controller only controls a Pawn at any given time.
There is a Default Pawn that is assigned automatically if you don't specify which kind of Pawn you want to spawn and has the basic components of a Pawn.

### Character

Character is a pawn but with a Movement Component, a Capsule Component and a Skeletal Mesh Component added. Thanks to it you can add Animations that uses a Skeleton, you can use Collision with other objects in the world, and run, fly, walk or jump using Physics, or use Navigation to move around the map.

## Player Controller

It represents the human player will. One thing to consider when setting up your Player Controller is what functionality should be in the Player Controller, and what should be in your Pawn. It is possible to handle all input in the Pawn, especially for less complex cases. However, if you have more complex needs, like multiple players on one game client, or the ability to change characters dynamically at runtime, it might be better to handle input in the Player Controller.

Another important aspect to consider is that the Player Controller persists throughout the game, while the Pawn can be transient. For example, in death match style gameplay, you may die and respawn, so you would get a new Pawn but your Player Controller would be the same. In this example, if you kept your score on your Pawn, the score would reset, but if you kept your score on your Player Controller, it would not.

## AI Controller

Like player controller, it can possess Pawns and Characters but it does not have access to Player's input. Instead, it has access to all AI tools like Behaviour Trees, sensing components etc…

## Game Instance

Game Instance is the best class for storing global data. This class will be spawned at game start and deleted when closing game.

## Game Mode

The most important thing of the Game Mode is that only the server has access to it. Game Mode stores data for Default Pawn, Default Player Controller, Default HUD class and Default Game State. The Game Mode is the place where game rules are defined to know when game is over or when a certain player has won or not.

## Game State

Game State objective is to track the progress of the Level and everyone has access to the Game State

### HUD

This is the main class to create the user interface (UI). It can store references from the UMG widgets you have created. Unreal Motion Graphics UI Designer (UMG) is a visual UI authoring tool, which can be used to create UI elements such as in-game HUDs, menus, or other interface related graphics you wish to present to your users.

## Network

Network is another important aspect we wanted to get into it and learn about it. We can clearly say that this is the part where we have had more troubles because we never have done anything multiplayer related and we have some problems finding examples of multiplayer games in C++.

Despite this, here we are going to explain how networking works in Unreal Engine. It uses a Server-Client architecture. This means, that the server is authoritative and all data must be sent client to server first.  A simple example of it is moving a character, where you do not move your character; instead, you tell the server that you want to move it.

One of the reasons of this architecture is basically never trust the client. Trusting the client means that you do not test their actions allowing them to cheat which is the main thing to avoid on multiplayer games. The simplest example in our case could be that if we do not test if the player has ammo on the server, every client could cheat having infinite ammo, which would mean a game without any fun.

### Server-Client architecture

We can split the architecture in four different sections:

- Server Only: These objects only exist in the server
- Server and Clients: These objects exist in both, server and client.
- Server and owning client: These objects only exist in the server and the owning client, not in all of them.
- Owning client only: These objects only exists in a single client.

In the following picture, we can see how the most important classes are laid in the network framework:

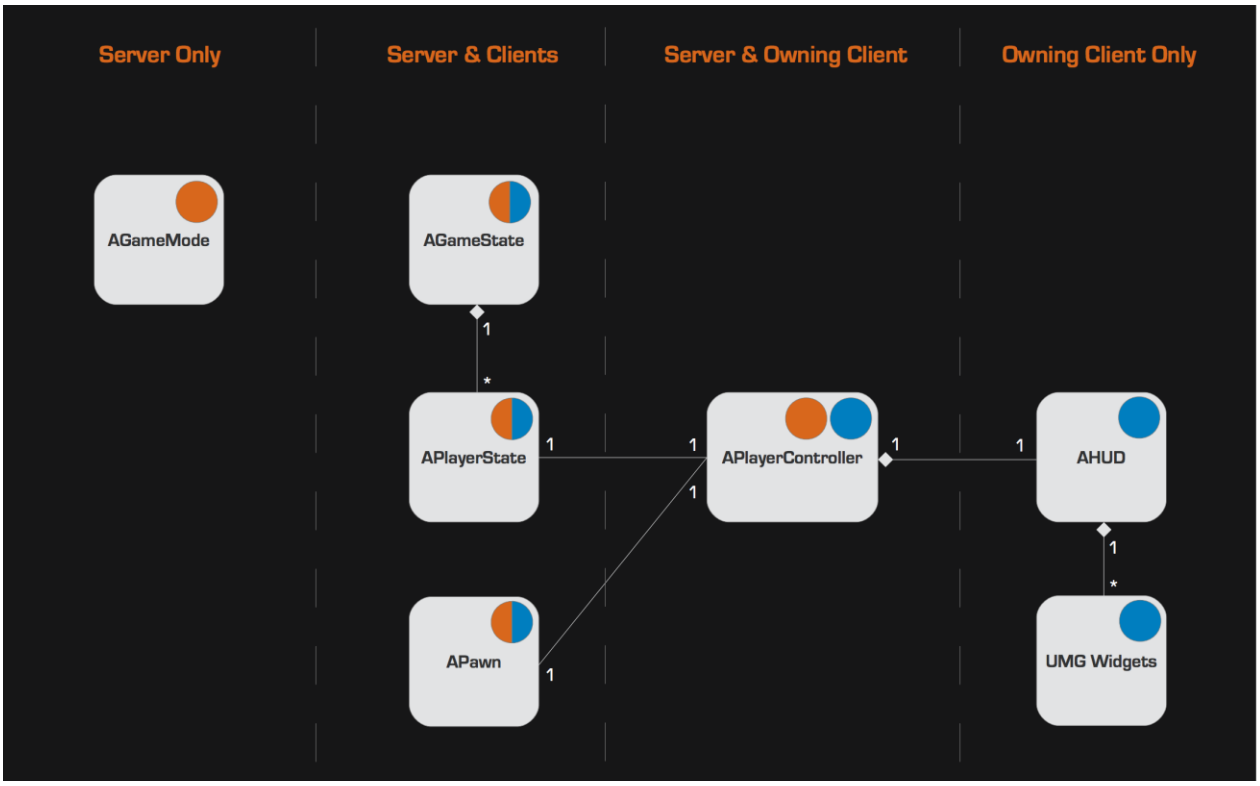
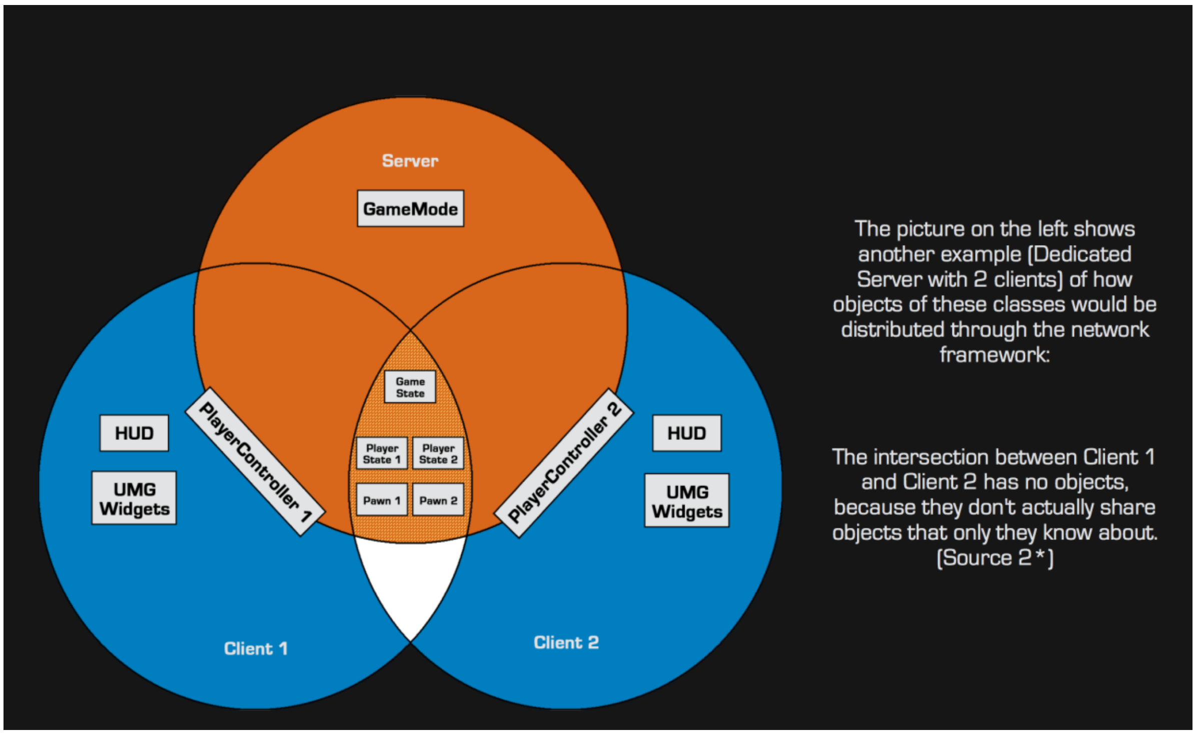


Figure 6: Classes in a client-server architecture [1]

In this image we can also see how they are divided in the case we have 2 clients:



The picture on the left shows another example (Dedicated Server with 2 clients) of how objects of these classes would be distributed through the network framework:

The intersection between Client 1 and Client 2 has no objects, because they don't actually share objects that only they know about. (Source 2 *)

Figure 7: Class execution in dedicated server [1]

In the previous images, we have seen which classes are used in networking and how they behave. But, let’s get a bit deeper and explain what is the purpose of every class and an example of it.

## Game Mode class

The class Game Mode is used to define the rules of the game. This includes the used cases like Pawn, Player Controller, Player State… An important thing is the fact that this class is only available in the Server. This means that clients do not have a game mode object and they will only get null pointer if they try to retrieve the game mode.
Common game modes examples in shooter games are the death match, team death match, or capture the flag.

## Game State class

The Game State class is probably the most important class for shared information between Server and Client. The Game state is used to keep track of the current state of the game. This include for example a List of connected Players.
The game state is replicated to all clients so everyone can access to it. This means that the game state is one of the most entered classes for multiplayer games.
For example, if the Game Mode has the number of kills that needs to be done in order to win, the Game State would be the one who keeps track of these kills. The information you need to store in the Game State is completely up to the game needs and the way to store them as well.

## Player State class

The Player State class is one of the most important classes for a specific player. Each player has his or her own player state. The player state is replicated to everyone and can be used to retrieve and display data on other clients. An easy way to access the Player State is to access the player array inside the game state class. The normal information you want to store in the player state can be the player name, the score it has, the ping, or any replicated information that other players might need to know about.

## Player Controller class

The player Controller class is the most interesting and complicated class. It is also the centre of a lot of Client stuff since it is the first class that the alien 'owns'. It can be seen as the input of the player, the link of the player with the server… This means, every Client has one Player Controller and it only exists for the client and the server, but other clients do not know about other player controllers. The result of this system is that the server has a reference of all client Player Controllers.

## HUD class

The HUD class it is only available in the Client and can be accessed through the Player Controller. It also will be an automatically spawned. It is mainly used to draw text, textures and more in the viewport of the Client. However, after the UMG (unreal motion graphic) appearance the HUD class is not as used as it was before. By now, widgets replace the HUD class most of the times. Widgets are only available in the locally client and are not replicated over the network.

## Dedicated Server vs Listen Server

Here we are going to compare the two available options when we attempt to develop an online video game.

**Dedicated Servers**

A dedicated server is a standalone server that does not require a Client. It runs separated from the Game Client and it is mostly used to have a server running that players can always join / leave.

Dedicated servers can be compiled for windows and can be run on virtual servers that players can connect to via a fixed IP-Adress. It does not need UI part or player controller since it is a standalone server with our clients.

**Listen servers**

Listen server is a Server that is also a client. This means, that the Server always has at least one Client connected. The client is called listen server and if he disconnects all the server will shutdown. Due to being also a Client the listen server needs the UI and the player controller which represents the client part. Since listen server is also the client, every player that wants to connect to it will connect to the client IP. This can carry some problems such as not having a static IP might be a problem but using OnlineSubsystem module can solve this problem.

Considering these two option we finally decided going for a dedicated server in our game because having a client as a server may cause problems in our game if for example the “server client” disconnects for any reason. This would cause the abort of the game and we want to prevent this to happen.

## Replication

Replication is the act of server passing information / data to the Clients. It can be specific to a group of entities. The first class capable of replication is the actor, and as all mentioned class explained before inheriting from actor there is no problem at replicating properties of these classes. We also have to consider that classes such as Game Mode do not need to replicate at all because they only exist on the server.

## RPC

Remote Procedure calls are functions that are called locally but executed remotely on another machine (separate from the calling machine). This kind of function allow the client and the server to send to each other over the network.
To declare a RPC function that will be called on the server but executed on the client, we have to do this:

```
UFUNCTION (Client)
Void MyFunction();
```

An RPC function that will be called on the client, but executed on the server is very similar to the previous one, it just uses Server keyword instead of Client.

```
UFUNCTION (Server)
Void MyFunction();
```

We can notice that we can see the UFUNCTION before the method declaration. What does it mean? This is a way for Unreal Engine to connect the C++ code with the Engine itself so they have defined different macros like UFUNCTION or UPROPERTY in order to connect the Editor with the C++ code. It

also helps because for example if you declare attributes with UPROPERTY() the garbage collector takes care of this attribute when it is used.
There is another type of RPC function type called multicast. Multicast RPCs are designed to be called from the server, and afterwards executed on the server as well as all currently connected clients.

By default, RPC functions are unreliable so to ensure that an RPC function is executed on the remote we need to mark it as reliable this way:

```
UFUNCTION (Client, Reliable)
```

Using this kind of function has some requirements that we need to follow.

- They must be called from Actors
- The actor must be replicated

Now we have explained the basic thing to know let explain what we have done and how.

First of all, these are the different tasks we had:

Felix Hortelano:

- Handmade maze.
- First player.
- Gold system.
- Database.
- Sound.

Guillem Bonafonte:

- Auto generated maze.
- Second player.
- Level system.
- Multiplayer.
- Graphic and User Interface.

After finishing the project, the final tasks have changed and we will explain the reason of it in the Deviations part. In order to know what everyone has done the final tasks are:

Felix Hortelano:

- Handmade maze.
- First player (shooter).
- Sound.
- Multiplayer.
- Artificial intelligence.

Guillem Bonafonte:

- Second player.
- Ability system.
- Gold system.
- Graphic and user interface.

# Felix Hortelano’s work

## Shooter character

The shooter character is one of the 2 type of players our game is going to have. This character as his name says is going to be of type shooter. This means it is going to have different weapons and he is going to be able to shoot, target and the normal things every shooter game have. We decided to go with these characteristics because shooter games are well known in the industry and we can give our company a better knowledge how hard or easy is do a shooter game with Unreal Engine.

When you start a project with Unreal Engine, you have the choice to select existing templates, which grants you some work done. For example, in this case there were 2 interesting templates to go with. The first one was the third person template, where Unreal gives you a basic third person character, which can move around the scenario. The second was the first-person template which is the same as the third but with the difference of having a first person mesh instead of a third person one. Once we thought about which template start with we realised there was none which really fits our needs because if we are going to do a multiplayer game we are going to need both views. The first person for the player who is going to possess the shooter character and the third person, which is the mesh that the other player is going to see when, sees the shooter character. Once we knew this, and after a bit of research of how games do this kind of behaviour we decided that our shooter character is going to have both meshes, the first-person mesh and the third person mesh. And to allow only the correct players to see it we are going to use the visibility of the mesh which is an option that allows to hide or show the mesh only to certain actors or for every actor, as the developers wants to.

In order to do our shooter character, we decided that his parent class was character so we can use all parent class method we needed in order to do the player.

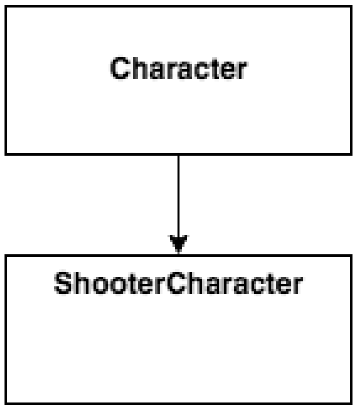


*Figure 8: Shooter character parent class*

Another option was use pawn or actor as parent class but as you can see in unreal engine main classes section using the Character class has more sense for this kind of actor.

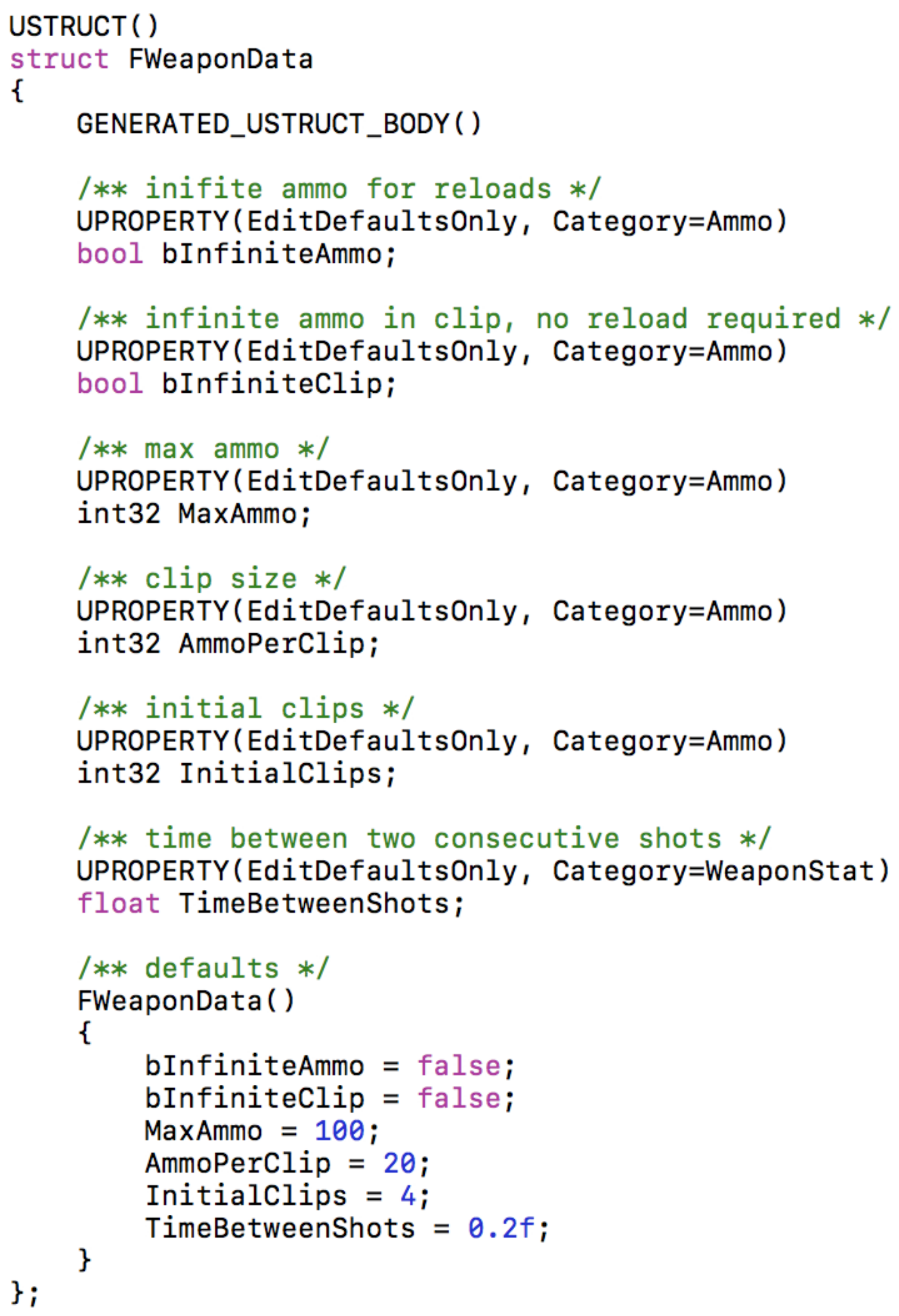
## Shooter Weapons

A weapon system is another important aspect, which we focused on study in this project. In order to archive this option, the best way possible we thought about a system, which allows adding new weapons at every moment. That is why we defined a base class with some parameters that will allow to instantiate any new weapon we create at any moment base on some common parameter in any weapon system. These common parameters between all the different weapons are:

- Ammo: the amount of ammunition every weapon is going to have. Every type of weapon is going to have a certain ammo.
- Ammo per clip: amount of ammunition per clip, that is another common asset in weapons they are divided in clips and every clip has an number of bullets.
- Initial clips: in order to define the initials clips of every weapon
- Time between shoots: The time the weapon has between shots. We considered is not as fair shooting the same number of bullets from a grenade launcher than from a fusel. That’s why we implemented this timer in order to balance a bit the weapons.

We also defined some special properties in order to test the game properly. An example of these properties is infiniteAmmo, or infiniteAmmoInClip this helped us to test the game without having to restart every time we are out of ammo.

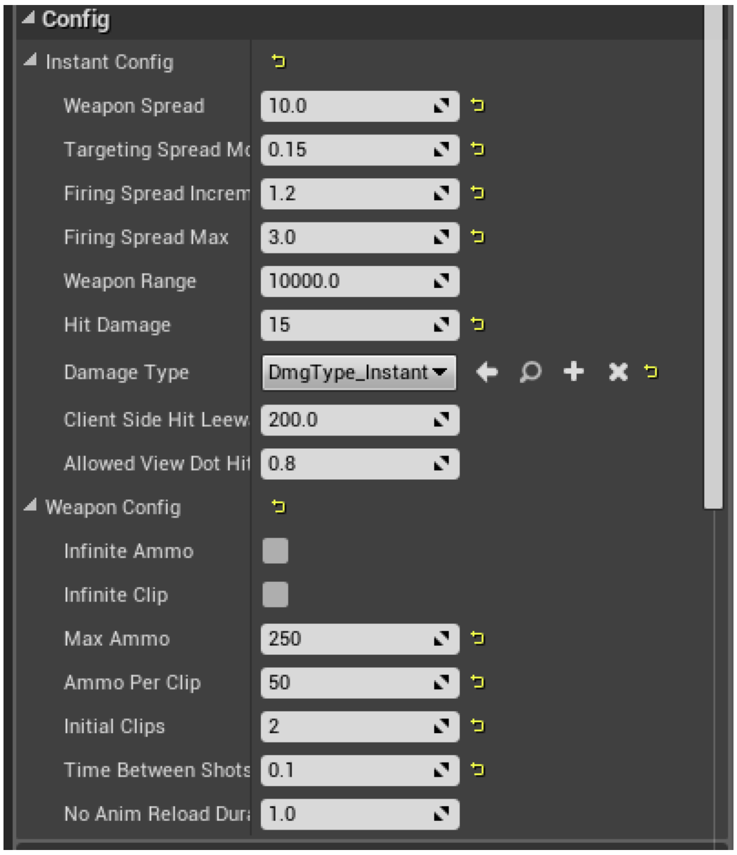
In the following picture, we can see the structure we defined in our ShooterWeapon class in order to use it on every new weapon. We can see different things in this image. First, we see the macro USTRUCT() this is telling Unreal Engine Editor to show the struct information in properties, doing that we allow every designer to create a new Blueprint based on our C++ class that will have our USTRUCT, so the designer only has to change the properties according to the new weapon he is creating.



```
USTRUCT()
struct FWeaponData
{
    GENERATED_USTRUCT_BODY()

    /** inifite ammo for reloads */
    UPROPERTY(EditDefaultsOnly, Category=Ammo)
    bool bInfiniteAmmo;

    /** infinite ammo in clip, no reload required */
    UPROPERTY(EditDefaultsOnly, Category=Ammo)
    bool bInfiniteClip;

    /** max ammo */
    UPROPERTY(EditDefaultsOnly, Category=Ammo)
    int32 MaxAmmo;

    /** clip size */
    UPROPERTY(EditDefaultsOnly, Category=Ammo)
    int32 AmmoPerClip;

    /** initial clips */
    UPROPERTY(EditDefaultsOnly, Category=Ammo)
    int32 InitialClips;

    /** time between two consecutive shots */
    UPROPERTY(EditDefaultsOnly, Category=WeaponStat)
    float TimeBetweenShots;

    /** defaults */
    FWeaponData()
    {
        bInfiniteAmmo = false;
        bInfiniteClip = false;
        MaxAmmo = 100;
        AmmoPerClip = 20;
        InitialClips = 4;
        TimeBetweenShots = 0.2f;
    }
};
```

*Figure 9: Weapon Data*

In this image, we can see how all these attributes are reflected in the blueprint so the designer can easily modify its values.



Config

| Instant Config | |
|---|---|
| Weapon Spread | 10.0 |
| Targeting Spread Mo | 0.15 |
| Firing Spread Increm | 1.2 |
| Firing Spread Max | 3.0 |
| Weapon Range | 10000.0 |
| Hit Damage | 15 |
| Damage Type | DmgType_Instant |
| Client Side Hit Leew | 200.0 |
| Allowed View Dot Hit | 0.8 |
| **Weapon Config** | |
| Infinite Ammo | ☐ |
| Infinite Clip | ☐ |
| Max Ammo | 250 |
| Ammo Per Clip | 50 |
| Initial Clips | 2 |
| Time Between Shots | 0.1 |
| No Anim Reload Dura | 1.0 |

*Figure 10: Weapon data in the editor.*

If you look carefully the previous image you probably notice that there is more configuration which we have not talked about yet. We have explained the weapon config but what it is instant config? Well the problem appeared when we were doing the grenade launcher because it does not behave like a normal fusel it uses projectiles and they have things such as explosion damage, which other weapons does not have. So that we came with the idea of use, our base class ShooterWeapon in order to categorise a bit more what kind of weapon it is. Therefore, we did this categorisation in two categories instant weapons and projectile weapons so finally out weapon architecture is going to look like:

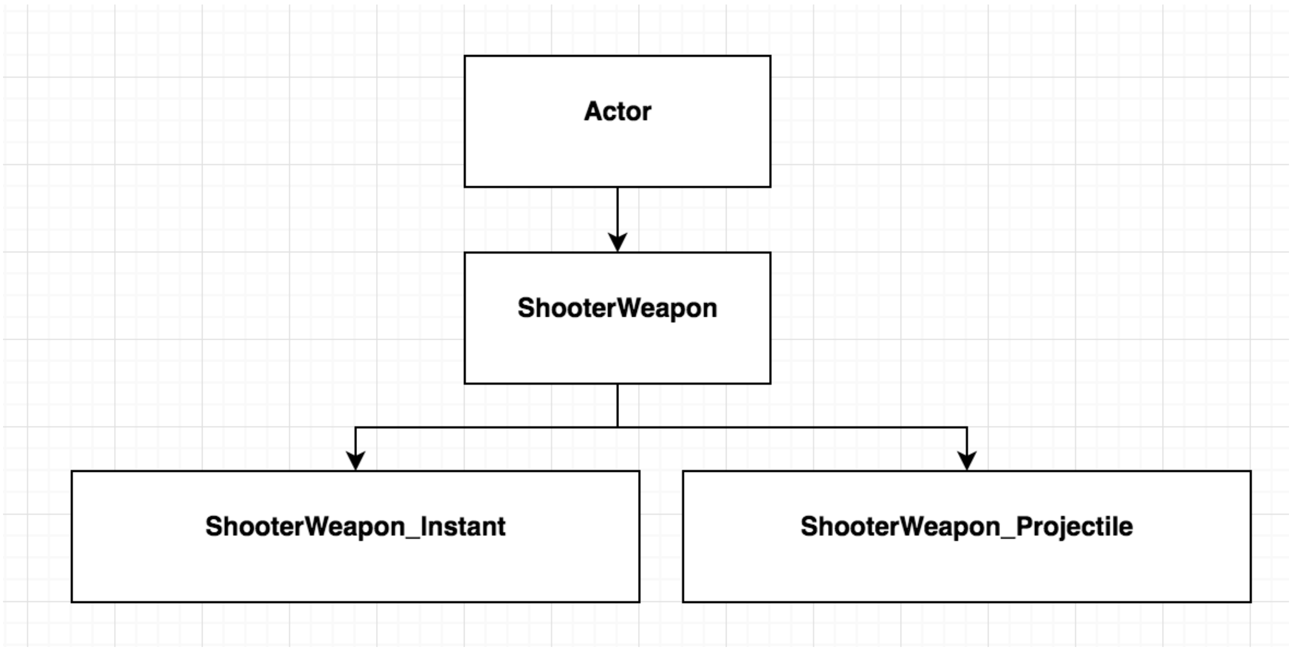


*Figure 11: Weapon types*

Weapon system is not only about defining a struct for weapon properties and after that, you are done. There are many more important things to do. For example, each weapon can have his custom animations, sound. You have to handle the different states of a weapon and that is another thing we have done. We have defined weapon states to know at every moment in which state is the current weapon and what we have to do next.

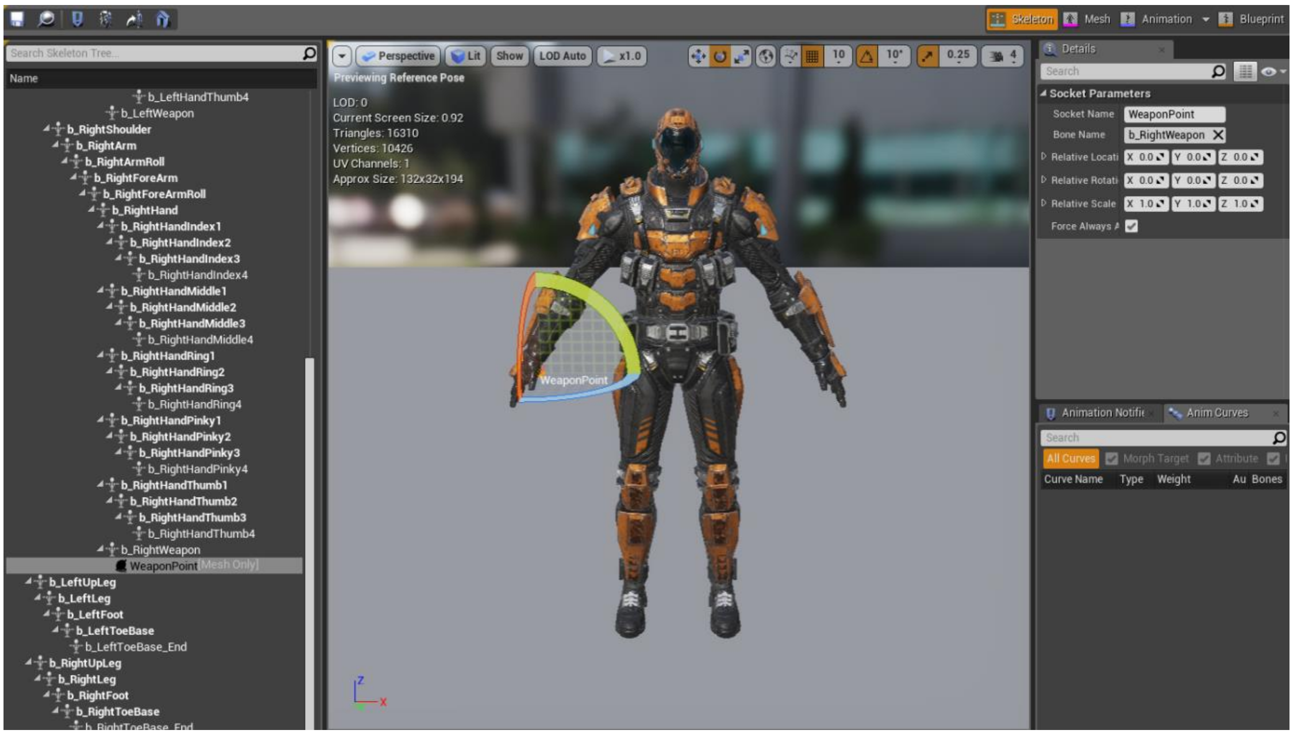
In this picture, we can see the different states every weapon is going to be.

```
enum Type
{
    Idle,
    Firing,
    Reloading,
    Equipping,
};
```

The idle state is the default one but after that state others can happen such as firing or reloading if you are out of ammo or simply because player want to reload. In addition, the last state is equipping since the player can have two different weapons and he is able to equip one or another.

After this explanation is left explain, which parent class has our ShooterWeapon and the answer is an Actor. Actor class is the best option to do weapons because they are attached to a skeleton, which it is defined in the Character class.

Attaching a weapon is done by creating a socket in a determinate bone of a mesh, so once we have the sockets we can attach the actor we want to this socket. This is especially useful in our case because it helps to attach weapons to the correct place.



Previewing Reference Pose
LOD: 0
Current Screen Size: 0.92
Triangles: 16310
Vertices: 10426
UV Channels: 1
Approx Size: 132x32x194

Socket Parameters
Socket Name: WeaponPoint
Bone Name: b_RightWeapon
Relative Locati: X 0.0 Y 0.0 Z 0.0
Relative Rotati: X 0.0 Y 0.0 Z 0.0
Relative Scale: X 1.0 Y 1.0 Z 1.0
Force Always: ✓

Figure 12: Weapon attach point in skeletal mesh

Finally, after we have the base done we have to search for the weapon models and add it to the game. We decide to add two weapons because of time mainly and because adding more it going to the same as the other weapons. These are the two weapons the base character has:

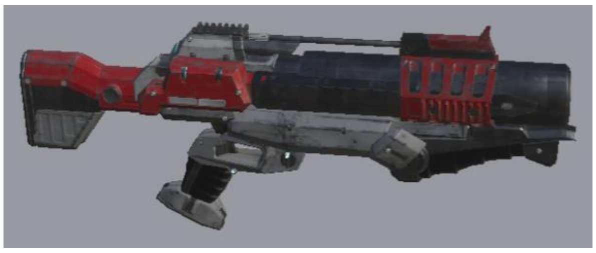


Figure 14: Grenade launcher

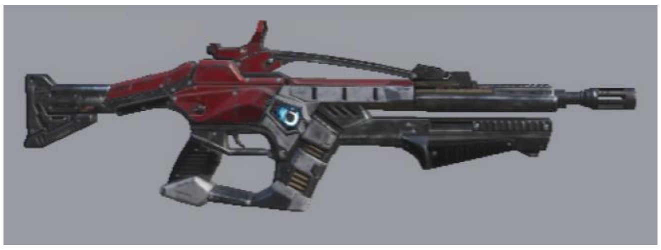


Figure 13: Rifle

## Artificial intelligence

Another important aspect in our game was the AI, as in almost every game is a key part of a game. We have to search how artificial intelligence works in Unreal Engine and how hard it is to. But before starting this point there is the need of explaining a bit some key concepts on this area.

### Blackboards

Blackboard behaves like the artificial intelligence memory. It basically stores values so you can know at every moment its value or update it corresponding to behaviour you have done.

In this image we can see one of our blackboards so you can see that our bot probably will have a destination point which is a vector in 3 dimensions where the bot should go, the enemy which is a reference pointing to the enemy and the selfActor which is a reference pointing to himself.

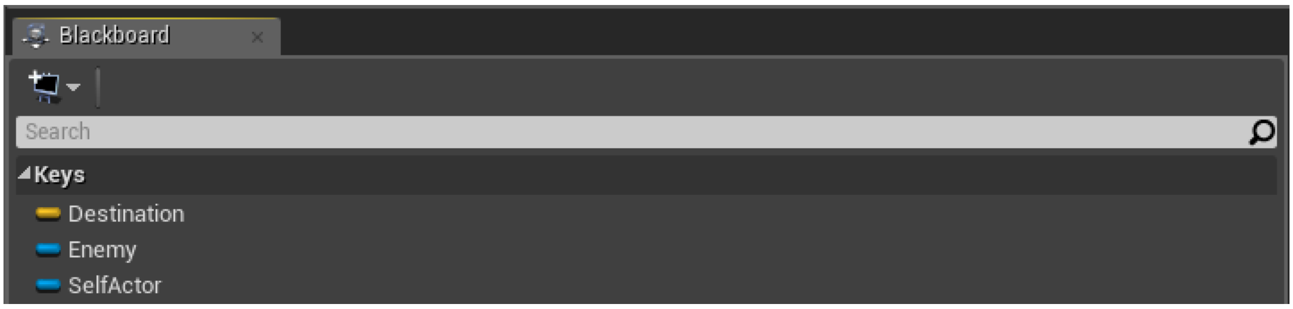


*Figure 15: Blackboard example*

## Behaviour trees

Unreal Engine artificial intelligence is based on behaviour trees. Behaviour trees are like the AI processor. It makes decisions and the acts based on those decisions.

Behaviour trees have different types of nodes that need to be explained in order to understand what we have done.

- Composite: These are nodes that define a root of a branch and the base rules on how the branch is executed.
- Task: These are the leaves of the behaviour trees and the ones, which explain what has to be done. Those does not have an output connection.
- Decorator: Also known as conditions. This are attached to another node and make decisions based on the condition.
- Service: These nodes are attached to composite nodes, and will be executed as long as their branch is being executed. These are normally used to make check or updates in the Blackboard.
- Root: The root node is unique in the behaviour tree and is the starting point of every behaviour tree.

We need to go deeper in Composite nodes in order to explain later what we have done. There are two different type of composite nodes and they do the following:

- Selector: This node executes his children from left to right and will stop executing when one of their children success.
- Sequence: This node executes their children from left to right and will stop executing when one of their children fails
- Simple parallel: Finally, there is this type of composite nodes, which we did not used but we though it is worth to know anyway. This node allows a single main task to be executed alongside of a full tree. When the main task finish there is a parameter called Finish Mode that dictates if the mode should finish immediately or aborting the secondary tree.

Now we know a bit more about behaviour trees nodes we can understand what the meaning of this behaviour tree, which corresponds, to our bot behaviour.

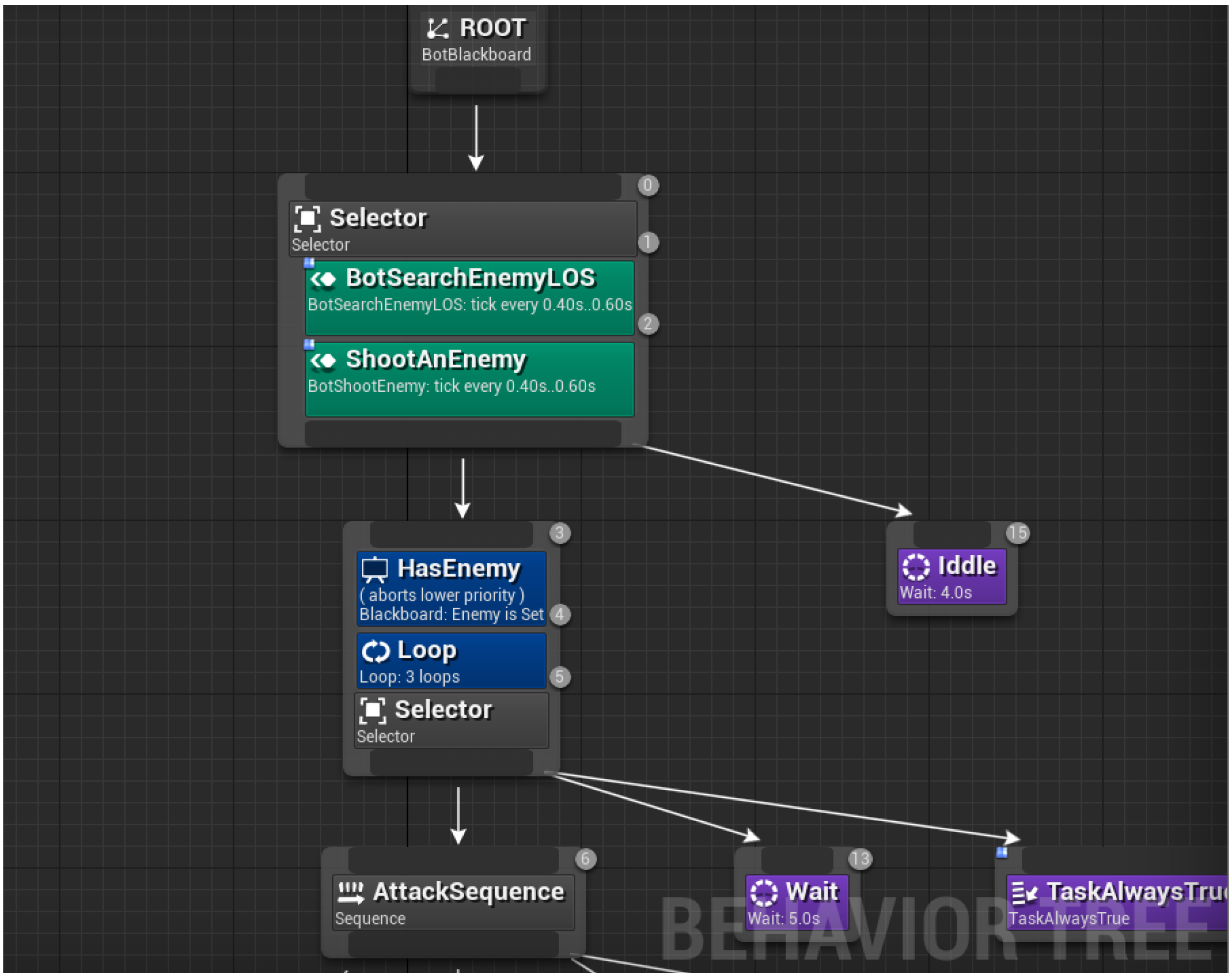


*Figure 16: Behaviour tree example.*

In the previous image, we can see in green colour all the services our application has. With blue collar are all the decorators and with purple all the task. Even in the image, we do not see the complete behaviour tree we can understand how this works.

Our services function is to update our blackboard components in order to know if there is an enemy around and to shoot him if there is the enemy. We also added an idle task, which is a wait node of 4 second in this case to avoid the enemy shooting you without you cannot do anything to kill them.

The first child of the selector node we can see it has two decorators, which are conditions for its node execution. The conditions are loop if there is an enemy if there is not the node will not be executed. After that, we do a wait of 5 second for the same reason as the idle in the previous level.

The rest of the behaviour tree follows the same idea as what we have seen until now, it is based on decorators and services to know when we have to do certain things or not.

## ShooterAIController

While Player Controller has access to player input with our artificial intelligence, we do not need this input anymore that is why our artificial intelligence class inherits from AIController.

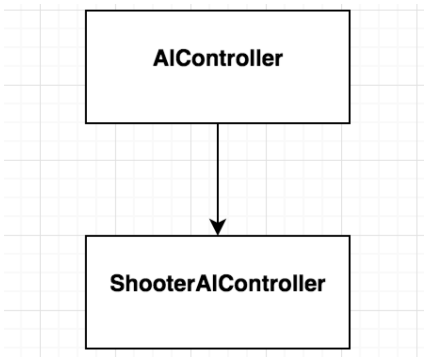


*Figure 17: Shooter AI Controller parent class*

The function of this class is to control the AI behaviour and to allow out bots attack other people, found them and update blackboard values in order to know what is going on at every moment.

## ShooterBot

This is the class that controls Bots which basically inherit from ShooterCharacter and add different functionalities that a ShooterCharacter does not have. One of these characteristics is the use of AIPerception, which is a module that allows AI to have sense and use it as they want. One example of this sense use is the hearing or the vision, we can allow our pawn to see or hear in a certain radius and we can activate events when they receive any kind of stimulus. With this we can do things such as make a bot follow us until there is some enemy in his radius of vision so the bot starts to defend us from dying. Another interesting thing we can do is to make bot patrol in a route and attack us if they are able to see or hear us.

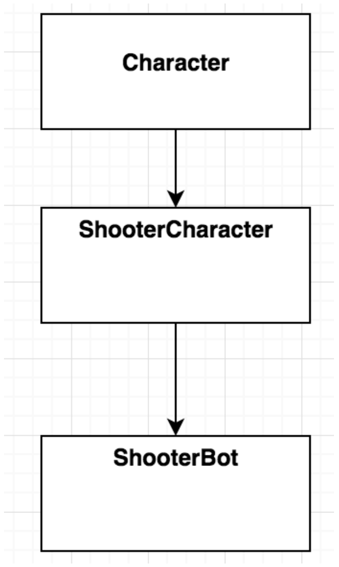


*Figure 18: Shooter Bot parent class*

## Escape The Maze Game Mode

This is our Game Mode, as we already explained before that the game mode is the class where you define the rules of the game and it is only executed in the server we now are going to explain what our Game Mode does.

The Game Mode class is very important as we already explained but in our case, it is a bit more. As it is the only class executed exclusively in the server here, we had to decide how every client behaves. In our case, this means that we have to say to each client, which Pawn is going to possess because we have two different players, and each client in the current game possesses one.

This class also handles different things such as the state of the game. Game Mode class in Unreal Engine contains a state machine to track the state of the current match. There is different function we can use such as GetMatchState or some wrappers like HasMatchStarted, IsMatchInProgress and HasMatchEnded. These are the possible match states:

- EnteringMap: The initial state where actors are not yet ticking and the world is not initialised.
- WaitingToStart: HandleMatchIsWaitingToStart is called when entering. Actors are ticking, but players are not yet spawned in. It changed to next state if ReadyToStartMatch returns true or if we call StartMatch.
- InProgress is the state where the main part of the game will take. HandleMatchHasStarted is called when entering. Then calls BeginPlay on all Actors. The match will change the state when ReadyToEndMatch is true or EndMatch is called.
- WaitingPostMatch is the second-to-last-state and HandleMatchHasEnded is called when entering it. It transitions to the next state when map transfer start
- LeavingMap is the last state in the normal flow and calls HandleLeavingMap upon entry. After this state the process starts again and EnteringMap is called.
- Aborted is the failure state and can be state calling AbortMatch as well.

There is also a couple of interesting functions in the game mode that we have done in order to handle the correct behaviour of the game. One example is the Killed function or the ModifyDamage function. The first one notifies the player state that something has happened, the kill of someone in this case. In addition, ModifyDamage is done to avoid friend firing so we cannot kill our friends if we shoot at them. It just sets to zero the amount of damage taken by a bullet if it has been fired from an ally.

We have also added a timer which Game Mode updates even it is stored in the Game State class because it is where game data is stored.

### Shooter Character Player State

Player state is useful to store data and this is how we use it. At this class, we store things such as the team of each player / bot on AI, the number of deaths each player has, number of kills. Number of bullets used in order to make a precision score I we wanted to. Or the team colour to differentiate between teams.

### Shooter Character Game State

Game State is the class where we store information between the Server and the Client. Here we just store variables such as the number of teams or the remaining time in order to know when the game has ended.

### End trigger

This class is an actor that ends the game when the shooter character enters in. For this, we use the function NotifyActorBeginOverlap, which is an event that is fired when an actor overlaps with this actor. Once this function is triggered, we call the end game function to finish the game,

### Sound

In order to add sound to the player actions such as firing or running we have used Sound Cue, which is anode-based editor that is used to work with audio.

## Guillem Bonafonte’s work

## God Character

As we explained before, our game is going to have 2 types of players, the Shooter one (explained above) and the God player. We decided to name it this way because its characteristics: it has no health, its invisible, and it does not move by walking, but flying.

The first time we created this player, we were extending from APawn. The movement of the default Pawn was perfect for the behaviour we wanted our player to have, so starting from there was a big help.

The problems appeared when trying to replicate the God Character so it could work in a multiplayer game, not only at local. At that moment our God Character could move around the world and place mines at his line of sight. When we replicated this character, and tried to run it at the server, mines were only being placed at the line of sight where the character started. So obviously there was something going wrong, and it was related with the movement. After trying how we could replicate the movement on the server we decided to go for another path. Shooter character was being correctly replicated so, why don't follow the same steps? We found out that the reason why the Shooter character had movement replicated correctly was because it extended from Character.

Right now, our God Character extends from character, but we modified all the character behaviour and tried to simulate the same behaviour that the Default Pawn has for movement. Obviously, the final movement of the God Character is not exactly the Default Pawn one, this is because there are some components that need to be different to keep replicating the movement. For example, the following line replicated completely the movement of the Pawn, but obviously then the character was not replicating correctly because it doesn't recognize the Movement Component.

```
MovementComponent = CreateDefaultSubobject<UPawnMovementComponent,
UFloatingPawnMovement>(ADefaultPawn::MovementComponentName);
```

## Ability System

Once we had the movement completed, we started with all the abilities we wanted the God Character to have and started to replicate them correctly. To achieve that we needed an Ability System, some way to manage all the abilities and to treat them.

We started creating a Parent class called "Ability" that have all the properties that an Ability needs. The 2 functions that do the raycasting to detect where is the God player looking at, the Deploy function that defines the behaviour when you want to spawn that Ability, and all the properties that help the Ability with the communication with the player (Gold cost, Timer/Cooldown...) or the properties that store where the deploy has to be (position, rotation, etc).

We set the Gold Cost and Timer as an "UPROPERTY(EditDefaultsOnly, Category = "Info")", so you can modify them at Blueprints and there are not only hardcoded variables.

Once the Ability Class was finished, we did the 3 Abilities we wanted to have: The Mine, The MinionSpawn and the FakeDestiny. The 3 Abilities work in a very similar way so most functions from the parent were used and only the Deploy function had to be reimplemented (obviously you need to Spawn different things in each Ability).

Back to the God Character class, we did there an Array of Abilities editable on blueprints. This way you can add on the editor all the abilities you want and dynamically it creates the array with all the instances of the abilities. This way, when firing, you check which is the active Ability at that moment and call the Deploy method of that Ability.

Obviously, some Abilities have an "extra" functionality that doesn't share with the other Abilities. For example, the Mine has a collider that when colliding with a Shooter from the opposite team itself destructs and it takes life away from the Shooter.

The good part of this Ability System is that we could reuse it later for the Shooter Character. Even that the behaviour of these ones is a bit different, reimplementing the "Deploy" function was enough to cover all the functionalities that we wanted.

Basically, what the "Deploy" function does is to Spawn the concrete Ability if possible. Every tick the Character calls the method "InitTheRay" of the active Ability, telling the direction where the player is looking. This sends a Ray that if it Ends on ground, lets the Ability to spawn. In the case of the Shooter Character, as it doesn't make sense to spawn the Ability in that direction, the deploy function just forces the Ability (in our case an ally) to Spawn next to the player.

## Gold System

To make all the Abilities work, and to create some logic to the video game, we needed a gold system, to control that abilities can only be used once in a while so put an extra of strategy to the player, that has to decide which Ability wants to use and how often. As explained before we want to avoid cheating when possible, so the variable that stores the Gold amount for each player is replicated at the server, so the sever have always control of it. The variable we used to store this information was the following:

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Replicated)
        int32 totalGold;
```

We chose these ones because we wanted the gold to be modified not only by code, but also on blueprints. When doing HUD (explained later) we had to use Blueprints, and of course to access to this value from there.

The gold logic was pretty simple. Every tick the player had a timer on-going, and whenever this timer reached 5 seconds, the value of totalGold increased so you could continue playing and don't just get out of gold.

Every time you want to deploy an Ability, you check that the actual gold value is higher than the gold cost of deploying that Ability. If it's true, all the deploy mechanic starts, and if the Ability is deployed correctly, it subtracts you the respectively gold.

## HUD

Every game nowadays has an HUD. The HUD makes the game much more intuitive, and helps the player to check the status of the game in a much easier way (for example the health of his character, the ammo left, etc).

Unreal Engine allows creating an object called “Widget Blueprint” that enables extending from “UUserWidget”. This way a new screen is opened and you get a new Editor Window that helps a lot creating the HUD you need.

In our case we needed 3 types of HUD. The Shooter Character ones, the God Character ones, and all the shared ones. This means, that we wanted some elements to be in both players screens.

**God Character:** For this character exclusively, we wanted one only element. As we said previously the God Character spawns his Abilities at the point where he is looking at, so we thought that a dot at the centre of the screen could help the player to know where is he going to spawn his Ability in case he presses “Fire”.

**Shooter Character:** The shooter character needs also a pointer so he knows the direction he is pointing with the weapon. We did a different type of image because shooters usually have the typical cross and we wanted to follow that pattern. The shooter has as well a bar indicating the health of your character. The god character doesn’t have any health so this element is exclusive of the shooter character. There is an extra element for the shooter character related with the health remaining. When the Health points of the Character go too low, the screen turns a bit red to indicate that you are in a critic state.

**Both Characters:** There is one thing in common for both characters: the gold system. This is the reason we decided to include a common HUD for both players that shows the amount of gold that you have at every moment.

The other element that both characters have is the timer that indicates the time remaining for the end of the game. There is a countdown starting at 5 min at the corner of the screen so the player knows how much time they have to get their objectives.

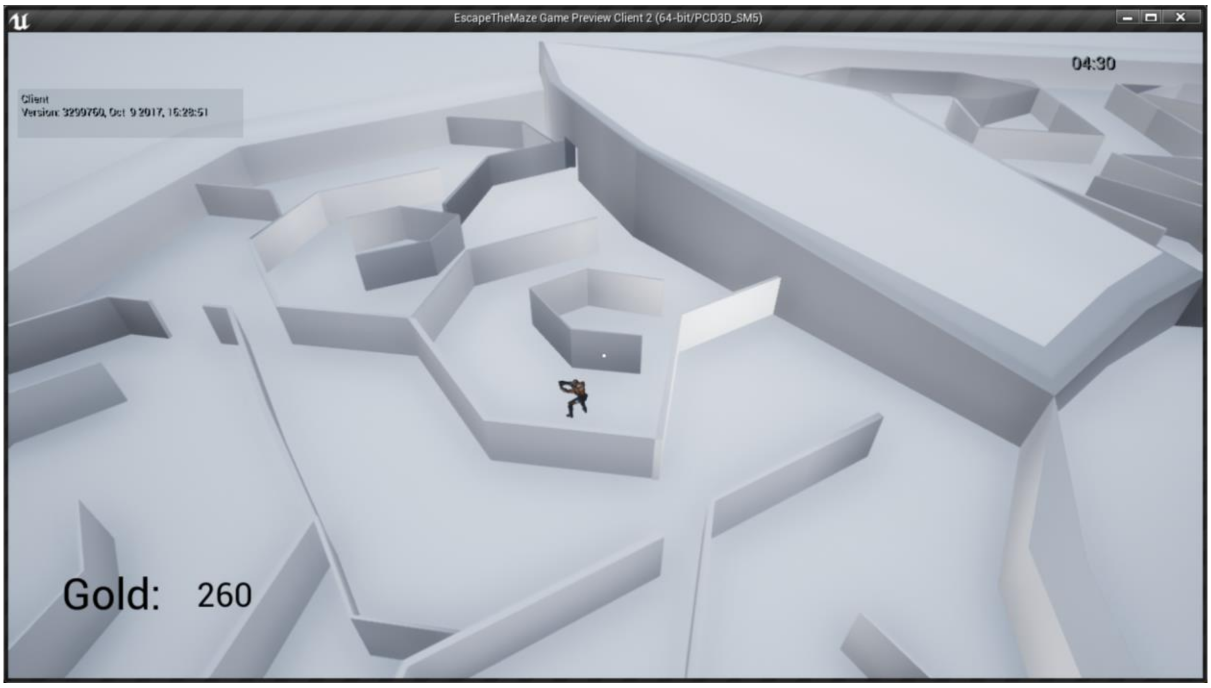

EscapeTheMaze Game Preview Client 2 (64-bit/PCD3D_SM5)
04:30
Client
Version: 3299760, Oct 9 2017, 16:28:51
Gold: 260

*Figure 19: HUD God Character*

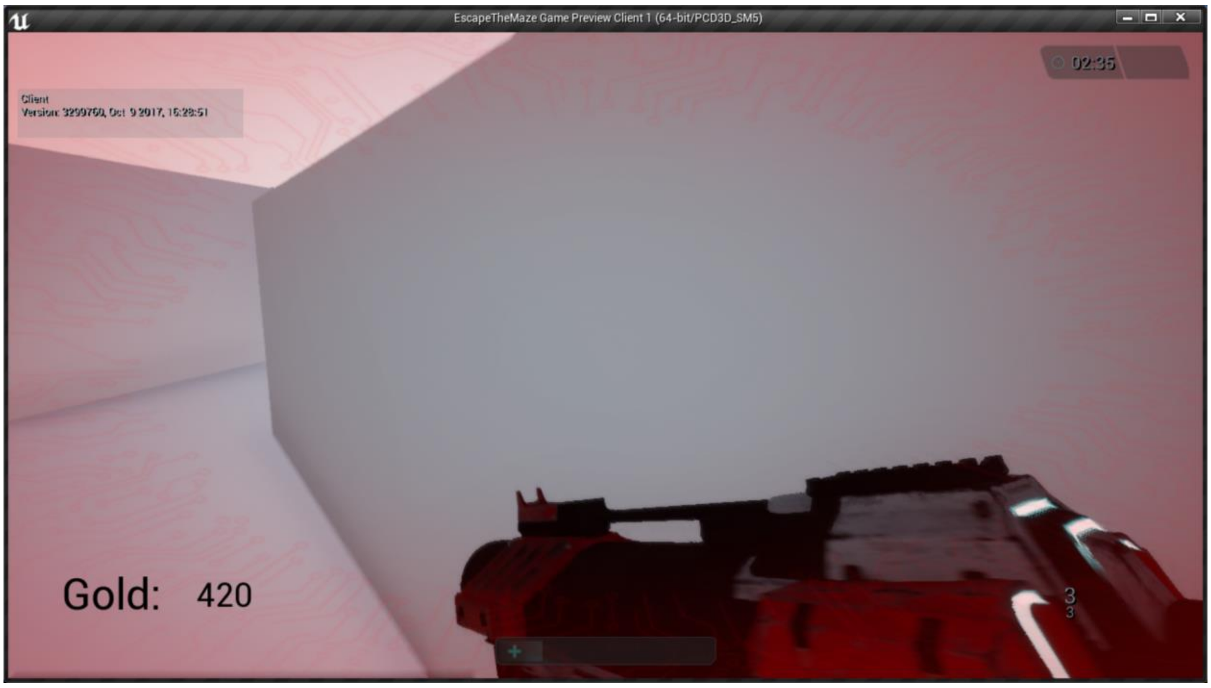

EscapeTheMaze Game Preview Client 1 (64-bit/PCD3D_SM5)
02:35
Client
Version: 3290760, Oct 9 2017, 15:28:51
Gold: 420
3
3

*Figure 20: HUD Shooter Character*

We needed to create some logic so the blueprint decides when to show each element and when not to. The only difference between both cases is the character, so we obtain the player and cast it to God Character or Shooter Character. If the cast fails, you hide that element, otherwise you make it visible.

In the case of the Gold, we want both to show their respectively gold, so the graph cannot be the same or you would see the same amount of gold for both players. In this case what we did was returning the gold value of the character after casting to that concrete character. Obviously, both cast cannot be true at the same time, so the result will be only the gold of the character that the player is controlling.

To change and explore other ways to crate the HUD, we implemented the health and ammo of the Shooter Character with code instead of blueprints. We found this way a bit more complex because you needed to look at a lot of examples to know how to place every element in the specific place, to add the icons, etc. And once it is done is harder to change small details and adapt it to your new needs.

## Work done but not included

During all the development process of the project there are several things that we have done in order to learn Unreal Engine or even things we initially though they were useful for the game but at the end have not been included in the project.

### Login system

Even we have not done the database task, it is not included in the final project, we look for different ways to store this information. After a bit of research, we decided that the best way was to use a backend as a service in order to avoid creating ourselves the complete backend of the game. For that we have looked for different services. We considered using GameSparks and Playfab which are two game backend services that several games use. First of all, we wanted to use Playfab because they grant us access earlier. But after some time, we were not able to connect Playfab and Unreal Engine. After that we email GameSparks with the hope of getting an educational license and use it service and after a week or so we finally got the license. Once we got the license we create a new Unreal Project in order to stablish the connection with the service. In this project we achieve user creation and a log in system which our users can use. We did it at the beginning because we knew that we would need this license and we have no time to waste.

But why we have not merged this project in our final one? The reason is because at the beginning it has no sense having users without having a playable game. That is why we focused on making a playable game rather than having users and skip things such as artificial intelligence, which are more important.

So, once we had the login system done we just need to store information of the player but as we already said there was not any information at this time, so we decided not continuing with GameSpark at least until we have useful information.

## Main Menu

When talking about User Interface, one of the features we wanted to include was a Main Menu. The screen you first see when launching the game, and lets you set the different options before starting to play the game itself.

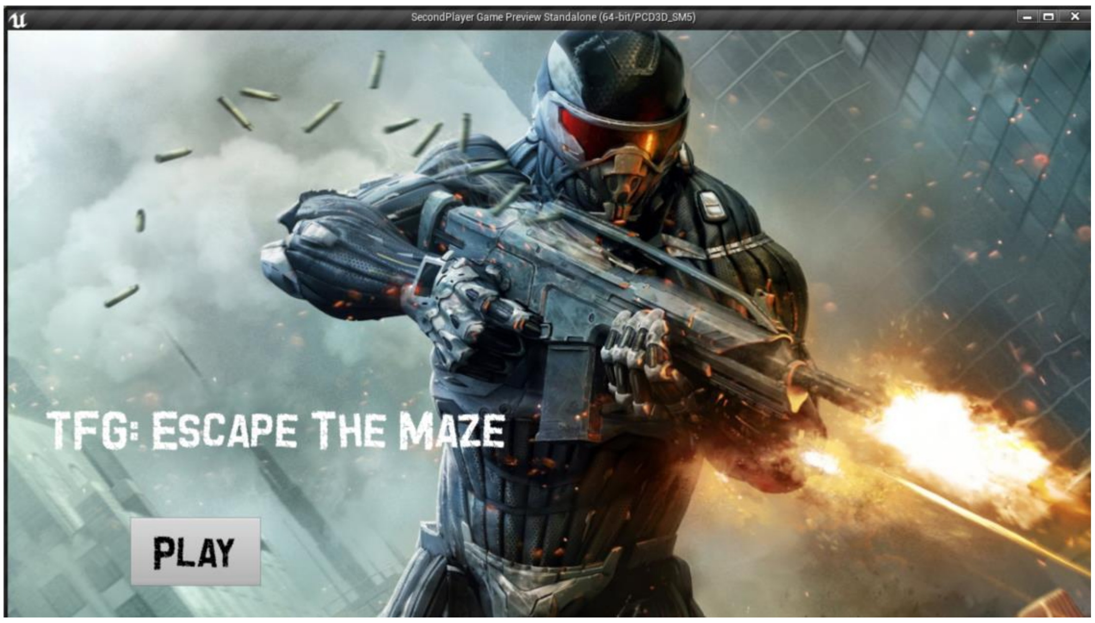


*Figure 21: Main menu*

As you can see in the image, the Main Menu was a simple photo with the Title and the button "play". This is the basic Menu type, when pressing the button, the game begins and redirects you to the corresponding screen. The logic is quite simple as well, "onClicked" function opens the level you want to load.

This worked perfectly, the problem started when we started joining both players and creating a game with a dedicated server. The play button no longer has to open the map but now, it has to connect to the server, get the IP, wait for another player to join the game, start with a concrete character, etc. The extra functionality that the Main Menu gave us compared with the load of work was not positive, so following the Agile methodology we decided to include it later, if we had time, as we already knew how to create the Image, the Button and add a bit of logic to it (at the end the purpose of the project is to learn all the characteristics of Unreal Engine, not necessarily getting deeper on them). Finally, we had a lot of extra problems (that we explained on planning) and we decided to not include a Main menu and when launching the App start right with the game.

## Safe zone to spawn

One of the ideas we had to make the game more intuitive was a safe zone spawner indicator. What this means is a green circle (or red) around your view, but projected to the ground that tells you if you can spawn that Ability at that point or if you cannot.

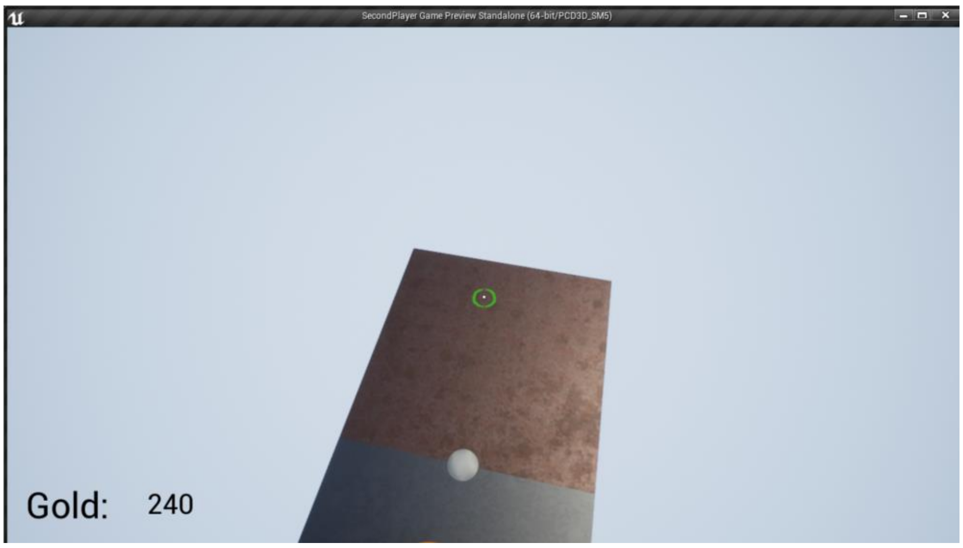


*Figure 22: Safe Zone to Spawn*

```
DrawDebugCircle(this->GetWorld(), EndPoint, 20.0f, 200, FColor::Green, false, -1.0f,
0, thickness, yAxis3,yAxis2, false);
```

This is the code we used to show the circle that will let the player know if he is going to spawn the Ability or not.

When raycasting, we can obtain the object that first impacts with that line, and then get its rotation so the circle always is over the surface of that object.

This worked perfectly and we thought it was a great inclusion for our project, but the problem was the same. When we tried to deploy our application on a sever and start to join both characters and make the game multiplayer, the debug circle just disappeared and stop showing to the players. For this reason, we focused on make the multiplayer work and when we ended with that we discovered that the debug circle will not work in a multiplayer game, so we finally decided to not include it to the final project.

## Deviations

As we already said, some deviations have happened during this project. Some task has been moved and others have been aborted. The project duration was initially of 4 months and has changed to 6 months due to we have changed the defence to October. The implementation phase was slower than what we initially though. So, let's explain all these deviations one by one and the reason of it.

### Calendar deviation

The project has had a duration of 6 months due to some inconvenience that appeared during the development of the project. The main problem was the lack of knowledge we have about how Unreal Engine work and after that the lack of awareness about how multiplayer games work. All of these "problems" have caused that we had to invest two extra months into learning how this work and some of the initial task have been cancelled due to the lack of time.

### Task description

Even the problems we had, the tasks are still the same as they were initially.

- Project planning: this part has not changed since it was the first part and it went good.
- Software analysis and get in touch: This part was initially good but once we changed to the software implementation part we realised that there were some key concepts we did not know about. This is due to we have done the project mostly in C++ but the amount of information about Unreal Engine in C++ is very short and most of the time we get stuck for very long time about how to do a determinate thing. Therefore, because of this problem we have to go back to this task and spend more time in order to get in touch with Unreal Engine.
- Software implementation: This part has changed; some tasks have been swapped between us and some of them have been aborted.
- Testing: This is a part we have been doing during all the implementation part but we consider we still need much more time to test everything because now there are still bugs that need to be fixed but we have no time.
- Defence of the project: This part has no changed at all. We just changed the day but the amount of time stablished for this task is going to be the same.

## Swapped task

When we were trying to solve problems, the lack of information could get you trying to do a single thing for a long time that is what happened to Guille with the multiplayer task. Not being able to do it could blind us when finding the best way to approach it. So, there are features that initially were for Guillem that ended to Felix (the multiplayer part was one of the hardest things to implement), and viceversa (the gold system). We always tried to have the same number of tasks to work on even when one helped the other with some difficult issues.

## Aborted tasks

Some of the tasks that we initially said we were going to do has been cancelled due to time issue. After prioritizing all the task and knowing what we have left, we decided that the Database part which can be a different module of the game was going to take us a lot of time to do it properly. So that we have not stored any information at any database.
The auto generated maze was another aspect we had no time to do it properly because although we have generated some procedural level it was not good enough to be considered a maze, and it leads us to complicated algorithms we do not need to spend time doing.

The reason of these tasks being aborted is because there were some important aspects that we initially did not considered that would took us as many time as it finally took. So, since we are seeing the viability of developing a videogame with Unreal Engine, things as the Artificial intelligence or the Ability system have caused us to explore different alternatives about how to do these things. This means that we spend more time in these tasks and this has caused to be out of time for doing some other things which we considered initially.

## Main cause of deviations

What is really the thing that has caused these deviations in the project? We can say that the lack of knowledge we have with this Engine was the main cause of all these deviations that have appeared. That is also because we thought that the amount of documentation we would have would be much higher because we have been struggling for some time in order to find something helpful for us.

## Deviation costs

These deviations have caused an increase of the costs and here we are going to recalculate the costs again.

Initially our project has this cost.

| Concept | Estimated cost |
|---|---|
| **Direct costs** | 11710€ |
| **Indirect costs** | 1900€ |
| **Subtotal** | 13610€ |
| **Unforeseen contingencies(10%)** | 1361€ |
| **Total** | **14971€** |

*Table 11: Project costs*

The part of the costs that has changed are the direct costs because we as software developers have spent more our since we have two more months to do the project. As we are software developers our price per hour is 15€ per hour. Considering all this information we just need to add the hours we did as software developers and add this price to the human resources costs and at the end to the project cost. Therefore, considering we have been July and September working 5 hours / day it has a cost of:

5 hour / day X 20 days X 2 months = 200 hours X 15€ / hour = 3000€

So, we can say that our project cost has **increased 3000€ now.**

**Final project costs**

| Concept | Estimated cost |
|---|---|
| **Direct costs** | 14710€ |
| **Indirect costs** | 1900€ |
| **Subtotal** | 13610€ |
| **Unforeseen contingencies(10%)** | 1361€ |
| **Total** | **17971€** |

*Table 12: Final project costs*

## Future work

Here we would like to explain what we would do if we have more time available in this project. The first thing would be to test everything properly because as we already said there are some bugs in the game now. Once these bugs have been solved, we would like to finish the tasks we initially said like the database or the level system, which we think, is interesting for any game. We would also add new abilities to each player in order to make the game more interesting for the players because the base, which is the ability system or weapon system, is already done so the game is very easy to expand. Another possible idea is to expand the game 2vs2 for example or add NxN number of payers but this idea has to be thought properly because although the architecture we have done support different players now the game might not be as fun.

We do not want to forget about adding the things that have not been included in the final project like the login system which let us having different users and therefore save information for every player. So, including GameSpark backend would be another thing we would like to do.

## Knowledge integration

During the different steps we have done, we tried to apply the knowledge we acquired during the degree. Some key aspects we have been working on so far are the project planning, where we have set what we have to do at every moment and we priories everything according to project needs. Another thing we considered important is the code extensibility, it means to allow our game to easily expand it in different kind of ways. In order to do that we have studied unreal architecture and tried to follow it every time without breaking it.
One example of an interesting thing we have done is that now you can play 1vs1 but thanks to how we have done it would be very easy to expand these to a team game where you have to play with different people in order to win.
Another important aspect that occurs in most of companies and we could have in this project is also the work in a group where you are not the only developer and you have to make every decision considering that other people has to understand your code and use it. Moreover, this might seem a non-important thing but we have made some decisions in order to make everything easy to understand for anyone new. One example of this is the name convention we have used, so if for example we create a Blueprint of anything it will be called Name_BP in order to know this object is a blueprint. These small things at the end of a project can help so much to any new developer that joins the team or even yourself if you see your code after some time.

## Identification of laws and regulations

One of the laws we have to take into account is the PEGI. The Pan-European Game Information (PEGI) is an age rating system that was established to help European parents make informed decisions on buying video games. It started in 2003 and this system applies into 25 countries with no relation with Europe. This system includes two different ways of classifying video games the first one is the age classifier and the second one is the content classifier.

Age classifier:

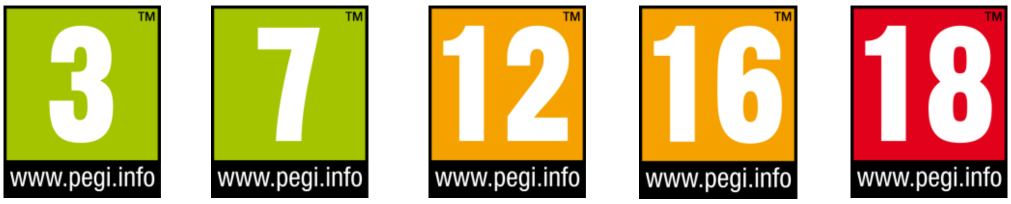


*Figure 19: Age classifiers*

Content classifier:

Violence: the game may contain scenes where people get injured or killed, normally due to weapons. It also may include blood.

*Figure 230: Violence icon*

Bad language: May contain profanity and all manner of slurs, insults.

*Figure 21: Bad language icon*

Fear / Horror: May contain scenes and plot elements too disturbing, overly suspenseful, or frightening to younger/sensitive players.

*Figure 22: Fear/Horror icon*

Sex: Depending on the age category, the game may contain scenes or references to nudity.

*Figure 23: Sex icon*

Figure 24: Drugs icon

Drugs: the game may contain references to or consumption of alcohol, tobacco or illegal drugs.

Figure 25: Discrimination icon

Discrimination: May contain scenes, behaviour, or references to cruelty or harassment to a group of specific people based on race, religion, ethnicity, gender, ability, or sexual identity/preferences.

Figure 26: Gambling icon

Gambling: May contain elements that encourage or teach gambling.

Figure 24: Online icon

Online: Descriptor was discontinued by PEGI in 2015, with the majority of present-day games and all consoles allowing for online interactions.

In order to obtain a rating for any piece of interactive software, the applicant submits the game with other supporting materials and completes a content declaration, all of which is evaluated by an independent administrator called the (NICAM). As our project is experimental what we have done is look though PEGI questionnaire and try to guess ourselves what our game would be. It is clearly violence since we use weapons and kill other human characters. Moreover, it would be +18 according to questionnaire description due to we can kill characters without need and this condition is enough to make it +18. Therefore, our cover would have those icons: Netherlands Institute for the Classification of Audiovisual Media (NICAM). As our project is experimental what we have done is look though PEGI questionnaire and try to guess ourselves what our game would be. It is clearly violence since we use weapons and kill other human characters. In addition, it would be +18 according to questionnaire description due to we can kill characters without need and this condition is enough to make it +18. Therefore, our cover would have those icons:

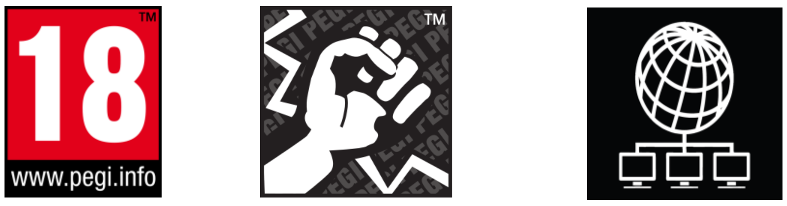


*Figure 25: Our game PEGI classification*

## Unreal Engine royalty payment

Another important aspect we have to consider that we already mentioned through the documentation we have done is that if we ever want to sell our game Epic Games, which is Unreal Engine owners, would earn the 5% of the product revenue after the first $3,000 per game per calendar quarter from commercial products. Whilst in Unity3D if you want to sell the product you need to get a certain version and the available ones are the Plus version which costs 35$ / month or the Pro version which cost 125$ / month.

# Conclusions

During the development of the project, we have tried to always look different alternatives to approach problems and we always tried to choose the best according on what we wanted to do. We have seen that Unreal Engine is a powerful engine but we also have seen that doing complex games is not as easy as we thought. We wanted to follow all the indications that we have learnt during our studies, such as the architecture of the game, the usability of it, the type of server, etc. Therefore, a lot of this decisions were dictated by these indications, and even that the solution wasn't the easier one to get the game working, it was the ones we thought that were the best.

This does not mean that we have not considered all the options that we were able to find in the different tutorial or forums, but even the experts said that some options were still not optimized when using Unreal or that there was very few information that is not even worth the try of implementing it that way.

Things like networking were an unknown world for us at the beginning but still now, we are not fully aware of how it works because we have found that debugging a networking game is complicated. Every single case is very complex and needs a lot of previous knowledge and previous experience at that field that we did not have. Some of the things we struggled most could have been "easily" solved if we had an expert or some guide that could lead us to the correct solution, but not having the experts' help, made us got stuck with the same problem and not being able to advance faster in the project, as that were priority tasks.

Another aspect we have been struggling with is the velocity of compiling C++ code. Sometimes is really slow and Unreal Engine and Visual Studio does not work always well. Having around 30-60 sec of compilation every time you want to test every single line of code can be annoying and not productive, as you lose focus while waiting and might take you the double or triple time to get the tasks done. Sometimes the editor crashes and you must restart the engine, which is annoying and also a big waste of time.

Despite of the problems we have had we cannot forget that the objective of the project is to tell our company about the viability of developing a videogame with Unreal Engine. We can clearly say that the answer is: It is viable to do the game we wanted but we will need experts in key areas such as networking. Previous experience in other game engines is not enough to move around Unreal with facility, so hiring a Senior developer that has worked several years with Unreal is one of the bests ideas you can have, as he can lead your team to better solutions and easier way to learn how to move around this game engine.

Moreover, the developers would need to spend at least a month or so learning how to use properly Unreal Engine before starting the game for the company. Doing a 1-month course about Unreal is worth the time/money, and will help the project to run faster for sure. If they have no experience as it happened to us, it would be hard to begin because as we already mentioned there is no information about many things through the internet, especially in C++. Most of the things that there are explained in Internet are in Blueprints and easy examples, that will not help if you plan to develop (as it might be the case) a complex video game and using C++ (because C++ gives you more options if you fully know how to use its potential).

## References

[1] Cedric_Exi_Neukirchen, “UE4_networking_Compendium”.

[2] "This engine is dominating the gaming industry right now - The Next Web." 24 mar.. 2016, https://thenextweb.com/gaming/2016/03/24/engine-dominating-gaming-industry-right-now/.

[3] "What engines do game companies use when programming games ...." https://www.quora.com/What-engines-do-game-companies-use-when-programming-games

[4] "Why are AAA video game developers making their own engines ...." https://www.quora.com/Why-are-AAA-video-game-developers-making-their-own-engines.

[5] "Unity - Fast Facts." https://unity3d.com/es/public-relations.

[6] "Unreal Engine 4 Documentation." https://docs.unrealengine.com/

[7] "Unity - Manual." https://docs.unity3d.com/.

[8] "Unity 5 vs Unreal Engine 4 – Create 3D Games." 7 sept.. 2015, https://create3dgames.wordpress.com/2015/09/07/unity-5-vs-unreal-engine-4/.

[9] "Software Engineer Salary (Spain) - PayScale."

http://www.payscale.com/research/ES/Job=Software_Engineer/Salary.

[10] “UE4 AnswetHub” - https://answers.unrealengine.com/index.html

[11] “The Unreal Engine Developer Course - Learn C++ & Make Games” www.udemy.com

[12] “Multiplayer questions” https://forums.unrealengine.com/

[13] “Networking videos” https://www.youtube.com/watch?v=EGnMgeeECwo

[14]“Networking”

https://www.reddit.com/r/unrealengine/comments/5jsavj/new_to_unreal_multiplayer/